PRACTICAL TYPE INFERENCE FOR FIRST-CLASS POLYMORPHISM

Dimitrios Vytiniotis

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Stephanie Weirich
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

in memory of Antonios Moutafis (1920 - 2005)

Acknowledgments

First of all, I would like to thank my family, Kiki, Haris, and Antonis, for their invaluable help during my studies. You have given me inspiration and security to pursue my goals. Ioanna Kourti has made me smile and provided support and motivation to complete this dissertation.

I would like to thank my advisor, Stephanie Weirich, for working with me, guiding me, teaching me important lessons about research and academic integrity, and for being patient with me. I feel grateful to Simon Peyton Jones for working with me, teaching me how to communicate my ideas, and for insisting on clarity and practical thinking. Many of the ideas in this work were developed in several short visits to Cambridge during exhausting, but immensely productive, sessions on Simon's whiteboard. Benjamin Pierce, Steve Zdancewic, and Val Tannen were always willing to give advice, to provide feedback on my dissertation, and to chat about anything—from type theory to art. Didier Rémy and Daan Leijen have helped me understand type inference better and were always happy to get involved in deep technical arguments. I am thankful to past and present members of Penn PLClub for numerous discussions and for sharing offices with me. I feel grateful to Mike Felker for helping me to submit this dissertation. I would also like to thank those who have introduced me to Computer Science and PL, Nikos Papaspyrou and Stathis Zachos.

My good friends Nate Foster, Vassilis Koutavas, and Polyvios Pratikakis have provided invaluable support in research and life. Last, but not least, during the past years many people have been companions and made my life in Philadelphia happier and easier: Ann-Katherine Carton, Thanassis Geromichalos, Dina Zhabinskaya, Kostas Danas, Grigoris Karvounarakis, Manolis Galenianos, Eva Zacharaki, Lila Fassa, Dimitris Theodorakatos, Colin Blundell, TJ Green, Pavol Černý. Thank you!

ABSTRACT

PRACTICAL TYPE INFERENCE FOR FIRST-CLASS POLYMORPHISM

Dimitrios Vytiniotis

Stephanie Weirich

Type inference is a key component of modern statically typed programming languages. It allows programmers to omit many—and in some cases all—type annotations from programs. Type inference becomes particularly interesting theoretically and from an implementation standpoint in the presence of polymorphism. Hindley-Damas-Milner type inference, underlying modern functional programming languages with `let`-bound polymorphism, allows for full type reconstruction for programs with types that (i) only contain top-level quantifiers, and (ii) can only be instantiated with quantifier-free types. As soon as one attempts to overcome these barriers to allow *higher-rank* or *impredicative* polymorphism full type reconstruction becomes either undecidable or non-modular because many typeable programs no longer possess principal types. The only hope then for modular and decidable type inference for first-class polymorphism arises from exploiting type annotations to *guide* the type checker in a mixture of type reconstruction and type checking.

The first contribution of this dissertation is a formal study of various theoretical properties of annotation-guided inference for predicative higher-rank types. The second contribution is the presentation of a type system and inference algorithm that lift both Damas-Milner restrictions, allowing impredicative higher-rank types.

A central claim of this dissertation is that annotation-guided type inference for first-class polymorphism is compatible with intuitive specifications that do not have to be more complicated than the familiar Damas-Milner type system.

Contents

List of Figures

Chapter 1

The problem

Type inference is a key component of statically typed functional programming languages such as ML [36, 33] and Haskell [18]. Type inference allows programmers to omit type annotations where types are "obvious" for the type checker to figure out, so that writing type-safe programs requires very little or no type documentation. The central theme of this dissertation is extending type inference for functional programming languages with `let`-bound polymorphism in the presence of polymorphic types that go beyond the type structure of the Damas-Milner type system [35, 5]—the basis of ML and Haskell.

1.1 The nature of type inference

A design for type inference typically involves two components: The first component is an *algorithm* that infers types for programs. The second component is a high-level type-safe *specification* usually presented as an inductively defined typing relation. A design for type inference has to meet a series of requirements, which we outline below.

Implementability A typical type system is already implementable in the sense that it specifies a recursively enumerable set. But in our setting we are interested in having a *decidable* algorithmic

implementation that satisfies a *soundness* and a *completeness* property. Soundness states that, if the algorithmic implementation can infer a type for a program then this program should be typeable in the type system specification. Completeness is a much stronger property: it asserts that the algorithm can infer types for *all* programs that are typeable in the high-level specification. Together these properties ensure that the algorithm accepts exactly the same set of programs that are typeable in the specification. The decidability requirement can sometimes be relaxed when the algorithm terminates and is fast in most common and well-specified cases. For instance, Pfenning's partial polymorphic type inference work [43] gives a sound and complete type inference algorithm that relies on higher-order unification, which is undecidable but terminates and is fast in many common cases.

Expressiveness Expressiveness can be determined by setting a "gold standard" programming calculus that suits our programming needs, and making sure that it can be embedded in a semantics-preserving way in the language for which we are designing type inference. In the case of polymorphic type inference, which is the topic of this dissertation, a suitable criterion for expressiveness is whether all programs of the polymorphic λ-calculus (System F) [7, 8, 9] can be typed (perhaps with the addition of semantically neutral type annotations) in the language for which we design type inference.

Simplicity On one hand, a type system specification, pragmatically, is targeting programmers. Therefore, it has to be compact (i.e. it can be specified with only a few inductive relations) and conceptually simple, to allow programmers to easily reason about typeability. Moreover, the annotations that programmers may have to add to programs to recover the programming power of their "gold standard" programming language should not be overwhelming and their placement should be predictable. On the other hand, a type inference algorithm is also targeting programmers—those that undertake the task of implementing it. As such, it should not be unreasonably complicated to get right and verify, especially when type inference for a particular feature of the language (e.g. polymorphism) has to interact with type inference for other potentially complex features (e.g. type classes [10], or guarded recursive datatypes [63, 4]).

Modularity Modularity states that expressions should be typed by only examining their subexpressions. Hence, a function definition should be typed only by examining the definition itself and not by having to determine all its possible use sites. For example, to infer a type for an ML `let`-bound definition, it should suffice to infer a type for the `let`-bound definition *once* and use that type throughout the scope of the definition.

Robustness Robustness of type inference asserts that typeability is preserved under small program transformations, such as reordering function arguments, or inlining / expanding `let`-bound definitions. Specifying which transformations break or preserve typeability is an important aspect of the metatheory of a type inference system.

Inertia requirements Type inference that addresses new features should yield type systems that extend previous type inference designs—to the extent that the previous designs are sensible. This means that programs previously typeable in the "base" language, such as library or legacy code, should continue to be typeable in the "extension". A different desirable feature is *conservativity* of the extension: Programs typeable in the extension, that do not use any of the features of the extension, should be typeable in the base system.

1.2 This dissertation in context: polymorphic type inference

We start by presenting the historical context of this dissertation and the ideas that most closely affected this work; discussion about other related research can be found in Chapter 2 and Chapter 8.

For the past thirty years the Damas-Milner [5] type system has been used as the basis for the specification of type inference for ML variations that support polymorphism. The Damas-Milner type system admits a very simple algorithm, to which we will refer as the Hindley-Damas-Milner [12, 35, 5] algorithm[1], based on first-order unification [50]. The Hindley-Damas-Milner algorithm can infer polymorphic types without requiring any type annotations. In order to ensure this remarkable

[1] Essentially the same algorithm appears in the work of Hindley and in the work of Damas and Milner. Hindley showed the soundness of the algorithm, whereas completeness with respect to the Damas-Milner type system was shown later by Damas and Milner.

property, the syntax of types is restricted to those that may contain universal quantifiers only at top level, such as $\forall ab\,.\,a \to b \to b$, and instantiation of polymorphic types may use only monomorphic types (*monotypes*), such as `Int`, or $c \to c$, but not $\forall a\,.\,a \to a$.

For the rest of this document we will distinguish between Damas-Milner types (types permitting only top-level quantification) and *rich* types (types with $\forall$ quantifiers under type constructors). For example, `Int` $\to$ `Int` and $\forall a\,.\,a \to a$ are Damas-Milner types; but `Int` $\to [\forall a\,.\,a \to a]$ and $\forall a\,.(\forall b\,.\,b) \to [a]$ are rich types.

Importantly, the Hindley-Damas-Milner algorithm is able to infer the best, or *principal*, type that the Damas-Milner type system can possibly assign to a program—such that all other Damas-Milner types for the same program can be assigned to the program by attaching to the derivation that yields the best type a sequence of type instantiations and generalizations.

The existence of principal types is important for type inference modularity. Consider the following program (we use Haskell syntax throughout for code fragments):

```
let foo = \x -> \y -> x
in (foo 3 4, foo True 'c')
```

The function `foo` can be assigned type `Int` $\to$ `Int` $\to$ `Int` or `Bool` $\to$ `Char` $\to$ `Bool` or many others, but the Hindley-Damas-Milner algorithm will infer type $\forall ab\,.\,a \to b \to a$ for `foo`, and both the two previous types can follow from $\forall ab\,.\,a \to b \to a$ by appropriate monomorphic instantiations of the quantified variables a and b. In the jargon, both the two specialized types are "instances" of $\forall ab\,.\,a \to b \to a$, an idea that we express with a *subsumption*, or *type instance*, relation $\leq$. For example we have:

$$\begin{aligned}
\forall ab\,.\,a \to b \to a &\leq \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
\forall ab\,.\,a \to b \to a &\leq \texttt{Bool} \to \texttt{Int} \to \texttt{Bool} \\
\forall ab\,.\,a \to b \to a &\leq \forall a\,.\,a \to a \to a \\
\forall ab\,.\,a \to b \to a &\leq \forall a\,.\,a \to \texttt{Int} \to a
\end{aligned}$$

In all the above cases, we say that $\forall ab\,.\,a \to b \to a$ is *more polymorphic than* or *more general than*

any type to the right of $\leq$. Notice that, if an expression can be assigned a type on the left of $\leq$ then the same expression can be assigned the type on the right via some type instantiations and generalizations. Additionally, any of the more specialized types will not be appropriate for typing the whole body of the `let`-bound definition `(foo 3 4, foo True 'c')`. Since the Hindley-Damas-Milner algorithm infers principal types, type annotations are not of any use to type inference, though it is handy for a language to support them for documentation purposes and improvement of type error reporting [28].

Predicativity and impredicativity Because of the restriction to monomorphic instantiation, the Damas-Milner type system is characterized in the type inference jargon as a *predicative* type system. Our usage of the term is related to but should not be confused with the more general notions of predicativity and impredicativity found, for instance, in set theory [62] or type theory [7]. In those settings predicativity refers to being able to instantiate a quantified type with a set (or type) that only has a lower rank (or belongs in a lower universe) than the type being instantiated. For the rest of this document, however, we will follow the type inference jargon and we will refer to predicativity as the ability to instantiate only with quantifier-free types.

1.2.1 Beyond Damas-Milner: predicative higher-rank types

Though the Damas-Milner restrictions on the type structure (quantifiers only top-level, predicative instantiation) allow most of everyday programming, in certain cases one needs to go beyond these restrictions [51]. For example, in the context of Haskell the encapsulated state monad transformer `ST` [24] requires a function `runST` with type:

```
runST :: forall a. (forall s. ST s a) -> a
```

Such a type is no longer a Damas-Milner type as it contains nested polymorphism under the top-level quantifiers—we call types that contain arbitrary polymorphism nested at the left of arrow types, types of *higher-rank*. The first striking observation about type inference for higher-rank types is that it becomes unclear how a type inference algorithm should actually be implemented—even when one

preserves the other requirement of Damas-Milner, monomorphic instantiation. For example consider the program below:

```
f get = (get 3, get True)
```

From the code, it is clear that the type of `get` must be polymorphic (a *polytype*), since we must be able to apply `get` both to `True` and `3`. But what is the exact type of it? And what is the exact type of `f`? Both types $(\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$ and $(\forall a\,.\,a \to \texttt{Int}) \to (\texttt{Int}, \texttt{Int})$ are valid types for `f`, but there exists no *principal* type that can be assigned to `f` such that all others follow from it by a sequence of type instantiations and generalizations. This immediately threatens modularity of type inference, as the definition of `f` may be in a context that requires a different type than the one a hypothetical algorithm would choose. Consequently, in the absence of principal types the only hope to type a program such as `let f get = (get 3, get True) in ...` would be by inlining the definition of `f`, so that it can be assigned different, incomparable types in each of its use sites: there is no single type that we may assign to `f` and use throughout its scope.

Even worse, type inference for the higher-rank System F is undecidable [61]. The proof of undecidability is a reduction from the undecidable semi-unification problem [21] and relies on a function argument being assigned a polymorphic type that has to be instantiated (predicatively) at two different types in the body of the abstraction.

Type annotations to the rescue The above difficulties in inferring principal types (or inferring any types at all) for polymorphic programs that use higher-rank types suggest that we should abandon the goal of unaided type inference, at least for programs that make use of higher-rank types, and instead require the programmer to supply some type annotations to guide type inference. For example, there should be no problem for typing the `f` function above if an annotation is provided on its argument `get`:

```
f (get::(forall a. a -> a)) = (get 3, get True)
```

In a ground-breaking paper, Odersky and Läufer [39] showed that type inference that involves predicative higher-rank types is possible, by annotating all function arguments that must be polymorphic. Such annotations no longer serve only documentation purposes, as in Damas-Milner, but

rather *guide* the type inference engine by giving it all the polymorphic information that it cannot automatically infer. Still, while these type annotations recover programs with higher-rank types, Damas-Milner-typeable programs remain typeable without annotations.

Polymorphic subsumption Types that contain nested polymorphism may induce new type instance relations, that are not based solely on type instantiations and generalizations. For example if an argument of a function is annotated with type $\texttt{Int} \to \forall a\,.\,a \to a$ while the function itself requires type $\forall a\,.\,\texttt{Int} \to a \to a$ the application should be typeable—intuitively, if an implicitly typed System F expression e has type $\texttt{Int} \to \forall a\,.\,a \to a$, then if we wrap e in appropriate abstractions, we can retype it to have type $\forall a\,.\,\texttt{Int} \to a \to a$, and vice-versa. Hence, in the presence of nested polymorphism under type constructors, subsumption can be generalized to include derivations such as $\texttt{Int} \to \forall a\,.\,a \to a \leq \forall a\,.\,\texttt{Int} \to a \to a$. It is also type safe to allow it to be contra-variant in argument types and co-variant in result types, as is standard for subtyping relations.

Bidirectionality One problem with the Odersky-Läufer original approach is that the annotation burden is quite heavy, and often the context in which an expression is placed can make a type annotation redundant. Consider again our example with an annotation on `f` (as is common in Haskell top-level definitions):

```
f :: (forall a. a -> a) -> (Int,Bool)
f (get::(forall a. a -> a)) = (get 3, get True)
```

The type signature for `f` makes the type of `get` clear ($\forall a\,.\,a \to a$), without explicitly annotating the latter. In fact, one would not want to annotate `get` *and* provide a separate type signature for `f`; for example, if `f` had multiple defining clauses one would not be happy to repeat the annotation on every defining clause.

The standard way to express this idea is by *bidirectional propagation of type annotations*, which in turn originates in *local type inference* [45]. In this approach there typically exist two typing judgement forms instead of one: $\Gamma \vdash_{\Uparrow} e : \sigma$ and $\Gamma \vdash_{\Downarrow} e : \sigma$. The former means "infer the type of e", while the latter means "check that e has the known type σ". Operationally, we think of σ as

an output of the first judgement, but as an input to the second. These two judgement forms are interwoven in way that allows contextual information propagation.

The separation between checking mode and inference mode allows us to assign higher-rank types to functions while still avoiding guessing polymorphic arguments. The idea is that the bound variable of a λ-abstraction can be assigned a polytype if we are in checking mode and hence all polymorphic information about that type is known but can only be assigned a monomorphic type in inference mode. When the argument is already annotated there is no problem for type inference:

$$\frac{\Gamma, x{:}\sigma_1 \vdash_{\Downarrow} e : \sigma_2}{\Gamma \vdash_{\Downarrow} \lambda x\,.\,e : \sigma_1 \to \sigma_2}\ \textsc{abs}1 \qquad \frac{\Gamma, x{:}\tau \vdash_{\Uparrow} e : \sigma}{\Gamma \vdash_{\Uparrow} \lambda x\,.\,e : \tau \to \sigma}\ \textsc{abs}2 \qquad \frac{\Gamma, x{:}\sigma_1 \vdash_{\delta} e : \sigma_2}{\Gamma \vdash_{\delta} \lambda(x::\sigma_1)\,.\,e : \sigma_1 \to \sigma_2}\ \textsc{abs}3$$

Reading rule ABS2 from bottom to top, to infer the type of an abstraction, we need to guess the argument type τ to add to the context and infer the type of the body. However, in ABS1, because we are checking that the λ-abstraction has a function type, we do not need to guess what type to add to the environment. Rule ABS3 simply shows that when the argument is annotated, we may simply push it in the environment using the annotation type, and infer or check ($\delta = \Uparrow | \Downarrow$) the body of the abstraction.

During type inference, input types for checking mode arise from user type annotations. For example, if the language allows the programmer to annotate a term e with a specified type σ, which we write as $(e::\sigma)$, then we could have the following rule that switches between inference and checking modes.

$$\frac{\Gamma \vdash_{\Downarrow} e : \sigma}{\Gamma \vdash_{\Uparrow} (e::\sigma) : \sigma}\ \textsc{ann}$$

Returning to the example, using local type inference we no longer need both annotations:

```
f :: (forall a. a -> a) -> (Int,Bool)
f get = (get 3, get False)
```

since the abstraction (λ`get`$\ldots$.) that defines `f` can be checked against type $(\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$ and using rule ABS1, `get` is pushed in the environment with type $\forall a\,.\,a \to a$. The body of the abstraction is then easily typeable in the extended environment.

This dissertation in context The Glasgow Haskell Compiler (GHC) included already since the early 2000's an implementation of a bidirectional type inference system for arbitrary-rank types, but its metatheory and its properties remained unexplored. The work on this dissertation fleshed out the technical details of applying local type inference and expressive subsumption relations for predicative higher-rank types, and gave the connections of that system with the original Odersky-Läufer and Damas-Milner type systems (Chapter 3).

1.2.2 Impredicative higher-rank types

Once one starts needing types of higher rank, very quickly one realizes that impredicative instantiation of type variables, that is, the lifting of the second restriction of the Damas-Milner type system, is also desirable. Of course, once one lifts the predicativity restriction and allows higher-rank types one is left with the full type structure of the polymorphic λ-calculus, System F. Here is one example, a slight variation of the example given earlier:

```
data Maybe a = Just a | Nothing


g Nothing    = (0, True)
g (Just get) = (get 3, get True)
```

Here, the polymorphic function is wrapped in a `Maybe` type, an ordinary algebraic data type. The function `g` can be typed with `Maybe` $(\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$. However, to allow such a type, we must allow a type variable—`a`, in the definition of `Maybe`—to be instantiated with a polymorphic type.

A second example comes from Haskell: programmers often with to apply `runST` to an argument `arg` using an infix polymorphic application combinator `app` of type $\forall ab\,.\,(a \to b) \to a \to b$ as follows:

```
... runST `app` arg ...
```

Since `runST` has type $\forall a\,.\,(\forall s\,.\,\texttt{ST}\ s\ a) \to a$ this implies that the quantified variable a of the type of `app` must be instantiated with some type that contains polymorphism.

The lack of principal types affects already the predicative fragment of higher-rank types; but impredicativity leads to new opportunities for lack of principal types—even for ordinary Damas-Milner typeable programs. Consider for example:

```
choose :: forall a. a -> a -> a
id  :: forall a. a -> a

bar = choose id
```

In the previous code fragment, `id` is the polymorphic identity with type $\forall a\,.\,a \to a$, and `choose` is a polymorphic selector with type $\forall a\,.\,a \to a \to a$. What should be the type of `bar`? One possible type would be its Damas-Milner type $\forall b\,.(b \to b) \to (b \to b)$. Another possible type for `bar` may be $(\forall a\,.\,a \to a) \to (\forall a\,.\,a \to a)$, if the quantified variable a in the type of `choose` is allowed to get instantiated to the polymorphic type $\forall a\,.\,a \to a$. However, there exists no single System F type that can be assigned to `bar` such that both types can be derived by a sequence of type instantiations and generalizations. This immediately makes type inference based on System F types non-modular. Furthermore, if the algorithmic implementation were to commit to an arbitrary choice between the two types, completeness would be at stake—there will exist program contexts that require the first type for `bar`, and others that require the second. That is, *unless this ambiguity were resolved by requiring the programmer to use a System F type annotation to indicate which one of the incomparable types he intended to use.*

The fundamental problem here is that System F types are not expressive enough to support modular type checking of `let`-bound definitions. Indeed, if the definition of `bar` were to be inlined in all its use sites, one type or the other could be chosen at every use site. Nevertheless, no System F type could be used to abstract all these common subexpressions in a `let`-bound definition. In other words, if $\mathcal{C}[\cdot]$ is a multi-hole System F program context and $\mathcal{C}[e]$ is typeable, then it is not necessarily true that there exists a System F type σ such that adding $(x{:}\sigma)$ to the typing environment makes $\mathcal{C}[x]$ typeable as well. System F is hence not closed under `let` expansion. By contrast, Damas-Milner *is* closed under such `let` expansions (by picking σ to be the principal type of e).

This observation appears in the core of the ML^F language of Le Botlan and Rémy [26, 25]. Le Botlan and Rémy observe that, in reality, the best type for `bar` above is $\forall(a \geq \forall b.b \to b)\,.\,a \to a$, where the $(a \geq \forall b\,.\,b \to b)$ is an *instance constraint* on the quantified variable a. It merely asserts that whatever type a can get instantiated with (e.g. $b \to b$, $[b] \to [b]$, or $\forall b\,.\,b \to b$), this type has to be an *instance* of $\forall b\,.\,b \to b$, for an appropriate definition of the instance relation. Hence, ML^F proposes to generate and update instance constraints, and moreover quantify over them, as must be done for the type of `bar`. It turns out that to type all System F programs, one only needs annotate those polymorphic function arguments that have to be typed at two or more different types in the function body [27].

This dissertation in context The second important challenge that this dissertation addresses is type inference for higher-rank *and* impredicative polymorphism. In Chapter 5 we propose the FPH type system [59] (acronym standing for *First-class Polymorphism for Haskell*), which supports impredicative higher-rank polymorphism. The FPH type system enjoys robustness of typeability and modular type inference, but, contrary to ML^F, it has a constraint-free declarative specification, which is only a modest variation of the Damas-Milner type system and employs System F types. Unsurprisingly, because FPH uses System F types, it does not enjoy the `let` expansion property. Hence, FPH has to provide a clean specification of which of the incomparable System F types it "prefers" for a `let`-bound definition—and this choice is driven by the requirement for extension of Damas-Milner. It additionally has to provide the ability for programmers to choose which of the other incomparable System F types they wish to use for a definition, via appropriate type annotations. The way to achieve this (and the key idea behind FPH) is the following: whenever the intuitively principal type for an expression is $\forall(a \geq \forall b\,.\,b \to b)\,.\,a \to a$, the specification of FPH—which is constraint-free by design—is only allowed to type the expression with any System F type which results from an appropriate instantiation of a with σ (such that $\forall b\,.\,b \to b \leq \sigma$), but *marking those instantiations that are impredicative* in the type syntax using "boxes". Such a type, for example, may be the type $\boxed{\forall b\,.\,b \to b} \to \boxed{\forall b\,.\,b \to b}$ when a is instantiated with $\boxed{\forall b\,.\,b \to b}$. With the mechanism of boxes in place, specifying in the type system where impredicative instantiations have occurred is quite straightforward: just check if the type of an expression contains boxes or is

box-free.

The path from the work on predicative higher-rank type systems (Chapter 3) to the FPH system passes through a less successful attempt to apply type inference that synthesizes impredicative instantiations locally (i.e. within the premises of a single typing rule) to impredicative polymorphism, Boxy Types [58] (to be described in detail in Chapter 8). The idea behind Boxy Types is the following: use aggressive type annotation propagation so that impredicative instantiations can be synthesized locally. What the Boxy Types experience shows is that, in the case of impredicative polymorphism, synthesizing impredicative instantiations locally threatens the simplicity of the type system specification, its robustness under program transformations, and may result in need for many type annotations. It turns out (Chapter 4) that, in order to lift the predicativity restriction without sacrificing the properties outlined in Chapter 1, one needs to abandon local type argument synthesis and embark to a global type argument synthesis algorithm (Chapters 5 and 6). Global type argument synthesis for impredicative polymorphism is however entirely compatible with bidirectional type annotation propagation (Chapter 7).

1.3 Predicative vs. impredicative higher-rank type systems

The work in this dissertation on predicative higher-rank type systems and the FPH proposal share some characteristics and differ in others. Chapter 7 presents more details, but we outline some similarities and differences below.

There are similarities between the higher-rank predicative type systems and FPH. First, type inference for these type systems relies on type annotations. More specifically, following the central idea behind the Odersky-Läufer type system, both the various predicative higher-rank systems and FPH require that polymorphic function arguments be annotated; however FPH may also require annotations on `let`-bound definitions with types that result from some impredicative instantiation. Second, all type systems that we present in this dissertation type all Damas-Milner-typeable programs with their original types.

However, allowing impredicative instantiations results, for technical reasons that we discuss in Chapter 7, to several differences between FPH and the various predicative higher-rank type inference systems. First, in the predicative higher-rank world the types of expressions can be converted arbitrarily along type instance relations that are contra-variant in function arguments and co-variant in function result types—but are predicative. In contrast, in FPH the types of expressions can be converted only as a result of top-level instantiations and generalizations—but instantiations can be impredicative. As a result, the programmer may sometimes have to η-covert an expression to make it typeable in FPH, while it may be typeable in a higher-rank predicative type system. Second, because of the restriction of type conversions to instantiations and generalizations, the FPH type system can only type a fragment of System F. In contrast, the predicative higher-rank type systems presented in this dissertation can type programs beyond System F (because of the elaborate instance relations that they employ)—that, however, only use predicative instantiations. Third, the low-level implementation of predicative higher-rank types is based on traditional first-order unification, whereas in FPH the unification algorithm has to manipulate types with constraints that express sets of System F types.

1.4 Thesis goal

The overall goal of this dissertation is two-fold. First, to present the formal properties of various type systems for predicative higher-rank polymorphism that are based on the Odersky-Läufer type system, and to formally describe the connection between each other. Second, to substantiate the claim that type inference for impredicative and higher-rank polymorphism can combine the virtues outlined in Section 1.1. The FPH system gives evidence for this claim, as it enjoys sound and complete type inference, has simple typing rules that are very similar to the typing rules of Damas-Milner, is based on the intuitive System F type structure, it extends Damas-Milner, and can type all System F programs with the addition of type annotations on abstractions and `let`-bound definitions with rich types.

1.5 Technical overview

We proceed in Chapter 2 with giving notation, auxiliary definitions, and some background work on type inference for first-class polymorphism. The reader who is familiar with the type inference jargon and with terminology such as Mitchell's type containment relation and the Odersky-Läufer type system may proceed to Chapter 3.

In Chapter 3 we present our work on type inference for higher-rank predicative types: we present type systems for higher-rank types and show the precise connection between each other, the Odersky-Läufer type system, and the traditional Damas-Milner type system. We start with a syntax-directed variation of the Odersky-Läufer type system, and arrive at a bidirectional version that supports propagation of type annotations. We give principal types properties and establish type soundness.

In Chapter 4 we outline our attempt to employ the same idea of local type inference to impredicative polymorphism with Boxy Types (a detailed discussion of Boxy Types will be given in Chapter 8) and discuss the reasons why we have to abandon local type argument synthesis and take the global type argument synthesis approach of FPH.

In Chapter 5 we present the FPH type system and give a detailed account of its properties: we show the expressiveness of FPH by giving a type-directed translation from System F that is economic in type annotations, we discuss robustness under program transformations, and we establish a connection of the basic FPH type system with a syntax-directed specification.

In Chapter 6 we describe the algorithmic implementation of FPH and its properties—proofs of these properties can be found in Appendix A. We give termination, soundness, and completeness results.

In Chapter 7 we present alternative design choices and discussion, focusing on the FPH type system. We elaborate on the reasons that the impredicativity that FPH supports is hard to combine with the forms of subsumptions, as (some impredicative versions of) the relations used in the higher-rank type inference work (Chapter 3).

In Chapter 8 we give a technical overview of recent and closely connected related works on higher-rank and impredicative type inference by classifying them in three coarse categories, (i) works that are based on explicitly marking where impredicative instantiations occur, (ii) works that are based on

local type argument synthesis (including Boxy Types), and (iii) works that are based on global type argument synthesis. Chapter 8 includes a comparative discussion based on simplicity, expressiveness, need for type annotations, and implementation difficulty.

We conclude and give directions for future research in Chapter 9.

Chapter 2

Notation and background

In this chapter we define notation and introduce terminology that is used throughout the document. We additionally present some background work that is connected to the work in this dissertation.

2.1 Notation

The syntax of the calculus that is going to be used throughout is given in Figure 2.1. The syntax of terms is standard and includes variables, abstractions, and applications. Our intention is to specify type inference for languages that do not include explicit type applications and type abstractions, but for this to be possible the term syntax must have provisions for user type annotations. Type annotations are placed either on λ-abstraction arguments, with $\lambda(x\texttt{::}\sigma)\,.\,e$, or on arbitrary expressions, with $(e\texttt{::}\sigma)$. One can imagine different forms of annotations, such as an annotated form of `let`-bound expressions ($\texttt{let}\ x\ \texttt{::}\ \sigma = u\ \texttt{in}\ e$), but we do not include such forms because it is not hard to understand how such forms interact with type inference, once all other bits have been explained.

Environments can be viewed as sets of bindings of the form $(x{:}\sigma)$ and we use the notation $(x{:}\sigma) \in \Gamma$ to denote that the variable x is bound in the environment Γ to type σ.

Term variables	x, y, z			
Expressions	e, u	$::=$	x	
		$\mid$	$\lambda x\,.\,e$	Abstraction
		$\mid$	$\lambda(x::\sigma)\,.\,e$	Annotated abstraction
		$\mid$	$e\ u$	Application
		$\mid$	$\texttt{let}\ x = u\ \texttt{in}\ e$	Let-bound definition
		$\mid$	$(e::\sigma)$	Type annotation
Types	σ	$::=$	$\ldots$	See Figure 2.2
Environments	Γ	$::=$	$\Gamma, (x{:}\sigma) \mid \cdot$	

Figure 2.1: Generic source syntax

Polytypes	σ	$::=$	$\forall\overline{a}\,.\,\rho$	
Rho-types	ρ	$::=$	$\ldots$	Depending on type system
Monotypes	τ	$::=$	$\tau \to \tau \mid a$	
Type variables	$a, b, \ldots$			

Figure 2.2: Generic syntax of types

The syntax of types will be given separately for each type system presented; however in general types will follow the grammar in Figure 2.2. We use the term *polytypes* and the metavariable σ for types that may contain top-level quantifiers. We treat the quantifiers of types as lists of distinct variables, which we write with the notation $\overline{a}$. We often treat σ-types with no quantifiers as ρ-types, that is, we treat $\forall\,.\,\rho$ as equal to ρ. Additionally, we identify α-equivalent types. The *ρ-types* are those that may or may not contain polymorphism nested inside other constructors but not at top level. The precise of ρ-types depends on the particular type system in discussion. Finally *monotypes*, τ, are types that may not contain any quantifiers. For simplicity we do not assume any type or data constructors in the generic syntax. Choosing appropriate syntax for ρ-types leads to different interesting type structures; for example by picking $\rho ::= \tau$ one gets the types of the Damas-Milner type system. On the other hand, by picking $\rho ::= \tau \mid \sigma \to \sigma$ one recovers the types of System F.

We use vector notation for types as well; for example $\overline{\sigma}$ is a list of polytypes. We define $\mathit{ftv}(\sigma)$ to be the set of free type variables of a type σ, and extend the definition to type environments in the obvious way: $\mathit{ftv}(\Gamma) = \bigcup\{\mathit{ftv}(\sigma) \mid (x{:}\sigma) \in \Gamma\}$. We use the notation $[\overline{a \mapsto \sigma}]\sigma$ for the capture-avoiding substitution of type variables $\overline{a}$ by types $\overline{\sigma}$ in σ. For two sets of type variables V_1, V_2, we write

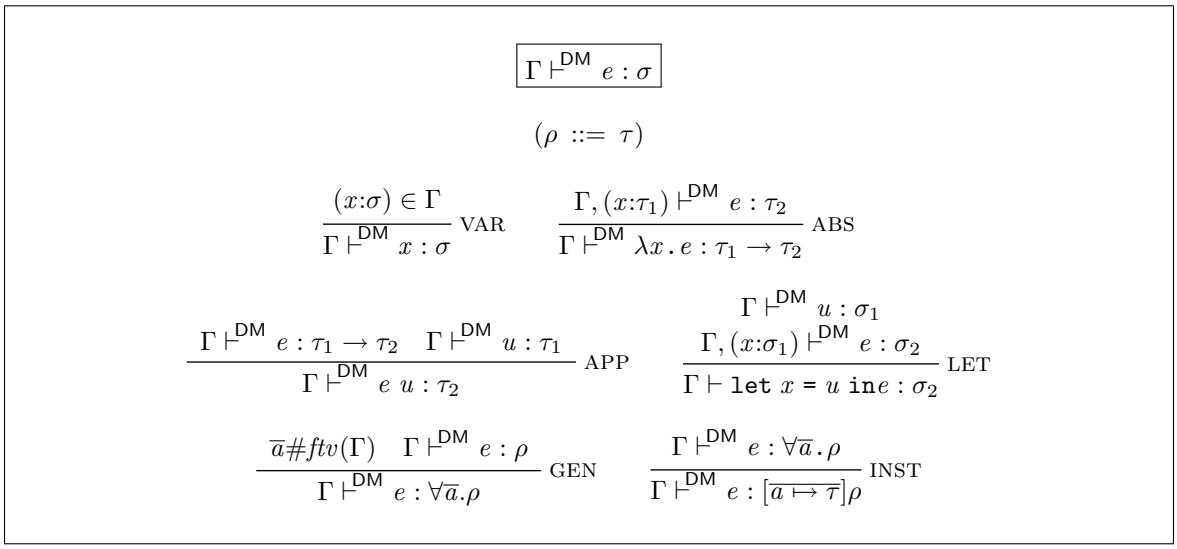

Figure 2.3: The Damas-Milner type system

$V_1 \# V_2$ to mean that $V_1 \cap V_2 = \emptyset$. We write $x \# V$ to mean $\{x\} \# V$.

To discriminate between two or more types of the same syntactic category, such as τ-types, we use subscripts on types, e.g. we write τ_1, τ_2, etc. Note however that primed meta variables (such as τ', ρ', or σ') usually will denote different syntactic categories altogether; for instance, it is not to be assumed that τ and τ' belong in the same syntactic category.

2.2 Damas-Milner type instance and principal types

We now formalize various subsumption (or type instance) relations that were mentioned in the introduction. We assume that the readers are familiar with the Damas-Milner type system. Typeability in that system is given with the judgement $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ in Figure 2.3, which we do not further discuss.

The Damas-Milner instance relation characterizes the retyping of expressions in the type system of Figure 2.3 through a sequence of monomorphic type instantiations and generalizations.

Definition 2.2.1 (Damas-Milner instance)**.** We define $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$ with the following inference

rule:

$$\frac{\overline{b}\#ftv(\forall\overline{a}\,.\,\rho)}{\vdash^{\mathsf{DM}} \forall\overline{a}\,.\,\rho \leq \forall\overline{b}\,.\,[\overline{a \mapsto \tau}]\rho}\ \textsc{shsubs}$$

Rule SHSUBS corresponds to an instantiation of the top-level quantifiers, and a subsequent generalization. If $\Gamma \vdash^{\mathsf{DM}} e : \forall\overline{a}\,.\,\rho$ in Damas-Milner, then it is also derivable that $\Gamma \vdash^{\mathsf{DM}} e : [\overline{a \mapsto \tau}]\rho$ by an instantiation step, and then $\Gamma \vdash^{\mathsf{DM}} e : \forall\overline{b}\,.\,[\overline{a \mapsto \tau}]\rho$ holds by a generalization step. The condition $\overline{b}\#ftv(\forall\overline{a}\,.\,\rho)$ ensures that the generalization step is possible. If this was not the case, it could have been that a $b \in \overline{b}$ also belongs in Γ, in which case generalization over it would be impossible.

An equivalent characterization of $\vdash^{\mathsf{DM}}$ in terms of a simpler set of inference rules, that attempts to do less in each inference rule, is often presented. It can be shown that $\vdash^{\mathsf{DM}}$ can equivalently be defined as the smallest relation closed under the following rules:

$$\frac{\overline{a}\#ftv(\sigma) \quad \vdash^{\mathsf{DM}} \sigma \leq \rho}{\vdash^{\mathsf{DM}} \sigma \leq \forall\overline{a}\,.\,\rho}\ \textsc{skol} \qquad \frac{\vdash^{\mathsf{DM}} [\overline{a \mapsto \tau}]\rho_1 \leq \rho_2}{\vdash^{\mathsf{DM}} \forall\overline{a}\,.\,\rho_1 \leq \rho_2}\ \textsc{spec} \qquad \frac{}{\vdash^{\mathsf{DM}} \rho \leq \rho}\ \textsc{rho}$$

Rule RHO deals with the trivial case of two monotypes in the case of Damas-Milner types (where $\rho ::= \tau$). When quantifiers are involved, to prove that $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$, for any given instantiation of σ_2 we must be able to find an instantiation of σ_1 that makes the two types match. In formal notation, to prove that $\vdash^{\mathsf{DM}} \forall\overline{a}.\rho_1 \leq \forall\overline{b}.\rho_2$ we must prove that for all $\overline{\tau}_b$ there exist $\overline{\tau}_a$ such that $[\overline{a \mapsto \tau_a}]\rho_1 = [\overline{b \mapsto \tau_b}]\rho_2$. To this end, rule SPEC is straightforward: it allows us to instantiate the outermost type variables of σ_1 arbitrarily to match ρ_2. But we need to check that σ_1 can be instantiated by SPEC to match *any* instantiation of σ_2. Suppose we were to instantiate the outermost type variables of σ_2 to arbitrary, completely fresh type constants, called *skolem constants*. If, having done this we can still instantiate σ_1 to match, then we will have shown that indeed σ_1 is at least as polymorphic as σ_2. Interestingly, rule SKOL does not actually instantiate σ_2 with fresh constants; instead, it simply checks that the quantified type variables of σ_2 are fresh with respect to σ_1 (perhaps by implicitly α-renaming σ_2); then these type variables will themselves serve very nicely as skolem constants, so we can vacuously instantiate $\forall\overline{a}.\rho$ with the types $\overline{a}$ to get ρ. That is the reason for the side condition in SKOL, $\overline{a}\#ftv(\sigma)$.

Now that the instance relation of Damas-Milner is defined, we give the theorem that states that

principal types exist for Damas-Milner typeable programs. The theorem was formally established by Damas and Milner [5].

Theorem 2.2.2 (Principal types for Damas-Milner programs)**.** *Assume that* $\rho ::= \tau$*, that is, we are within the Damas-Milner type structure. Then, if* $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ *then there exists a* σ_0 *such that* $\Gamma \vdash^{\mathsf{DM}} e : \sigma_0$ *and for all* σ *such that* $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ *it is the case that* $\vdash^{\mathsf{DM}} \sigma_0 \leq \sigma$*.*

The type σ_0 is the principal type for the expression e in the environment Γ.

2.3 Implicitly typed System F

If the structure of types is augmented such that ρ-types include arbitrary nested polymorphism, and we allow arbitrary impredicative instantiation, we can define the instance relation of System F, which expresses retyping via a sequence of potentially polymorphic type instantiations and subsequent generalizations.

Definition 2.3.1 (System F instance)**.** We define $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_2$ with the following inference rule:

$$\frac{\overline{b} \# ftv(\forall \overline{a} \,.\, \rho)}{\vdash^{\mathsf{F}} \forall \overline{a} \,.\, \rho \leq \forall \overline{b} \,.\, [\overline{a \mapsto \sigma}]\rho} \text{ FSUBS}$$

Notice the characteristic difference of $\vdash^{\mathsf{F}}$ compared to $\vdash^{\mathsf{DM}}$, which is the freedom to instantiate with arbitrary polymorphic types. The relation $\vdash^{\mathsf{F}}$ could also be easily expressed using rules SKOL, SPEC (where the instantiation types can be σ-types), and RHO. The relation $\vdash^{\mathsf{F}}$ is referred to as the System F instance because the typing relation for the implicitly typed System F (where the term syntax is simply the λ-calculus), can be given as an inductive definition that includes a rule that allows a type to be instantiated arbitrarily along $\vdash^{\mathsf{F}}$.

The type system which generalizes Figure 2.3 that (i) allows polymorphic instantiations in rule INST and (ii) allows arbitrary σ-types as the types of functions and function arguments in the abstraction and application rules is the *implicitly typed* System F enhanced with `let`-bound definitions.

Unfortunately, Wells shows that type inference for System F is undecidable [61], and we give below some of the earlier work on addressing the problem.

Finite-rank fragment of System F Some of the early work on type inference for System F is based on detecting rich fragments of System F for which full type reconstruction *is* decidable. We say that a type is of *rank* k if there is a quantifier in the type that appears to the left of k (but no more) $\rightarrow$ constructors in a tree representation of the type. Stratifying System F by the rank of types that are allowed to appear in a given typing derivation has given rise to several results. The rank-k subset of System F consists of all expressions that can be typed using types of rank $\leq k$. Kfoury and Wells show that typeability is decidable for rank ≤ 2, and undecidable for all ranks ≥ 3 [22]. For the rank-2 fragment, the same paper gives a type inference algorithm. This inference algorithm relies on a rewriting relation on terms. We conjecture that undecidability of full type reconstruction carries over to the arbitrary-rank *predicative* System F as well, but we are not aware of such a result.

Partial polymorphic type inference Pfenning's work on the other hand gives an early solution to type inference for full System F based on some form of source annotations. Pfenning shows that even *partial type inference* [43], where only type abstractions and the positions of type applications are known, but not the types of λ-abstraction arguments, for the n-th order polymorphic λ-calculus is equivalent to n-th order unification. This implies that even partial type inference for System F where type abstractions and type applications are marked is undecidable. On the other hand, higher-order unification relies on an algorithm [14] that terminates in most common cases, and hence if one is willing to add partial type information that includes type abstractions and type applications in System F programs, one can have type reconstruction that is undecidable, but effective in practice. Pfenning's work treats `let`-bound expressions effectively as if they are inlined in the bodies of the definitions, in order to avoid the problems arising from the lack of principal types. This may threaten type inference modularity.

To type all System F typeable programs, one actually *does not need* to provide annotations on type abstractions or applications, but rather the more crucial to type reconstruction annotations on polymorphic function arguments. Indeed, the work of Le Botlan and Rémy [27] (and partially this dissertation) shows that one only needs polymorphic function argument annotations (if they are used at two different types in the case of ML^F) to type all System F programs.

However, Pfenning's work offers two advantages. The first is that it seamlessly scales to higher-order

polymorphism. In contrast, most other works (including this work) handle only System F and the treatment of type-level abstractions is considered a separate problem. The second is that, in the presence of effects and operational semantics that is based on an explicitly typed target language, programmer annotations on type abstractions result in better control over the operational behavior of programs. For example, consider a function f of type $\forall a\,.\,a \to \texttt{Int}$ and an expression e that involves effects. For instance, e may be $x := !x + 1\,; \lambda y\,.\,y$. There are two ways (among others) for typing the application $(f\ e)$. If the type checker instantiates f predicatively with $b \to b$ (another possible type for e) then no type-level abstraction has to be placed around e, and one may expect that the effect of e *is* going to be executed. On the other hand, if the type checker instantiates f impredicatively with $\forall a\,.\,a \to a$ (the type of e) then it presumably has to place a type-level abstraction around e, treating it effectively as a value in the explicitly typed target language. But in a call-by-value setting it seems unsatisfactory for the elaboration driven by type inference to cause a delay in the evaluation of e. Pfenning's work gives *the programmer* the control on whether the effect should be executed or not when the target language includes type abstractions and treats them as values. Arguably, this is less of a problem if the operational semantics is lazy and type-erasure-based (as is the case with Haskell and FPH), but this may not always be the case (e.g. it is not the case in some presentations of ML [11]).

2.4 Putting type annotations to work

Since type reconstruction for full System F is undecidable, Odersky and Läufer proposed a type system [39] that only includes predicative instantiation (and hence can be implemented using ordinary unification, as Damas-Milner) and requires annotations on polymorphic function arguments (and hence avoids the undecidability concerns raised by Wells' work).

We now turn our attention to the details of the predicative arbitrary-rank type system proposed by Odersky and Läufer. Figure 2.4 presents the Odersky-Läufer typing relation for our term language, in a non-syntax-directed form.

First, the syntax of ρ-types is: $\rho \ ::= \ \tau \mid \sigma \to \sigma$. Crucially, a polytype may appear in both the

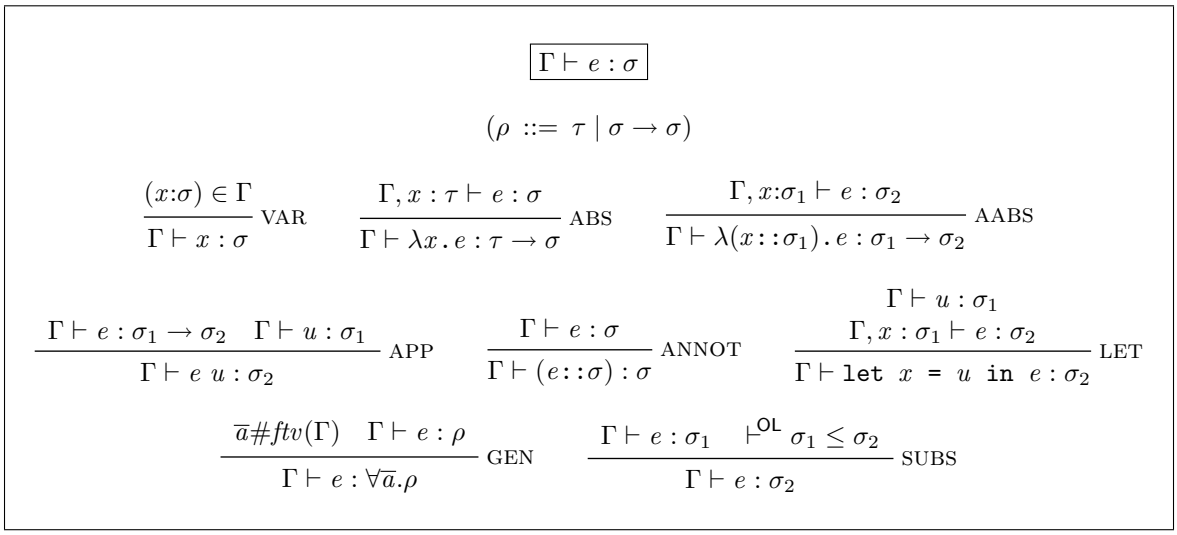

Figure 2.4: The Odersky-Läufer type system

argument and result positions of a function type, and hence polytypes may be of *arbitrary rank*. For annotated λ-abstractions the argument type of such an abstraction can be a σ-type (rule AABS) in contrast to an ordinary, unannotated λ-abstraction whose argument type is a mere monotype, τ (rule ABS). Hence the type system does not guess polymorphic function arguments, but requires type annotations for them. Rules APP, LET, and ANNOT are straightforward. Rule GEN follows the corresponding rule from the Damas-Milner type system, generalizing over type variables that do not appear in the typing environment.

A crucial part of this type system is rule SUBS which allows the retyping of an expression along a special subsumption relation $\vdash^{\mathsf{OL}}$. This rule appears also in alternative presentations of the Damas-Milner type system, where the instance used there is $\vdash^{\mathsf{DM}}$, and in presentations of the implicitly typed System F where the instance used there is $\vdash^{\mathsf{F}}$. We now turn to the definition of $\vdash^{\mathsf{OL}}$.

Definition 2.4.1 (Odersky-Läufer instance)**.** The relation $\vdash^{\mathsf{OL}} \sigma_1 \leq \sigma_2$ is the smallest relation closed under the following inference rules:

$$\frac{\overline{a}\#ftv(\sigma) \quad \vdash^{\mathsf{OL}} \sigma \leq \rho}{\vdash^{\mathsf{OL}} \sigma \leq \forall \overline{a}\,.\,\rho}\ \text{SKOL} \qquad \frac{\vdash^{\mathsf{OL}} [\overline{a \mapsto \tau}]\rho_1 \leq \rho_2}{\vdash^{\mathsf{OL}} \forall \overline{a}\,.\,\rho_1 \leq \rho_2}\ \text{SPEC}$$

$$\frac{\vdash^{\mathsf{OL}} \sigma_3 \leq \sigma_1 \quad \vdash^{\mathsf{OL}} \sigma_2 \leq \sigma_4}{\vdash^{\mathsf{OL}} \sigma_1 \to \sigma_2 \leq \sigma_3 \to \sigma_4}\ \text{FUN} \qquad \frac{}{\vdash^{\mathsf{OL}} \tau \leq \tau}\ \text{MONO}$$

The definition of $\vdash^{\mathsf{OL}}$ follows generally that of $\vdash^{\mathsf{DM}}$, with one crucial generalization: The extra rule FUN allows it to dissect function types in the usual co- and contra-variant manner. Adding this rule allows us to instantiate deeply nested quantifiers, rather than only outermost quantifiers. For our examples, we additionally assume that various type constructors (such as `List` or `Pair`) exist in the syntax of the language, and $\vdash^{\mathsf{OL}}$ is *invariant* under such constructors.[1] Using $\vdash^{\mathsf{OL}}$, we can deduce that:

$$\begin{aligned} \vdash^{\mathsf{OL}} \texttt{Bool} \to (\forall a.a \to a) &\leq \texttt{Bool} \to \texttt{Int} \to \texttt{Int} \\ \vdash^{\mathsf{OL}} (\texttt{Int} \to \texttt{Int}) \to \texttt{Bool} &\leq (\forall a.a \to a) \to \texttt{Bool} \\ \vdash^{\mathsf{OL}} (\forall b.[b] \to [b]) \to \texttt{Bool} &\leq (\forall a.a \to a) \to \texttt{Bool} \end{aligned}$$

Notice that the $\vdash^{\mathsf{OL}}$ relation relates types that may go beyond the Damas-Milner type structure. It additionally extends the Damas-Milner instance. Compared to the System F instance, because $\vdash^{\mathsf{OL}}$ is predicative, there exist types that are related in $\vdash^{\mathsf{F}}$ but not in $\vdash^{\mathsf{OL}}$. For example $\vdash^{\mathsf{F}} \forall a\,.\,a \leq \forall b\,.\,b \to b$ but $\not\vdash^{\mathsf{OL}} \forall a\,.\,a \leq \forall b\,.\,b \to b$. On the other hand, $\vdash^{\mathsf{OL}}$ relates some more types because it descends the structure of arrow types, whereas System F instance is invariant on all data constructors because it is based only on top-level instantiations and generalizations. Nevertheless, in Section 3.1.1 we show that it is convenient to yet enrich $\vdash^{\mathsf{OL}}$ (and consequently the original Odersky-Läufer system).

2.4.1 Mitchell's type containment relation

The Odersky-Läufer type instance relation is merely a restriction of the more general relation, proposed by Mitchell. Mitchell has proposed this particular subsumption relation to form the basis of polymorphic type inference, called *type containment* [37]. The reason for this name is because it captures set containment of the denotations of types in extensional models of the polymorphic λ-calculus. In operational terms, the relation $\vdash^{\eta} \sigma_1 \leq \sigma_2$ has been shown to hold iff there exists a function of the polymorphic λ-calculus, accepting σ_1 and returning σ_2, whose type erasure is $\beta\eta$-equivalent to the identity. The resulting polymorphic λ-calculus that uses $\vdash^{\eta}$ as its instance relation was named System F_{η}, since it represents the closure of System F under η-conversions, a superset of System F (which only preserves typeability of programs under η-expansions). Mitchell's type containment has been shown undecidable [55, 60], but we show that the restriction of type

[1] We will explain why it is convenient to do so in Chapter 3.

containment with predicative instantiations is decidable, and plays its own role in annotation-driven type inference for arbitrary-rank types (Chapter 3) .

Definition 2.4.2 (Predicative restriction of Mitchell's type containment)**.** We let $\vdash^{\eta} \sigma_1 \leq \sigma_2$ be the smallest relation closed under the following inference rules (we slightly deviate from our syntax and do not treat quantified variables as vectors below, in order to express the following definition as close as possible to the original presentation):

$$\frac{\overline{b} \# ftv(\forall a\,.\,\sigma)}{\vdash^{\eta} \forall a\,.\,\sigma \leq \forall \overline{b}\,.\,[a \mapsto \tau]\sigma}\ \text{SUB} \qquad \frac{\vdash^{\eta} \sigma_1 \leq \sigma_2 \quad \vdash^{\eta} \sigma_2 \leq \sigma_3}{\vdash^{\eta} \sigma_1 \leq \sigma_3}\ \text{TRANS}$$

$$\frac{\vdash^{\eta} \sigma_3 \leq \sigma_1 \quad \vdash^{\eta} \sigma_2 \leq \sigma_4}{\vdash^{\eta} \sigma_1 \to \sigma_2 \leq \sigma_3 \to \sigma_4}\ \text{FUN}$$

$$\frac{\vdash^{\eta} \sigma_1 \leq \sigma_2}{\vdash^{\eta} \forall a\,.\,\sigma_1 \leq \forall a\,.\,\sigma_2}\ \text{ALL} \qquad \frac{}{\vdash^{\eta} \forall a\,.\,\sigma_1 \to \sigma_2 \leq (\forall a\,.\,\sigma_1) \to \forall a\,.\,\sigma_2}\ \text{DISTRIB}$$

Observe that this specification of $\vdash^{\eta}$ is rather different than the syntax-directed specification of $\vdash^{\mathsf{OL}}$, which has been used before. In particular, $\vdash^{\eta}$ includes non-syntax-directed rules, such as TRANS. We examine the precise connection of $\vdash^{\eta}$ and $\vdash^{\mathsf{OL}}$ in the next chapter.

The least familiar rule is perhaps rule DISTRIB, which distributes a $\forall$ quantifier down a function type. It allows us to view a function of type $\forall a\,.\,\sigma_1 \to \sigma_2$ as a function that may (i) accept an argument of type $\forall a\,.\,\sigma_1$, (ii) subsequently instantiate the argument with unknown a and apply the original function to it, and (iii) yield a σ_2 type—which can (iv) eventually be generalized to $\forall a\,.\,\sigma_2$. Consequently, the original function can safely be viewed as having type $(\forall a\,.\,\sigma_1) \to (\forall a\,.\,\sigma_2)$.

2.5 Propagation of polymorphic type annotations

Recall from Chapter 1 that the basic idea behind bidirectional type inference is the introduction of two mutually defined typing relations, $\Gamma \vdash_{\Uparrow} e : \tau$ for inference, and $\Gamma \vdash_{\Downarrow} e : \tau$ for checking. A contribution of this dissertation is showing the formal properties of a predicative higher-rank type

inference system with bidirectional typing rules. However, in the context of polymorphism, the idea of augmenting a type system with forms of type annotation propagation is not new.

Pierce and Turner's local type inference Pierce and Turner [45] coined the term "Local Type Inference" to refer to a partial inference technique for a language with bounded impredicative quantification and higher-rank types. Pierce and Turner base their type system on two ideas, local type argument synthesis and bidirectional propagation, and attribute the original bidirectionality idea to John Reynolds. Local type argument synthesis infers the type argument to a polymorphic function by examining the types of its arguments. Bidirectional type checking operates in one of two modes: inference and checking. A subsequent development, Colored Local Type Inference (CLTI), by Odersky and Zenger [40], reformulated bidirectional checking for $F_{\leq}$ so that the *type* and not the *judgement form* describes the direction in which type information flows. In CLTI, the colors trace the flow of information, either from the leaves to the root of the derivation or vice versa. Local type argument synthesis is not completely satisfactory for ML and Haskell programmers, who are used to writing very few or no type annotations, due to the global unification-based type inference that these languages offer. Hosoya and Pierce [13] note a few such situations.

Finally, although there is folklore work about combining bidirectional propagation with Damas-Milner type inference, there has been relatively little published work that describes such systems [1, 28]. Our work on bidirectional type inference for higher-rank types, as well as our subsequent work with Boxy Types [58] fills this gap. Boxy Types uses the CLTI idea of combining the two forms of typing judgments and uses the structure of the types to specify inference or checking mode. We defer any further discussion on Boxy Types for Chapter 8, where we give a thorough description and comparison to other systems for first-class polymorphism.

Two-phase type inference based on type containment In parallel with the development of Boxy Types, Rémy designed a two-phase approach to type inference for higher-rank and impredicative polymorphism called $\mathsf{F}^{?}_{\mathsf{ML}}$ [46]. The first phase infers the "shape" of the type of each variable in the program, where "shape" means the exact location of all quantifiers and the type variables they bind. Once shapes are known, only monotypes need be "guessed" by the second phase, which

is done using ordinary unification. This division separates the mechanisms for propagating local type information (which must be done in a syntax-directed way) from the underlying first-order type inference. As a result, the two separate components of the type system may be thought about independently—but of course, both must be understood together to understand whether a program should type check. There is an argument on the other hand in favor of merging the two phases together, an approach that has been demonstrated in the Boxy Types paper: the separation between shape propagation and type inference in $\mathsf{F}^{?}_{\mathsf{ML}}$ means that shape propagation can not take advantage of polymorphic types inferred by type inference (through generalization). Boxy Types adds the inferred polymorphic type of the `let`-bound expression into the context as known type information before checking the body of the definition. As a result, Rémy proposes incremental elaboration: the type of each top level definition is completely determined before continuing to the next one. However, this strategy treats top-level definitions differently than internal `let`-bound definitions. Finally, $\mathsf{F}^{?}_{\mathsf{ML}}$ supports the full predicative version of Mitchell's type containment. On the other hand FPH and Boxy Types require instance relations that are more restrictive in one axis (Boxy Types requires function argument invariance, and FPH relies on System F instance), but more expressive in a different axis (both Boxy Types and FPH allow impredicative instantiations).

Chapter 3

Type inference for predicative higher-rank types

The work presented in this chapter connects a folklore bidirectional type system (implemented originally in GHC) to the Odersky-Läufer higher-rank type system. The type structure of that folklore system supported only the higher-rank types in *prenex form*, where all the $\forall$ quantifiers to the right of arrows were hoisted to the top level. For example, a type annotation as $\texttt{Int} \to \forall a\,.\,a \to a$ would be hoisted internally to $\forall a\,.\texttt{Int} \to a \to a$. With the work presented here we were able to extend the syntax of types that type inference accepts to arbitrary System F types, by devising a subsumption relation that captures the essence of Mitchell's type containment, restricted to its predicative part. Despite the fact that the impredicative type containment relation is undecidable [55, 60], we observed that the predicative part is decidable, indeed using a rather straightforward algorithm. Incidentally, the subsumption relation employed in this work ensures extension of the Odersky-Läufer type system, internalizes the hoisting of quantifiers—contrary to the prior ad-hoc pre-processing of type annotations—and improves robustness of type inference under program η-conversions.

The modified bidirectional system can be implemented by an algorithm [56] that supports a mixture of type inference and type checking, and is sound and complete with respect to the proposed bidirectional type system. As a result, a principal types property is established: For inference mode, there

exist principal types that can be assigned to typeable programs. The crucial bit here is that typeable programs may include type annotations, and hence our result does not contradict the aforementioned lack of principal types. In addition, despite the fact that the subsumption relation employed is no longer the Damas-Milner subsumption that arises from type instantiations and generalizations, we showed that the most general type for an expression (and the one that the type inference algorithm returns) differs from all others that can be inferred for the same expression only in the Damas-Milner instance relation, confirming our intuition that the type system does not "guess" polymorphism: the polymorphic type structure under the top-level quantifiers remains the same for all inferred types for the same program, because it arises solely from type annotations.

We now turn to the detailed presentation of the work on type inference for predicative higher-rank types. We give a syntax-directed variation on the original Odersky-Läufer type system and propose a bidirectional version that supports type annotation propagation. We give properties of various polymorphic subsumption relations, connect the various type systems, and sketch a sound and complete type inference algorithm.

The work on this chapter supports predicative higher-rank polymorphism. Because FPH supports both impredicative and higher-rank types, without losing many of the typeable programs in the higher-rank predicative work, we believe that FPH supersedes this work. Hence we omit the details of the algorithmic implementation and the proofs of the theorems in this chapter. The extra material can be found in the accompanying paper [41] and technical report [56].

3.1 A syntax-directed higher-rank system

The typing rules of Figure 2.4 are not syntax-directed, meaning that for every syntactic form of a term there are possible more than two typing rules that are applicable in constructing a typing derivation for the term. In particular, rule GEN allows generalization at arbitrary points, and rule SUBS allows us to move along the $\vdash^{\mathsf{OL}}$ instance relation anywhere on the syntax tree. Hence, the typing rules of Figure 2.4 do not directly suggest an implementation.

The idea originating in the Damas-Milner type system, is to instantiate at variable occurrences, and

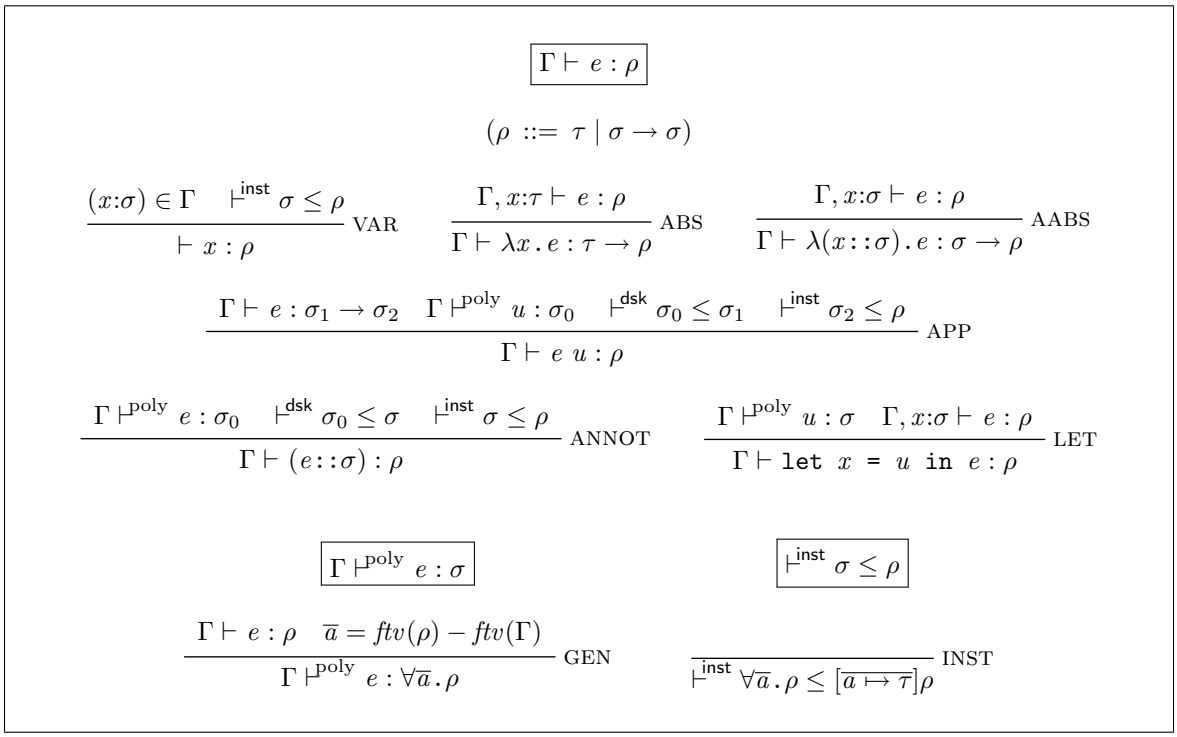

Figure 3.1: Syntax-directed higher-rank type system

generalize at `let` nodes. Applying the same idea in the Odersky-Läufer type system (and slightly deviating from the original Odersky-Läufer syntax-directed system) leads us to the system presented in Figure 3.1. Notice that the instantiation judgement, $\vdash^{\mathsf{inst}} \sigma \leq \rho$, instantiates only the *outermost* quantified type variables of σ; and similarly the generalization judgement, $\Gamma \vdash^{\mathrm{poly}} e : \sigma$, generalizes only the *outermost* type variables of σ. Any polytypes hidden under $\to$ constructors are unaffected. Additionally, in rule ANNOT of Figure 3.1 we must invoke subsumption, but we use yet another form of subsumption, $\vdash^{\mathsf{dsk}}$, for reasons we discuss next. The other interesting feature of the rules is that in rule APP we must use $\vdash^{\mathrm{poly}}$ to infer a polytype σ_0 for the argument, because the function may require the argument to have a polytype σ_1. These two types may not be identical, because the argument may be more polymorphic than required, so $\vdash^{\mathsf{dsk}}$ is used to compare the two.

3.1.1 Too weak a subsumption for Odersky-Läufer

In the syntax-directed typing relation of Figure 3.1 we have used a new form of subsumption (not yet defined), which we write $\vdash^{\mathsf{dsk}}$. If we instead used the Odersky-Läufer subsumption, $\vdash^{\mathsf{OL}}$, the type system would be perfectly sound, but it would type fewer programs than the non-syntax-directed system of Figure 2.4. To see why, consider this program:

```
let f = \x y -> y
in (f :: forall a. a -> (forall b. b -> b))
```

A Haskell programmer would expect to infer the type $\forall ab\,.\,a \to b \to b$ for the `let`-binding of `f`, and that is what the rules of Figure 3.1 would do. The type-annotated occurrence of `f` then requires that `f`'s type be more polymorphic than the supplied signature, but alas, under the subsumption rules of Figure 2.4, it is *not* the case that $\vdash^{\mathsf{OL}} \forall ab\,.\,a \to b \to b \leq \forall a\,.\,a \to (\forall b\,.\,b \to b)$, although the converse is true. On the other hand, the typing rules of Figure 2.4 *could* give the `let`-binding the type $\forall a.a \to (\forall b.b \to b)$ and then $\vdash^{\mathsf{OL}}$ would succeed. In short, the syntax-directed rules do not find the most general type for `f`, according to $\vdash^{\mathsf{OL}}$.

One obvious solution is to fix Figure 3.1 to infer the type $\forall a\,.\,a \to (\forall b\,.\,b \to b)$ for the `let`-binding for `f`. Odersky and Läufer's syntax-directed version of their language does this by generalizing every abstraction body in rules ABS and AABS, so that the $\forall$'s in the result type occur as far to the right as possible. Here is the modified rule ABS (cast in our syntax and typing rules):

$$\frac{\Gamma, x{:}\tau \vdash^{\text{poly}} e : \sigma}{\Gamma \vdash \lambda x\,.\,e : \tau \to \sigma}\ \text{EAGER-ABS}$$

We call this approach *eager generalization*; but is not desirable for practical reasons. A superficial but practically-important difficulty is that it yields inferred types that programmers will find unfamiliar. Furthermore, if the programmer adds a type signature, such as `f::`$\forall ab.a \to b \to b$, he may make the function type less general without realizing it. Finally, there is a problem related to the possible addition of conditionals. For example, consider the term:

```
if ... then (\x y -> y) else (\x y -> x)
```

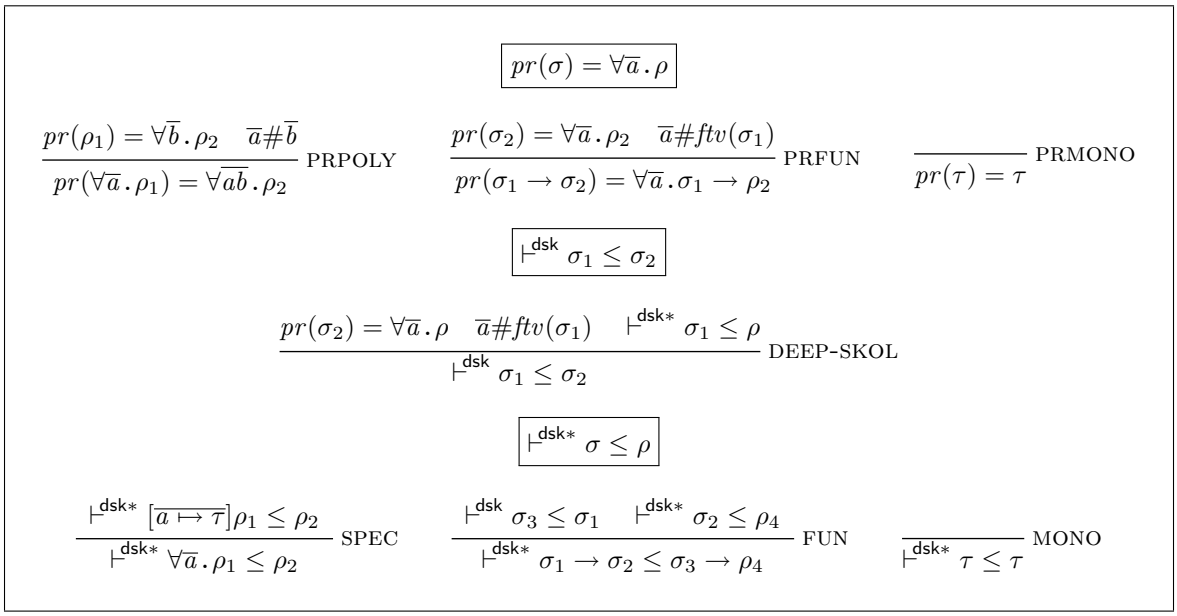

Figure 3.2: Subsumption with deep skolemization

This term will type fine in Haskell, but eager generalization would yield $\forall a \,.\, a \to (\forall b \,.\, b \to b)$ for the `then` branch, and $\forall a \,.\, a \to (\forall b \,.\, b \to a)$ for the `else` branch—and now to unify the two types one would need to compute their "join" under the subsumption relation used. Such joins exist for the predicative instance relations that we consider, but the implementation of the type checker becomes more complicated. In contrast, if one does not generalize the bodies of abstractions then, before generalizing the two types for the branches of a conditional, one can perform ordinary unification, a simpler alternative. Similar concerns arise with the possible branches of pattern matching constructs.

3.1.2 The solution: deep skolemization

Fortunately, a solution that does not involve eager generalization of abstraction bodies is available. The difficulty arises because it is not the case that $\vdash^{\mathsf{OL}} \forall ab \,.\, a \to b \to b \leq \forall a \,.\, a \to (\forall b \,.\, b \to b)$. But the two types are isomorphic in a "semantic sense": More concretely, if $f : \forall ab \,.\, a \to b \to b$, then we can construct, using f, a System F term of type $\forall a \,.\, a \to (\forall b \,.\, b \to b)$, namely: $(\Lambda a \,.\, \lambda (x{:}a) \,.\, \Lambda b \,.\, \lambda (y{:}b) \,.\, f\ [a]\ [b]\ x\ y)$. Likewise, if $g : \forall a \,.\, a \to (\forall b \,.\, b \to b)$ then we can also construct: $(\Lambda a \,.\, \Lambda b \,.\, \lambda (x{:}a) \,.\, \lambda (y{:}b) \,.\, g\ [a]\ x\ [b]\ y)$, of type $\forall ab \,.\, a \to b \to b$. So, from a semantic point of view, the two types should be equivalent. We would like the two types to be equivalent as well in the syntactic

subsumption relation. The solution to the problem is to *enrich* the definition of subsumption, so that the type systems of Figure 2.4 and 3.1 admit the same programs. That is the reason for the new subsumption judgement $\vdash^{\mathsf{dsk}}$, defined in Figure 3.2. This relation subsumes $\vdash^{\mathsf{OL}}$; it relates strictly more types.

The key idea is that in DEEP-SKOL (Figure 3.2), we begin by pre-processing σ_2 to float out all its $\forall$s that appear to the right of a top level arrow, so that they can be skolemized immediately. We call this rule "DEEP-SKOL" because it skolemizes quantified variables even if they are nested inside the result type of σ_2. The floating process is done by an auxiliary function $pr(\sigma)$, called *weak prenex conversion*, also defined in Figure 3.2. For example, $pr(\forall a\,.\,a \to (\forall b\,.\,b \to b)) = \forall ab\,.\,a \to b \to b$. In general, $pr(\sigma)$ takes an arbitrary polytype σ and returns a polytype of the form $pr(\sigma) = \forall \overline{a}\,.\sigma_1 \to \ldots \to \sigma_n \to \tau$. There can be $\forall$ quantifiers inside some σ_i, but there are no $\forall$ quantifiers in the result types of the top-level arrows. Of course, when floating out the $\forall$s, we must be careful to avoid accidental capture, which is the reason for the side condition in the second rule for $pr()$. (We can always α-convert the type to satisfy this condition.) We call it "weak" prenex conversion because it leaves the argument types σ_i unaffected.

To keep the system syntax-directed, we have split the subsumption judgement into two. The main one, $\vdash^{\mathsf{dsk}}$, has a single rule that performs deep skolemization and invokes the auxiliary judgement, $\vdash^{\mathsf{dsk}*}$. The latter has the remaining rules for subsumption, unchanged from $\vdash^{\mathsf{OL}}$, except that FUN invokes $\vdash^{\mathsf{dsk}}$ on the argument types but $\vdash^{\mathsf{dsk}*}$ on the result types. To see why the split is necessary, consider trying to check $\vdash^{\mathsf{dsk}} \forall a.\mathtt{Int} \to a \to a \leq \mathtt{Int} \to \forall a.a \to a$. Even though the $\forall$ quantifier on the right is hidden under the arrow, we must still use DEEP-SKOL before SPEC.

The function $pr(\sigma)$ converts σ to weak prenex form "on the fly". Another workable alternative, indeed the one that was used in the GHC implementation prior to the development of this work, is to ensure that *all* types are syntactically constrained to be in prenex form, using a restricted syntax for ρ-types, by taking $\rho ::= \tau \mid \sigma \to \rho$. In theory this seems less elegant, and is a little less convenient in practice because programmers often want to write types in non prenex-form. For instance, this need has occured in the context of generic programming in Haskell [23]. The syntactically-constrained system also seems more fragile if we wanted to move to an impredicative

system, because instantiation could yield a syntactically-illegal type.

Although $\vdash^{\mathsf{dsk}}$ subsumptions are justified since there exist System F functions that are $\beta\eta$-convertible to the identity that witness them, once effects or non-termination are added to the language this semantic argument is in question. In the case of Haskell, lazyness eliminates the problem of differentiations in observable behavior due to non-termination. But the deep-skolemization approach would not work for ML, because in ML the types $\forall ab\,.\,a \to b \to b$ and $\forall a\,.\,a \to (\forall b\,.\,b \to b)$ are *not* isomorphic: one cannot push $\forall$ quantifiers around freely, as this transformation affects which (potentially effectful) expressions are values and which not. Alternative approaches are discussed in work by Rémy [46].

3.2 Bidirectional type inference

The rules of Figure 3.1 are syntax-directed, but they share with the original Odersky-Läufer system the property that the type of the argument of the λ-abstraction can be polymorphic only if the λ-bound variable is explicitly annotated; compare rules ABS and AABS in Figure 3.1. As outlined in the introduction, this seems far too heavyweight. Somehow we would like to "push type annotations inwards", so that type signatures can be exploited to eliminate other function argument annotations. The idea is to apply bidirectionality from local type inference [45], combining it however with ordinary unification-based type inference (for global type argument synthesis). As an added benefit, compiler type error messages can become more informative in the presence of local type inference, as more "known" type information is propagated closer to the expressions that must be typed using it.

3.2.1 Bidirectional inference judgements

Figure 3.3 gives rules that express the idea of propagating types inwards. The figure describes two mutually defined judgements. The judgement $\Gamma \vdash_{\Uparrow} e : \rho$ can be read as "in the environment Γ, type ρ can be *inferred* for the term e", whereas the judgement $\Gamma \vdash_{\Downarrow} e : \rho$ can be read as "in environment

Γ, the term e can be *checked* against type ρ". The judgements $\vdash^{\text{poly}}$ and $\vdash^{\mathsf{inst}}$ are generalized in the same way.

$$\boxed{\Gamma \vdash_\delta e : \rho}$$

$$(\rho ::= \tau \mid \sigma \to \sigma \qquad \text{and} \qquad \delta ::= \Uparrow \mid \Downarrow)$$

$$\frac{(x{:}\sigma) \in \Gamma \quad \vdash^{\mathsf{inst}}_\delta \sigma \leq \rho}{\Gamma \vdash_\delta x : \rho}\ \text{VAR} \qquad \frac{\Gamma, (x{:}\tau) \vdash_\Uparrow e : \rho}{\Gamma \vdash_\Uparrow \lambda x\,.\,e : \tau \to \rho}\ \text{ABS1} \qquad \frac{\Gamma, (x{:}\sigma_a) \vdash^{\text{poly}}_\Downarrow e : \sigma_r}{\Gamma \vdash_\Downarrow \lambda x\,.\,e : \sigma_a \to \sigma_r}\ \text{ABS2}$$

$$\frac{\Gamma, (x{:}\sigma) \vdash_\Uparrow e : \rho}{\Gamma \vdash_\Uparrow \lambda(x::\sigma)\,.\,e : \sigma \to \rho}\ \text{AABS1} \qquad \frac{\vdash^{\mathsf{dsk}} \sigma_a \leq \sigma_x \quad \Gamma, (x{:}\sigma_x) \vdash^{\text{poly}}_\Downarrow e : \sigma_r}{\Gamma \vdash_\Downarrow \lambda(x::\sigma_x)\,.\,e : \sigma_a \to \sigma_r}\ \text{AABS2}$$

$$\frac{\Gamma \vdash_\Uparrow e : \sigma_1 \to \sigma_2 \quad \Gamma \vdash^{\text{poly}}_\Downarrow u : \sigma_1 \quad \vdash^{\mathsf{inst}}_\delta \sigma_2 \leq \rho}{\Gamma \vdash_\delta e\ u : \rho}\ \text{APP} \qquad \frac{\Gamma \vdash^{\text{poly}}_\Downarrow e : \sigma \quad \vdash^{\mathsf{inst}}_\delta \sigma \leq \rho}{\Gamma \vdash_\delta (e::\sigma) : \rho}\ \text{ANNOT} \qquad \frac{\Gamma \vdash^{\text{poly}}_\Uparrow u : \sigma \quad \Gamma, x{:}\sigma \vdash_\delta e : \rho}{\Gamma \vdash_\delta \texttt{let}\ x = u\ \texttt{in}\ e : \rho}\ \text{LET}$$

$$\boxed{\Gamma \vdash^{\text{poly}}_\delta e : \sigma}$$

$$\frac{\overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma) \quad \Gamma \vdash_\Uparrow e : \rho}{\Gamma \vdash^{\text{poly}}_\Uparrow e : \forall \overline{a}\,.\,\rho}\ \text{GEN1} \qquad \frac{\overline{a} \# \mathit{ftv}(\Gamma) \quad \Gamma \vdash_\Downarrow e : \rho \quad pr(\sigma) = \forall \overline{a}\,.\,\rho}{\Gamma \vdash^{\text{poly}}_\Downarrow e : \sigma}\ \text{GEN2}$$

$$\boxed{\vdash^{\mathsf{inst}}_\delta \sigma \leq \rho}$$

$$\frac{}{\vdash^{\mathsf{inst}}_\Uparrow \forall \overline{a}\,.\,\rho \leq [\overline{a \mapsto \tau}]\rho}\ \text{INST1} \qquad \frac{\vdash^{\mathsf{dsk}} \sigma \leq \rho}{\vdash^{\mathsf{inst}}_\Downarrow \sigma \leq \rho}\ \text{INST2}$$

Figure 3.3: Bidirectional version of Odersky-Läufer

A central characteristic of the bidirectional typing rules is that a term might be typeable in checking mode when it is not typeable in inference mode; for example the term $\lambda g\,.\,(g\ 3, g\ \texttt{True})$ can be checked against type $(\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$, but is not typeable in inference mode. However, if we infer the type for a term, we can always check that the term has that type: if $\Gamma \vdash_\Uparrow e : \rho$ then $\Gamma \vdash_\Downarrow e : \rho$. Furthermore, checking mode allows us to impress on a term any type that is more specific than its most general (inferable) type. In contrast, inference mode may only produce a type that is some substitution of the most general type. For example, if a variable has type $\forall a\,.\,a \to (\forall b\,.\,b \to b)$,

we can check that it has the same type and also that it has types $\texttt{Int} \to (\forall b\,.\,b \to b)$ and $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$. On the other hand, of these types, we will only be able to infer $c \to (\forall b\,.\,b \to b)$ and $\texttt{Int} \to (\forall b\,.\,b \to b)$; and in general any $\tau \to (\forall b\,.\,b \to b)$.

Finally, any term typeable by the uni-directional rules of Figure 3.1 is also typeable in inference mode by Figure 3.3. That is, if $\Gamma \vdash e : \rho$ using the rules of Figure 3.1 then $\Gamma \vdash_{\Uparrow} e : \rho$. The reverse is of course false.

3.2.2 Bidirectional inference rules

Many of the rules in Figure 3.3 are "polymorphic" in the direction δ. For example, the rules VAR, APP, ANNOT, and LET are insensitive to δ, and can be seen as shorthand for two rules that differ only in the arrow direction.

The rule APP, which deals with function application $(e\ u)$, is of particular interest. Regardless of the direction δ, we first *infer* the type $\sigma_1 \to \sigma_2$ for e, and then *check* that u has type σ_1. In this way we take advantage of the function's type (which is often directly extracted from Γ), to provide the type context for the argument. We then use $\vdash^{\mathsf{inst}}$ to check that σ_2 and ρ are compatible. Notice that, even in the checking case $\Downarrow$, we ignore the required type ρ when inferring the type for the function e. There is clearly some information loss here: we know the result type, ρ, for e, but we do not know its argument type. The rules provide no way to express this *partial* information about e's type; follow-up work on Boxy Types [58] has addressed this problem.

Dually, ordinary (un-annotated) λ-abstractions are dealt with by rules ABS1 and ABS2. The inference case (ABS1) is just as in the syntax-directed Odersky-Läufer type system, but the checking case (ABS2) is more interesting. To check that $\lambda x\,.\,e$ has type $\sigma_a \to \sigma_r$, we bind x to the polytype σ_a, *even though x is not explicitly annotated*, before checking that the body has type σ_r. In this way, we take advantage of contextual information, in way that reduces the necessity for type annotations.

We also need two rules for annotated λ-abstractions. In the inference case, AABS1, we extend the environment with the σ-type specified by the annotation, and infer the type of the body. In the checking case, AABS2, we extend the environment in the same way, before checking that the body has

the specified type—but we must also check that the argument type expected by the environment σ_a is more polymorphic than that specified in the type annotation σ_x. Notice the argument type contravariance that makes expressions such as $(\lambda(f :: \texttt{Int} \to \texttt{Int}).f\ 3) :: (\forall a.\, a \to a) \to \texttt{Int}$ typeable.

3.2.3 Instantiation and generalization

The $\vdash^{\mathsf{inst}}_{\delta}$ judgement also has separate rules for inference and checking. Rule INST1 deals with the inference case: just as in the old INST rule of Figure 3.1, we simply instantiate the outer $\forall$ quantifiers. The checking case, INST2, is more interesting. Here we are checking that an inferred type σ is more general than the expected type ρ, and we can simply use the subsumption judgement $\vdash^{\mathsf{dsk}}$. Rules ANNOT and VAR both make use of the $\vdash^{\mathsf{inst}}_{\delta}$, just as they did in Figure 3.1, but ANNOT becomes slightly simpler. In the syntax-directed rules of Figure 3.1, we inferred a type for e, and performed a subsumption check against the specified type; now we can simply push the specified type inwards, into e.

The generalization judgement $\Gamma \vdash^{\mathrm{poly}}_{\delta} e : \sigma$ also has two cases. When we are inferring a polytype (rule GEN1) we need to quantify over all free variables of the inferred ρ type that do not appear in the environment Γ, just as before. On the other hand, when we check that a polytype can be assigned to a term (rule GEN2), we simply skolemize the quantified variables, checking that they do not appear free in the environment Γ. The situation is very similar to that of DEEP-SKOL in Figure 3.2, so GEN2 must perform weak prenex conversion on the expected type σ, to bring all its quantifiers to the top. If it fails to do so, the following would not be derivable:

$$(f{:}\forall ab.\, \texttt{Int} \to a \to b \to b) \vdash^{\mathrm{poly}}_{\Downarrow} f\ 3 : \texttt{Bool} \to \forall c.\, c \to c$$

The problem is that the type of f is instantiated by VAR before rule APP invokes $\vdash^{\mathsf{inst}}_{\Downarrow}$ to match the result type with the type of $f\ 3$, and hence before the $\forall c.\, c \to c$ is skolemized. Once we use GEN2, however, the reader may verify that $\Gamma \vdash_{\Downarrow} e : \rho$ is invoked only when ρ is in weak prenex form. However, for generality we prefer to define $\vdash_{\Downarrow}$ over arbitrary ρ-types. For example, this generality allows us to state, without side conditions, that if $\Gamma \vdash_{\Uparrow} e : \rho$ then $\Gamma \vdash_{\Downarrow} e : \rho$.

3.3 Elaboration semantics

We turn to the question of type soundness for the type system in Figure 3.3. Since type inference can be viewed as reconstructing the appropriate type information from our *implicitly-typed source language* into an *explicitly-typed target language*, we can give a definition of the operational semantics of the source language via the semantics of the target language and the translation. In our case, the target of type inference is System F. Notably, type-checking an explicitly typed System F program at a later stage (which is very easy to do) gives a very strong consistency check that the intermediate stages have not performed an invalid transformation [38, 53, 52, 20]. In the rest of this section we sketch how to specify the translation into System F; for full details we refer the reader to the accompanying paper and report. We assume familiarity with the explicitly typed System F (see Pierce's book [44] for a modern presentation).

3.3.1 The translation

The translation to target programs is *type-directed*, as it is specified using our typing relations. For example, the main judgement for the bidirectional system becomes $\Gamma \vdash_\delta e : \rho \rightsquigarrow t$ meaning that e has type ρ, and translates to the *explicitly typed* System F term t. Furthermore the term t will have type ρ in System F; we write $\Gamma \vdash^{\mathsf{F}} t : \rho$. As far as notation is concerned we use meta-variables t, f for System F terms in what follows. The translated System F terms have explicit type annotations on binders. For example, rule ABS1 from Figure 3.3 becomes

$$\frac{\Gamma, (x : \tau) \vdash_{\Uparrow} e : \rho \rightsquigarrow t}{\Gamma \vdash_{\Uparrow} \lambda x \,.\, e : \tau \to \rho \rightsquigarrow \lambda x{:}\tau \,.\, t} \text{ ABS1}$$

Many of the other rules in Figure 3.3 can be modified in a similar routine way. In effect, the translated program encodes the exact shape of the derivation tree, and therefore amounts to a proof that the original program is indeed well typed. However matters become interesting when we consider instantiation and generalization. Consider rule VAR from Figure 3.3 below, at the left:

$$\frac{(x{:}\sigma) \in \Gamma \quad \vdash^{\mathsf{inst}}_\delta \sigma \leq \rho}{\Gamma \vdash_\delta x : \rho} \qquad \frac{(x{:}\sigma) \in \Gamma \quad \vdash^{\mathsf{inst}}_\delta \sigma \leq \rho \rightsquigarrow f}{\Gamma \vdash_\delta x : \rho \rightsquigarrow f\ x}$$

To see what term we must create, we first observe that x cannot translate to simply x, because x has type σ, not ρ. Hence, the $\vdash^{\mathsf{inst}}$ judgement should return a *coercion function* f of type $\sigma \to \rho$, which can be thought of as concrete—indeed, executable—evidence for the claim that $\sigma \leq \rho$. Then we can add translations to the VAR rule, resulting to the rule at the right. The rules for the $\vdash^{\mathsf{inst}}$ judgement are straightforward:

$$\frac{}{\vdash^{\mathsf{inst}}_{\Uparrow} \forall \overline{a}\,.\,\rho \;\leq\; [\overline{a} \mapsto \overline{\tau}]\rho \rightsquigarrow \lambda(x\!:\!\forall \overline{a}\,.\,\rho).x\ [\overline{\tau}]}\ \textsc{inst}1 \qquad \frac{\vdash^{\mathsf{dsk}} \sigma \;\leq\; \rho \rightsquigarrow t}{\vdash^{\mathsf{inst}}_{\Downarrow} \sigma \;\leq\; \rho \rightsquigarrow t}\ \textsc{inst}2$$

The inference case, rule INST1, uses a System F type application ($x\ [\overline{\tau}]$) to record the types at which x is instantiated (we use the notation $[\overline{\tau}]$ to mean a sequence of type applications). For the checking case, rule INST2 defers to $\vdash^{\mathsf{dsk}}$, which also returns a coercion function. Dually, generalization is expressed by System F type abstraction, as we can see in the rules for $\vdash^{\mathrm{poly}}$:

$$\frac{\overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma) \quad \Gamma \vdash_{\Uparrow} e : \rho \rightsquigarrow t}{\Gamma \vdash^{\mathrm{poly}}_{\Uparrow} e : \forall \overline{a}\,.\,\rho \rightsquigarrow \Lambda \overline{a}\,.\,t}\ \textsc{gen}1 \qquad \frac{\overline{a} \# \mathit{ftv}(\Gamma) \quad pr(\sigma) = \forall \overline{a}\,.\,\rho \rightsquigarrow f \quad \Gamma \vdash_{\Downarrow} e : \rho \rightsquigarrow t}{\Gamma \vdash^{\mathrm{poly}}_{\Downarrow} e : \sigma \rightsquigarrow f\ (\Lambda \overline{a}\,.\,t)}\ \textsc{gen}2$$

Rule GEN1 directly introduces a type abstraction, but GEN2 needs a coercion function, just like VAR, to account for the prenex-form conversion.

For completeness, we include the rules for weak prenex conversion and for $\vdash^{\mathsf{dsk}}$ in Figure 3.4. The key invariants that one has to keep in mind when reading the rules is that if $\vdash^{\mathsf{dsk}} \sigma_1 \leq \sigma_2 \rightsquigarrow t$ then $\vdash^{\mathsf{F}} t : \sigma_1 \to \sigma_2$, and if $pr(\sigma_1) = \sigma_2 \rightsquigarrow t$ then $\vdash^{\mathsf{F}} t : \sigma_2 \to \sigma_1$.

As a final remark, although the semantics of the language is defined by translation, it is fairly simple and expected. If we erase types in the source and target languages it is easy to verify that, except for the insertion of coercions, the translation is the identity translation. Furthermore, the coercions themselves only produce terms that, after type erasure, are $\beta\eta$-equivalent to the identity.

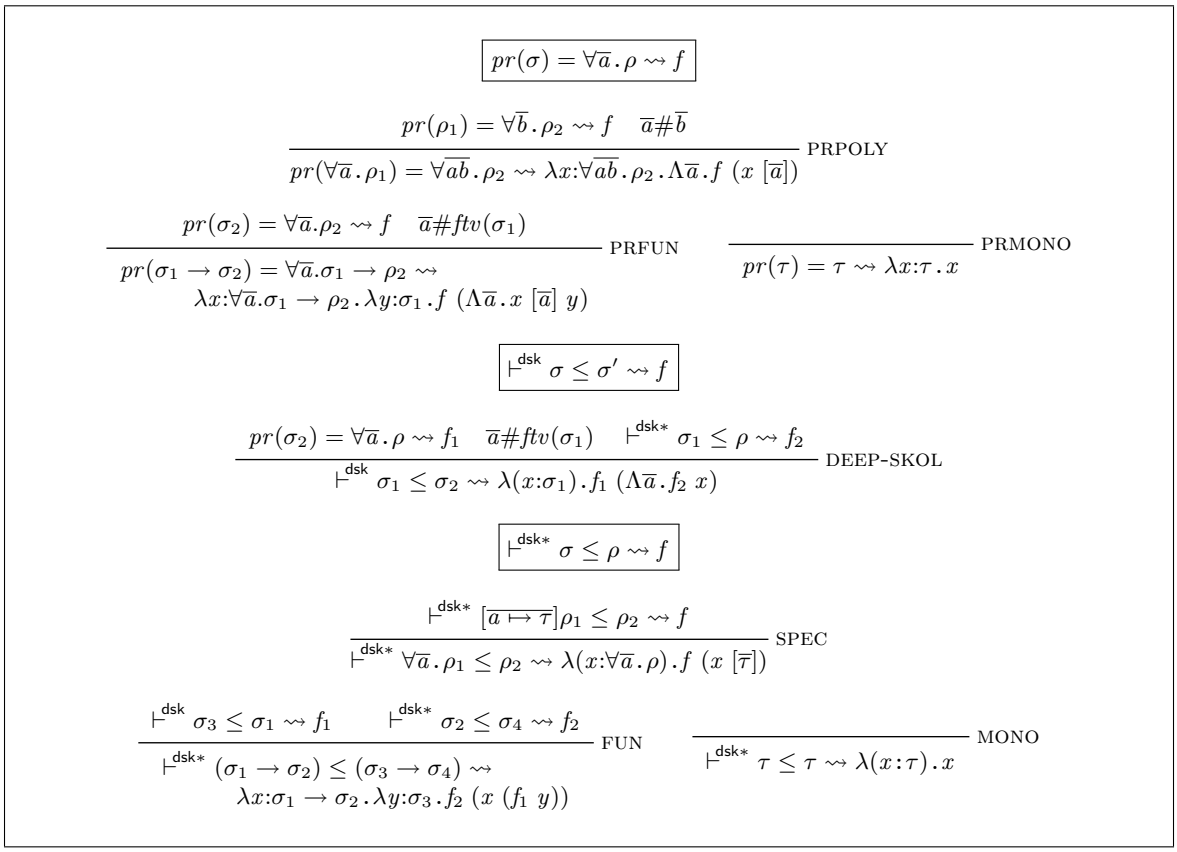

Figure 3.4: Creating coercion terms

3.3.2 Subsumption, elaboration, and datatypes

The fact that the semantics of the bidirectional system is defined via an elaboration to System F where usages of $\vdash^{\mathsf{dsk}}$ introduce coercion functions has consequences for the subsumption relation on datatypes other than $\to$.

A first, more superficial, complication has to do with detecting the variance of a datatype. In some cases this is easy (such as for pairs or lists), but in general an analysis of the datatype constructors is required.

A second complication has to do with the elaboration semantics. Assuming that the list datatype

is co-variant, the typing rule for the subsumption relation on lists should be the following:

$$\frac{\vdash^{\mathsf{dsk}} \sigma_1 \leq \sigma_2}{\vdash^{\mathsf{dsk}} [\sigma_1] \leq [\sigma_2]}$$

Consequently, if $\vdash^{\mathsf{dsk}} \sigma_1 \leq \sigma_2$ gives rise to a coercion function $f : \sigma_1 \to \sigma_2$, we need to lift f to $[\sigma_1] \to [\sigma_2]$ to create the coercion witnessing $\vdash^{\mathsf{dsk}} [\sigma_1] \leq [\sigma_2]$. For lists (and functorial datatypes) there is an obvious map function which performs exactly this. However, mapping coercion functions down data structures does not seem entirely satisfactory as it may have operational cost.

To avoid these complications, we would require that possible extensions of the $\vdash^{\mathsf{dsk}}$ relation be invariant on all defined datatypes. This way there is no need for mapping coercions. In the rest of this document we will assume that $\vdash^{\mathsf{dsk}}$ is invariant on user-defined datatypes such as lists or pairs.

3.4 Formal properties of higher-rank type systems

In this section we give formal statements of the most important properties of the type systems and subsumption relations presented so far.

3.4.1 Properties of the subsumption judgements

Let us recall the three relations, $\vdash^{\mathsf{DM}}$ (Damas-Milner instance), $\vdash^{\mathsf{OL}}$ (Odersky-Läufer subsumption), and $\vdash^{\mathsf{dsk}}$ (deep-skolemization subsumption), all based on the type structure that is produced with $\rho ::= \tau \mid \sigma \to \sigma$. These three relations are connected in the following way: Deep skolemization subsumption relates strictly more types than the Odersky-Läufer subsumption, which in turn relates strictly more types than the Damas-Milner instance.

Theorem 3.4.1. *If* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$ *then* $\vdash^{\mathsf{OL}} \sigma_1 \leq \sigma_2$. *If* $\vdash^{\mathsf{OL}} \sigma_1 \leq \sigma_2$ *then* $\vdash^{\mathsf{dsk}} \sigma_1 \leq \sigma_2$.

The following theorem captures the essence of $\vdash^{\mathsf{dsk}}$; any type is equivalent to its prenex form. While it is only $\vdash^{\mathsf{OL}} \sigma \leq pr(\sigma)$ and not $\vdash^{\mathsf{OL}} pr(\sigma) \leq \sigma$, we have:

Theorem 3.4.2. $\vdash^{\mathsf{dsk}} \sigma \leq pr(\sigma)$ *and* $\vdash^{\mathsf{dsk}} pr(\sigma) \leq \sigma$.

All three relations are reflexive and transitive. However, only deep skolemization subsumption enjoys a distributivity property, that lets us distribute quantification among the components of an arrow type:

Theorem 3.4.3 (Distributivity)**.** $\vdash^{\mathsf{dsk}} \forall a\,.\,\sigma_1 \to \sigma_2 \leq (\forall a\,.\,\sigma_1) \to \forall a\,.\,\sigma_2$.

Distributivity is essential for showing that the coercion functions generated by our $\vdash^{\mathsf{dsk}}$ derivations correspond exactly to the System F functions that, after erasure of types, are $\beta\eta$-convertible to the identity. In fact, it turns out that our deep skolemization relation corresponds to the restriction of Mitchell's type containment relation that only includes predicative instantiations, $\vdash^{\eta}$, from Chapter 2.

Theorem 3.4.4. $\vdash^{\mathsf{dsk}} \sigma_1 \leq \sigma_2$ *iff* $\vdash^{\eta} \sigma_1 \leq \sigma_2$.

Though Mitchell's containment was originally presented in a declarative style (and we use the same presentation in Chapter 2), syntax-directed presentations of containment are also known [54, 34]. In particular, Longo *et al.* [34] employ an idea similar to our deep skolemization. To the best of our knowledge, no one had previously considered whether the predicative variant of the containment relation was decidable, although it is not a hard problem; our presentation in Figure 3.2 is a syntax-directed presentation, with a straightforward implementation based on first-order unification.

3.4.2 Connections between the type systems

We now present the formal connections between all the systems that have been discussed so far. To state our results in their full generality, we assume the existence of a syntax-directed Damas-Milner type system, which very much like the syntax-directed system of Figure 2.4 includes a generalization judgement, $\vdash_{\mathsf{sd}}^{\mathsf{DM}_{\mathsf{poly}}}$ [56]. The following table summarizes our notation:

$\vdash^{\mathsf{DM}}$	Damas-Milner typing relation
$\vdash_{\mathsf{sd}}^{\mathsf{DM}_{\mathsf{poly}}}$	Syntax-directed Damas-Milner type generalization
$\vdash_{\mathsf{nsd}}$	Odersky-Läufer typing relation (Figure 2.4) with $\vdash^{\mathsf{dsk}}$ instead of $\vdash^{\mathsf{OL}}$
$\vdash_{\mathsf{sd}}$	Syntax-directed Odersky-Läufer typing relation
$\vdash_{\Uparrow}$	Bidirectional inference relation
$\vdash_{\Downarrow}$	Bidirectional checking relation

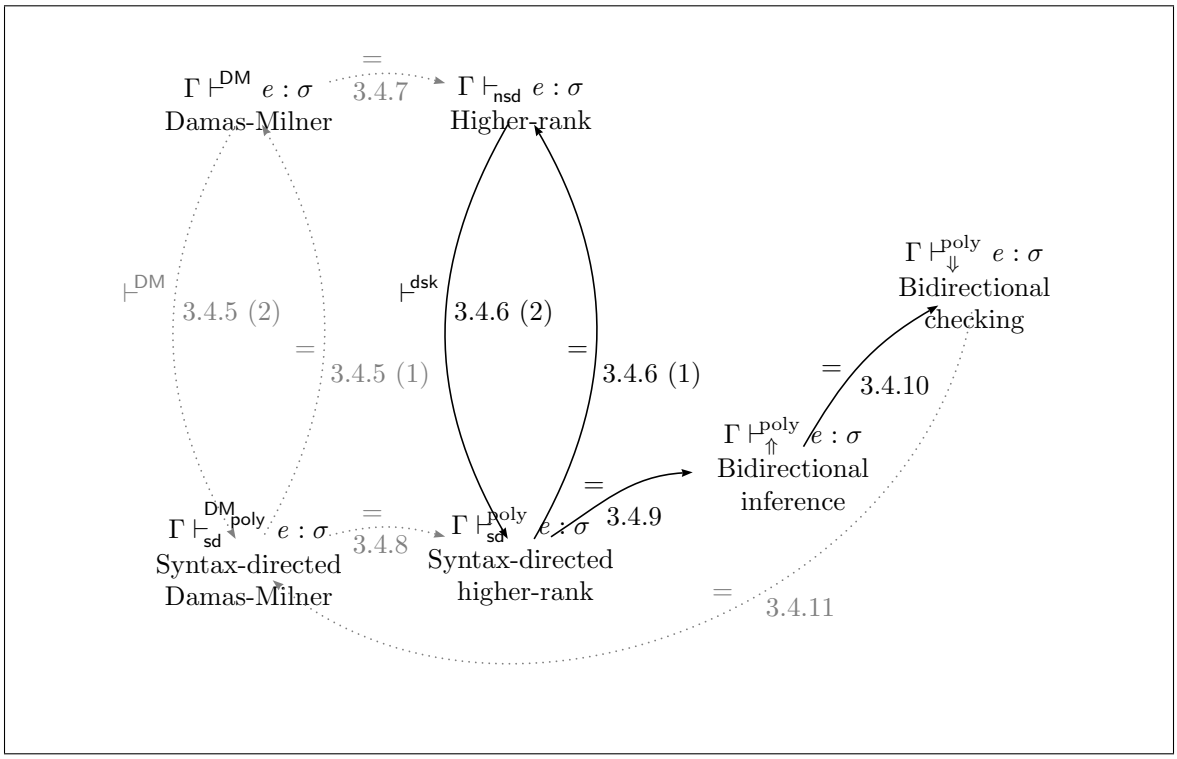

Figure 3.5: The world of predicative and higher-rank type systems

The results of this section are summarized in Figure 3.5. In some of these results, it matters whether we are talking about Damas-Milner types and terms, or higher-rank types and terms. In the figure, dashed lines correspond to connections where we assume that the types appearing in the judgements are only Damas-Milner types and that the terms contain no type annotations.

Some of the connections in this figure have already been shown. In particular the relation between the syntax-directed and the non-syntax-directed Damas-Milner type system is captured by the theorem below due to Damas and Milner.

Theorem 3.4.5. *Suppose that e contains no type annotations and the context Γ contains only Damas-Milner types.*

1. *If $\Gamma \vdash^{\mathsf{DM}}_{\mathsf{sd}} {}^{\mathsf{poly}} e : \sigma$ then $\Gamma \vdash^{\mathsf{DM}} e : \sigma$.*
2. *If $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ then there is a σ_0 such that $\Gamma \vdash^{\mathsf{DM}}_{\mathsf{sd}} {}^{\mathsf{poly}} e : \sigma_0$ and $\vdash^{\mathsf{DM}} \sigma_0 \leq \sigma$.*

We can show an analogous result for the higher-rank systems. We began our discussion of higher-rank

polymorphism with the Odersky-Läufer type system (Section 2.4), and developed a syntax-directed version of it (Section 3.1). Recall that with the Odersky-Läufer definition of subsumption, but without eager generalization, the two type systems did not agree. There are some programs that type check in the original version, but do not type check in the syntax-directed version (Section 3.1.1). By changing the subsumption relation in the syntax-directed version to deep skolemization, we can make it accept all of the programs accepted by the original type system.

However, it turns out that the two systems are still not equivalent: the syntax-directed system, using deep skolemization, accepts some programs that are rejected by the original typing rules! For example, the derivation

$$(x{:}\forall b\,.\,\texttt{Int} \to b) \vdash (x\,\texttt{::}\,\texttt{Int} \to \forall b\,.\,b) : \texttt{Int} \to \forall b\,.\,b$$

is valid in the syntax-directed version. But, because it uses deep skolemization in checking the type annotation, there is no analogue in the original system. Fortunately, if we replace subsumption in the original system with deep skolemization, the two type systems do agree.

Theorem 3.4.6 (Agreement of $\vdash_{\mathsf{nsd}}$ and $\vdash_{\mathsf{sd}}$)**.**

1. *If* $\Gamma \vdash^{\text{poly}}_{\mathsf{sd}} e : \sigma$ *then* $\Gamma \vdash_{\mathsf{nsd}} e : \sigma$.
2. *If* $\Gamma \vdash_{\mathsf{nsd}} e : \sigma$ *then there is a* σ_0 *such that* $\Gamma \vdash^{\text{poly}}_{\mathsf{sd}} e : \sigma_0$ *and* $\vdash^{\mathsf{dsk}} \sigma_0 \leq \sigma$.

The first two clauses of this theorem say that if a term can be typed by the syntax-directed system, then the non-syntax-directed system can also type it, and with the same type. The exact converse is not true; for example, in the non-syntax-directed system we have $\vdash_{\mathsf{nsd}} \lambda x\,.\,\lambda y\,.\,y : \forall a\,.\,a \to \forall b\,.\,b \to b$, but this type is not derivable in the syntax-directed system. Instead we have $\vdash^{\text{poly}}_{\mathsf{sd}} \lambda x\,.\,\lambda y\,.\,y : \forall ab\,.\,a \to b \to b$. In general, as clause (2) says, if a term in typeable in the non-syntax-directed system, then it is also typeable in the syntax-directed system, but perhaps with a different type that is at least as polymorphic as the original one.

Next, we show that the Odersky-Läufer system is an extension of the Damas-Milner system. Any term that type checks using the Damas-Milner rules, type checks with the same type using the Odersky-Läufer rules.

Theorem 3.4.7 (Odersky-Läufer extends Damas-Milner)**.** *Suppose e contains no type annotations and the context Γ only contains Damas-Milner types. If $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ then $\Gamma \vdash_{\mathsf{nsd}} e : \sigma$.*

Likewise, our version of the Odersky-Läufer syntax-directed system extends the Damas-Milner syntax-directed system.

Theorem 3.4.8 (Syntax-directed extension)**.** *Suppose e contains no type annotations and the context Γ only contains Damas-Milner types. If $\Gamma \vdash_{\mathsf{sd}}^{\mathsf{DM}_{\mathsf{poly}}} e : \sigma$ then $\Gamma \vdash_{\mathsf{sd}}^{\mathrm{poly}} e : \sigma$.*

Furthermore, the bidirectional system extends the syntax-directed system. Anything that can be inferred by Figure 3.1 can be inferred in the bidirectional system. The converse is not true, of course; the bidirectional system is designed to type check more programs.

Theorem 3.4.9 (Bidirectional inference extends syntax-directed system)**.**

1. *If $\Gamma \vdash_{\mathsf{sd}} e : \rho$ then $\Gamma \vdash_{\Uparrow} e : \rho$.*
2. *If $\Gamma \vdash_{\mathsf{sd}}^{\mathrm{poly}} e : \sigma$ then $\Gamma \vdash_{\Uparrow}^{\mathrm{poly}} e : \sigma$.*

Checking mode extends inference mode for the bidirectional system. If we can infer a type for a term, we should be able to check that this type can be assigned to the term.

Theorem 3.4.10 (Bidirectional checking extends inference)**.**

1. *If $\Gamma \vdash_{\Uparrow} e : \rho$ then $\Gamma \vdash_{\Downarrow} e : \rho$.*
2. *If $\Gamma \vdash_{\Uparrow}^{\mathrm{poly}} e : \sigma$ then $\Gamma \vdash_{\Downarrow}^{\mathrm{poly}} e : \sigma$.*

Finally, the bidirectional system is *conservative* over the Damas-Milner type system. If a term type checks in the bidirectional system without any higher-rank annotations, and with a monotype, then the term type checks in the syntax-directed Damas-Milner system, with the same type.

Theorem 3.4.11 (Bidirectional conservative over Damas-Milner)**.** *Suppose e contains no type annotations and Γ contains only Damas-Milner types. If $\Gamma \vdash_{\delta} e : \tau$ then $\Gamma \vdash_{\mathsf{sd}}^{\mathsf{HM}} e : \tau$.*

3.4.3 Properties of the bidirectional type system

The bidirectional type system, in Figure 3.3, is the main type system presented in this work. We turn to the two important properties of *type safety* and *principal types*.

As we defined the semantics of our language to be the semantics of System F plus a translation function, it suffices for type soundness to show that the translation to System F produces well-typed terms. This way we ensure that all terms accepted by the bidirectional system will evaluate without error. In other words:

Theorem 3.4.12 (Soundness of bidirectional system)**.**

1. *If* $\Gamma \vdash_\delta e : \rho \rightsquigarrow t$ *then* $\Gamma \vdash^{\mathsf{F}} t : \rho$.
2. *If* $\Gamma \vdash^{\text{poly}}_\delta e : \sigma \rightsquigarrow t$ *then* $\Gamma \vdash^{\mathsf{F}} t : \sigma$.

The proof of this theorem relies on a number of theorems that say that the coercions produced by the subsumption judgment are well typed.

The bidirectional type system enjoys a *principal types* property. In other words, for all terms typeable in a particular context *using the rules of our type system*, there is some "best" type for that term:

Theorem 3.4.13 (Principal types for bidirectional system)**.** *If there exists some* σ *such that* $\Gamma \vdash^{\text{poly}}_{\Uparrow} e : \sigma$*, then there exists* σ_0 *(the principal type of* e *in context* Γ*) such that*

1. $\Gamma \vdash^{\text{poly}}_{\Uparrow} e : \sigma_0$
2. *For all* σ*, if* $\Gamma \vdash^{\text{poly}}_{\Uparrow} e : \sigma$ *then* $\vdash^{\mathsf{DM}} \sigma_0 \leq \sigma$.

Notice that, in the second clause of the theorem, all types that are inferred for a given term are related by the Damas-Milner definition of subsumption, $\vdash^{\mathsf{DM}}$. The theorem holds *a fortiori* if $\vdash^{\mathsf{DM}}$ is replaced by $\vdash^{\mathsf{dsk}}$. We prove this theorem in the same way that Damas and Milner showed that the Damas-Milner type system has principal types: by developing an algorithm that unambiguously assigns types to terms and showing that this algorithm is sound and complete with respect to the rules. The full formalization of the algorithm can be found in the accompanying report [56], but we give a sketch of the implementation at the end of this section.

Crucial to showing that the algorithm is complete are the following two lemmas, which state that if our typing assumptions are strengthened along the Damas-Milner instance relation, the result of inference and checking is not affected. Consider for example the following program fragment:

```
g :: forall b. Int -> [forall a. a -> b]
h = g 3
```

Notice that `h` can get many different types, for instance $\forall b.[\forall a.a \to b]$, or $[\forall a.a \to \texttt{Int}]$, or $\forall c.[\forall a.a \to (c,c)]$. Nevertheless, the quantifier under the list constructor is determined because of the type annotation for `g`, and these three types can only differ according to top-level instantiations and generalizations. In other words, since the polymorphic structure of types *under* the top-level quantifiers is fully determined by programmer type annotations, the types arising in the specification can only differ to the types in the algorithm (the most general ones) in the shallow Damas-Milner sense. Hence, when the algorithm pushes a `let`-bound variable in the environment with a type, that algorithmic type is more general *only* in the Damas-Milner sense than any of the types that the specification would have assigned to the `let`-bound variable. Lemma 3.4.15 below states that this difference does not matter for typeability—consequently using more general types in the Damas-Milner sense preserves typeability. The notation $\vdash^{\mathsf{DM}} \Gamma_1 \leq \Gamma_2$ means that the type bounds in the environments are pointwise related along $\vdash^{\mathsf{DM}}$. On the contrary, Lemma 3.4.15 is not true if $\vdash^{\mathsf{DM}}$ is replaced with $\vdash^{\mathsf{OL}}$. If we have had to use Lemma 3.4.15 using $\vdash^{\mathsf{OL}}$ this would immediately ring a bell: It would mean that the type system *was indeed able* to guess arbitrary polymorphism under top-level quantifiers. Happily, the fact that we only need the Lemma 3.4.15 with $\vdash^{\mathsf{DM}}$ serves as a nice sanity check that our type system does not "guess" polymorphism.

Lemma 3.4.14 (Damas-Milner instance saturation of $\vdash^{\mathsf{inst}}$)**.**

1. *If* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$ *and* $\vdash^{\mathsf{inst}}_{\Downarrow} \sigma_2 \leq \rho$ *then* $\vdash^{\mathsf{inst}}_{\Downarrow} \sigma_1 \leq \rho$.
2. *If* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$ *and* $\vdash^{\mathsf{inst}}_{\Uparrow} \sigma_2 \leq \rho$ *then* $\vdash^{\mathsf{inst}}_{\Uparrow} \sigma_1 \leq \rho$.

Lemma 3.4.15 (Weakening)**.** *If* $\vdash^{\mathsf{DM}} \Gamma_1 \leq \Gamma_2$ *then the following are true:*

1. *If* $\Gamma_2 \vdash_\delta e : \rho$ *then* $\Gamma_1 \vdash_\delta e : \rho$ *for* $\delta = \Uparrow|\Downarrow$.
2. *If* $\Gamma_2 \vdash^{\mathrm{poly}}_{\Downarrow} e : \sigma$ *then* $\Gamma_1 \vdash^{\mathrm{poly}}_{\Downarrow} e : \sigma$.
3. *If* $\Gamma_2 \vdash^{\mathrm{poly}}_{\Uparrow} e : \sigma$ *then* $\Gamma_1 \vdash^{\mathrm{poly}}_{\Uparrow} e : \sigma_1$, *where* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma$.

The principal types theorem, Theorem 3.4.13, only deals with *inference* mode. An analogous version is not particularly informative for *checking* mode because we know exactly what type the term should

have—there is no ambiguity (modulo monomorphic information). And in fact, such a proposition is not true. For example, the term $\lambda g\,.\,(g\ 3, g\ \texttt{True})$ type checks in the empty environment with types $(\forall a\,.\,a \to \texttt{Int}) \to (\texttt{Int}, \texttt{Int})$ and $(\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$, but there is no type that we can assign to the term that is more general than both of these types.

Even though there are no most general types that terms may be assigned in checking mode, checking mode still satisfies properties that help programmers predict when their programs type check. For example, it is the case that if we can check a term, then we can always check it at a more specific type. The following theorem formalizes this, and other, claims:

Theorem 3.4.16.

1. *If* $\Gamma \vdash^{\text{poly}}_{\Downarrow} e : \sigma_1$ *and* $\vdash^{\mathsf{dsk}} \sigma_1 \leq \sigma_2$ *then* $\Gamma \vdash^{\text{poly}}_{\Downarrow} e : \sigma_2$.
2. *If* $\Gamma \vdash_{\Downarrow} e : \rho_1$ *and* $\vdash^{\mathsf{dsk}} \rho_1 \leq \rho_2$ *and* ρ_1 *and* ρ_2 *are in weak prenex form, then* $\Gamma \vdash_{\Downarrow} e : \rho_2$.
3. *If* $\Gamma_2 \vdash^{\text{poly}}_{\Downarrow} e : \sigma$ *and* $\vdash^{\mathsf{dsk}} \Gamma_1 \leq \Gamma_2$ *then* $\Gamma_1 \vdash^{\text{poly}}_{\Downarrow} e : \sigma$.
4. *If* $\Gamma_2 \vdash_{\Downarrow} e : \rho$ *and* $\vdash^{\mathsf{dsk}} \Gamma_1 \leq \Gamma_2$ *and* ρ *is in weak prenex form then* $\Gamma_1 \vdash_{\Downarrow} e : \rho$.

The first clause is self explanatory, but the second might seem a little surprising: why must ρ_1 and ρ_2 be in weak prenex form? Here is a counter-example when they are not. Suppose $\sigma_1 = \forall a\,.\,a \to \forall b\,.\,b \to \forall c\,.\,b \to c$, $\sigma_2 = \texttt{Int} \to \forall c\,.\,\texttt{Int} \to c$, and $\sigma_3 = \forall abc\,.\,a \to b \to b \to c$. Then it is derivable that $\vdash_{\Downarrow} \lambda x\,.\,x\ 3 : \sigma_1 \to \sigma_2$ but it is not derivable that $\vdash_{\Downarrow} \lambda x\,.\,x\ 3 : \sigma_3 \to \sigma_2$, although $\vdash^{\mathsf{dsk}} \sigma_1 \to \sigma_2 \leq \sigma_3 \to \sigma_2$. However, because GEN2 converts the checked type into that form before continuing, any pair of related types may be used for the $\vdash^{\text{poly}}_{\Downarrow}$ judgement, so the first clause needs no side condition. Just as the first two clauses say that we can make the result type *less* polymorphic, dually, the third and fourth clauses allow us to make the environment *more* polymorphic. Notice, that in contrary to the inference case (Lemma 3.4.15), we can arbitrarily generalize the environments along $\vdash^{\mathsf{dsk}}$, instead of only $\vdash^{\mathsf{DM}}$.

Despite the fact that the bidirectional type system possesses principal types in the sense described above, it still does not satisfy the `let` expansion property mentioned in the introduction. In some cases one has to provide an explicit type annotation to the `let`-bound variable that will abstract all the common sub-expressions; but there are cases where even this is not possible, because of

contextual type information. In particular one can imagine a rank-3 context where $f_1 : ((\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})) \to \ldots$ and $f_2 : ((\forall a\,.\,a \to \texttt{Int}) \to (\texttt{Int}, \texttt{Int})) \to \ldots$, and $\mathcal{C}[\cdot] = (f_1\ \cdot, f_2\ \cdot)$. Then, while it is true that $\mathcal{C}[(\lambda g\,.\,(g\ 3, g\ \texttt{True}))]$ is typeable, there is no single System F type that can be used as the type of $\lambda g\,.\,(g\ 3, g\ \texttt{True})$ in a `let`-bound definition, even if we are willing to annotate the `let`-bound definition with an explicit type signature.

3.5 On the algorithmic implementation

The syntax-directed type system of Figure 3.3 does not fully specify an inference algorithm. At certain points in the syntax-directed system, guessing is still required—for example, in the rule INST, the rules do not specify what types $\overline{\tau}$ should be used to instantiate the bound variables of a polytype. Because of this guess, typing is non-deterministic. By making different choices for $\overline{\tau}$ we can show that a given term has many different types. There exists however a type inference algorithm [56] for the bidirectional Odersky-Läufer system, based on the Hindley-Damas-Milner algorithm. The algorithm, exactly like the Hindley-Damas-Milner algorithm, makes use of *unification variables*. Whenever we must "guess" monotypes, such as in rule INST, we make up instead fresh unification variables. We additionally carry around an idempotent substitution that maps unification variables to *monotypes* (possibly involving other unification variables). The fact that unification variables range only over monotypes is because the type system is predicative. As the algorithm progresses, we generate equality constraints, which we solve by unification, extending the current substitution to reflect this solution [56].

3.6 Summary of the work on higher-rank type systems

The bidirectional type inference system that has been presented is an application of local type inference for predicative higher-rank polymorphism. It uses the idea of bidirectional typing judgements, but employs global type argument synthesis. It relies on first-order unification, is not harder to implement than the standard Hindley-Damas-Milner algorithm [41], and has formed the basis of a successful implementation in previous versions of GHC. Although it cannot type all System F

programs because it is predicative, it is simple enough to explain and present to programmers and conservatively extends Damas-Milner; in fact, it assigns to Damas-Milner typeable programs exactly the same types that the Damas-Milner type system would do. On the other hand, it goes beyond System F in a different axis: it relies on a more elaborate type instance relation.

Chapter 4

Local type argument synthesis for impredicativity

Because of the implementation simplicity of the higher-rank type systems, local type inference that relies on ordinary first-order unification appears to be a tempting approach for impredicative polymorphism. The idea is this: use aggressive annotation propagation so that impredicative instantiations may be synthesized locally—meaning within the premises of a single type inference rule.

Indeed, in many common cases the impredicative instantiation of type variables can be determined locally. Consider:

```
g   :: [forall c. c -> c] -> [forall c. c -> c]
Nil :: forall a. [a]
```

When attempting to infer a type for the application `g Nil`, The Damas-Milner type system and the predicative higher-rank type systems in Chapter 3 would instantiate the quantified variable of `Nil` with a monomorphic type, causing the example to be ill-typed. But in this case, a different approach suggests itself. Given that `g` requires type $[\forall c\,.\,c \to c]$ from its argument, there is only one

instantiation of the quantified variable of `Nil` that can make the application well-typed, $\forall c\,.\,c \to c$; and we can learn this information locally, without having to use global unification.

The Boxy Types proposal [58, 57] (to be described in detail in Chapter 8) is a previous attempt to allow for impredicative instantiation of type variables using the aforementioned idea. Hence, Boxy Types exploit locally known information at a function call to determine potentially impredicative instantiations of quantified variables.

Intuitively, Boxy Types is an attempt to reconcile the guessing of monomorphic information, which is eventually resolved through global first-order unification, and local discovery of polymorphic information. In order to infer a type for the application `g Nil` we first infer the type $? \to ?$ for `g`. But we immediately learn that this ?-type must be $\forall c\,.\,c \to c$. Next, we check that the argument `Nil` can be assigned type $[\forall c\,.\,c \to c]$, but instead of instantiating the quantified variable of `Nil` with monomorphic types, we instantiate it with a different unknown type, which can also be determined locally to be $\forall c\,.\,c \to c$.

4.1 A critique against local type argument synthesis

A fair critique against local type argument synthesis is that sometimes no polymorphic information is available locally. Additionally, a particular flow of polymorphic information is often (e.g. in Boxy Types) hardcoded in the typing rules. For example, the vanilla Boxy Types cannot accept the application `length ids` where:

```
length :: forall a. [a] -> Int
ids    :: [forall a. a -> a]
```

since the types of functions are inferred prior to examining the types of arguments, and hence the instantiation of `length` becomes monomorphic. The only possibility to type this program would be if the flow of polymorphic information in application nodes were reversed from arguments to functions. Additionally, Boxy Types suffers from little robustness under program transformations, since some transformations break locality.

These problems generalize to systems based on local type inference that hard-code the flow of polymorphic information and locally synthesize polymorphic instantiations. Consider the following program fragment:

```
length :: forall a.[a] -> Int
ids    :: [forall a.a->a]
f      :: [forall b.b->b] -> Int
Nil    :: forall a.[a]

h0 = length ids

h1 = f Nil

h2 :: [forall a.a -> a]
h2 = cons (\x -> x) Nil

h3 = cons (\x -> x) (reverse ids)
```

The functions `h0`, `h1`, `h2`, and `h3` show that reasonable impredicative instantiations may require quite different approaches in the choices that the type system makes about polymorphic information flow. For example, the impredicative instantiation of `length` in the definition of `h0` requires a flow of polymorphic information from the argument `ids` to the function `length`. In contrast, in `h1`, it is the polymorphic information of the type of the function `f` that determines the impredicative instantiation of the argument `Nil`. Moreover, in the case of `h2` it is a type annotation around the whole application (i.e. on the result type of the application) that determines the polymorphic instantiation of `Nil`. To complicate things even more, if no annotation is provided we want this Damas-Milner typeable program to remain typeable with its Damas-Milner type. Finally, in `h3`, some subexpression deep in the term (`ids`) determines the impredicative instantiation of `cons`.

Apart from the problem of committing to a particular polymorphic information flow in the type system specification, local type argument synthesis becomes complicated due to implicit instantiations of function arguments. For example, in `h3`, the argument `(\x -> x)` can get type $\forall a\,.\,a \to a$, which can also be implicitly instantiated to $\sigma \to \sigma$. Which type do we synthesize as the type argument for `cons`? It is only *later*—i.e. outside the local premises of the application rule for `cons (\x -> x)`—that we learn that the correct type argument must be $\forall a\,.\,a \to a$.

What the examples—and many more are reproducible—show is that it is hard to combine all the possible flows of polymorphic information in a specification that uses local information to determine impredicative instantiations.

4.2 Global type argument synthesis

Returning to the application `length ids` above, the polymorphic instantiation of `length` should not be too complicated to figure out; in the particular example it is straightforward to see that the quantified variable of the type of `length` should be instantiated to $\forall a\,.\,a \to a$. How one can implement this algorithmically, without committing to a particular polymorphic information flow path?

Intuitively, for this last application, if we infer a type for the function and the argument, we must compare the inferred type of the argument against a *yet unknown instantiation* of the quantified variable of `length`'s type, according to a suitable instance relation. We may introduce hence unification variables that abstract yet-unknown potentially impredicative instantiations of previously quantified variables. We call such variables *constrained variables* (to emphasize that they generalize ordinary unification variables with constraints) because all the information about what types they stand for is returned in suitable *constraints*. In an application node, when a polymorphic function type is instantiated with constrained variables, the inferred type of the argument produces suitable constraints on these variables. In contrast to local type inference, we may choose to resolve these constraints later, when we obtain full knowledge of the type inference problem.

Typing the following program shows how this delaying solves the locality problems of Boxy Types:

```
f   :: forall a. a -> [a] -> Bool
ids :: [forall a. a -> a]
foo = f (\x -> x) ids
```

To infer a type for `foo`, we can instantiate the type of `f` with a constrained variable α to get type $\alpha \to [\alpha] \to \texttt{Bool}$. Subsequently, we may infer a type for `\x -> x`, $\forall a\,.\,a \to a$, and produce a constraint that witnesses the fact that α must be an instance of $\forall a\,.\,a \to a$. We write this with $(\alpha \geq \forall a\,.\,a \to a)$. This constraint is satisfiable with $\alpha = \texttt{Int} \to \texttt{Int}$, or $\alpha = \forall a\,.\,a \to a$, or many other types but we cannot determine yet which is the one we should choose. If we were in Damas-Milner, where α could only be mapped to a monomorphic type, we would know already that α must simply be equated to some $\beta \to \beta$, but in the presence of impredicative instantiation we cannot assume the same. So we instead *delay the instantiation* and proceed to the second argument. Comparing the type of `ids` ($[\forall a\,.\,a \to a]$) against the *expected* type $[\alpha]$ gives rise to the constraint $(\alpha = \forall a\,.\,a \to a)$, where as in System F the instance relation is invariant on any type constructor. From the union of the two constraints, $\{\alpha \geq \forall a\,.\,a \to a, \alpha = \forall a\,.\,a \to a\}$, we deduce that the only solution is $\alpha = \forall a\,.\,a \to a$. In the Boxy Types system, the application `f (\x -> x) ids` would be ill-typed: essentially right after checking the first argument, `\x -> x`, the type that would instantiate `a` would have to be forced to *a monomorphic type*, because locally there is no polymorphic information present.

This suggests, algorithmically at least, that a more effective approach to the problem is global type inference, that delays as much as possible the commitment to particular instantiations. This idea is present in the ML^F language, and also underlies the algorithmic implementation of FPH, to be presented in the next chapter.

Chapter 5

FPH: Type inference for impredicative higher-rank types

We now turn the focus on the FPH type system and its algorithmic implementation: a type system and the associated type inference algorithm that lift both restrictions of Damas-Milner. The main idea is the following: internally, the type inference implementation uses types with constraints and global type argument synthesis in the style of ML^F [26], but this internal sophistication is never shown to the programmer; it is simply the mechanism used by the implementation to support a simple declarative specification, which only uses guessed System F solutions to the underlying constraints. Thus, the FPH specification remains within the type structure of System F. Of course, naïvely lifting either of the two restrictions of Damas-Milner (allowing higher-rank or impredicative types) leads to lack of principal types for expressions. As a consequence, FPH has to use mechanisms that restrict the set of System F typeable programs that type check without any type annotations.

We start with an introduction to FPH, and proceed with giving the formal declarative specification of the type system and the formal properties that address the issues of expressiveness, placement of type annotations, type soundness, and robustness. We finally give a syntax-directed specification to serve as an intermediate step leading to the algorithmic implementation of FPH (to be described in Chapter 6).

5.1 Introduction to FPH

To give a flavor of FPH and motivate its design principles, we use several examples. Consider this program fragment:

```
str :: [Char]
ids :: [forall a. a->a]
length :: forall b. [b] -> Int

l1 = length str
l2 = length ids
```

First consider type inference for `l1`. The polymorphic `length` function returns the length of its argument list. In the standard Damas-Milner type system, one instantiates the type of `length` with `Char`, so that the occurrence of `length` has type [`Char`] $\rightarrow$ `Int`, which matches correctly `length`'s argument, `str`. Since in Damas-Milner a polymorphic function can only be instantiated with a monotype, `l2` is untypeable, because to type its definition we must instantiate `length` with $\forall a.a \rightarrow a$. We cannot simply lift the Damas-Milner restriction, because that directly leads to different choices giving incomparable types. However, `l2` also shows that there are benign uses of impredicative instantiation. Although we need an impredicative instantiation to make `l2` type check, there is no danger here—the type of `l2` will always be `Int`. It is only when a `let`-bound definition can be assigned two ore more incomparable types that we run into trouble.

Our idea is to mark impredicative instantiations so that we know when an expression may be typed with different incomparable types. Technically, this means that we instantiate polymorphic functions with a form of type τ' that is more expressive than a mere monotype, but less expressive than an arbitrary polymorphic type:

$$\begin{array}{lcl} \tau' & ::= & a \mid \tau'_1 \rightarrow \tau'_2 \mid \mathtt{T}\,\tau' \mid \boxed{\sigma} \\ \sigma & ::= & \forall a.\sigma \mid a \mid \sigma \rightarrow \sigma \mid \mathtt{T}\,\sigma \end{array}$$

Unlike a monotype τ, a *boxy monotype* τ' may contain polymorphism, but only inside a box, thus $\boxed{\sigma}$. We hence allow polymorphic expressions to be *instantiated with boxy monotypes*. A boxy type marks the place in the type where "guessing" is required to fill in a type that makes the rest of the typing derivation go through.

Now, when typing `l2` we may instantiate `length` with $\boxed{\forall a.a \to a}$. Then the application `length ids` has a function expecting an argument of type $[\boxed{\forall a.a \to a}]$, applied to an argument of type $[\forall a.a \to a]$. These types match, if we discard all the boxes around them. That completes the typing of `l2`.

Boxes are ignored when typing an application, but they play a critical role in `let`-bound polymorphism. Since the only ambiguity can arise because of guessed polytypes we naturally require that the type environment contains no boxes.

Let us return to the example `bar = choose id` given above. If we instantiated `choose` with the boxy monotype $\boxed{\forall a.a \to a}$, the application `(choose id)` would type just fine, but its result type would be $\boxed{\forall a.a \to a} \to \boxed{\forall a.a \to a}$. However, we prevent that box-containing type from entering the environment as the type for `bar`, so this instantiation for `choose` is rejected. If we instead instantiate `choose` with $c \to c$, the application again type checks (this time by additionally instantiating the type of `id` with c), so the application has type $(c \to c) \to c \to c$, which can be generalized and then enter the environment as the type of `bar`. This type is the principal Damas-Milner type of `bar`—all Damas-Milner types for `bar` are also available without annotation. What we have achieved effectively is that, instead of having two or more incomparable types for `bar`, we have allowed only those System F typing derivations for `bar` that admit a principal type.

However, if the programmer actually wanted the other, rich, type for `bar`, she can use a type annotation:

```
bar = choose id :: (forall b.b->b) -> (forall b.b->b)
```

Such type annotations are typed using the same idea as application nodes—when typing an annotated expression $e\texttt{::}\sigma$, we may ignore boxes on e's type when comparing with σ (which is box-free, being a programmer annotation). Now we may instantiate `choose` with $\boxed{\forall a.a \to a}$, because the type annotation is compatible with the type of `(choose id)`, $\boxed{\forall a.a \to a} \to \boxed{\forall a.a \to a}$.

Expressive power As we have seen, a type annotation may be required on a `let`-bound expression, but annotations are never required on function applications, even when they are nested and higher order, or involve impredicativity. Here is the `runST` example with some variants:

```
runST  :: forall a. (forall s. ST s a) -> a
app    :: forall a b. (a -> b) -> a -> b
revapp :: forall a b. a -> (a -> b) -> b
arg    :: forall s. ST s Int


h0 = runST arg
h1 = app runST arg
h2 = revapp arg runST
```

All definitions `h0`, `h1`, `h2` are typeable without annotation because, in each case, the return type is a (non-boxy) monotype `Int`.

Actually, we have a much more powerful guideline for programmers, which does not even require them to think about boxes:

> **FPH annotation guideline.** Programmers may write programs as they would in implicitly typed System F (extended with `let`-bound definitions), and place annotations on `let`-bindings and λ-abstractions that must be typed with rich types.

To demonstrate the guideline in practice, assume that Γ contains the bindings of the previous example and consider the System F derivation for the expression $\texttt{let}\ h = \texttt{app runST arg in}\ h$ below:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{app} : ((\forall s.\texttt{ST}\ s\ \texttt{Int}) \to \texttt{Int}) \to (\forall s.\texttt{ST}\ s\ \texttt{Int}) \to \texttt{Int} \quad \Gamma \vdash \texttt{runST} : (\forall s.\texttt{ST}\ s\ \texttt{Int}) \to \texttt{Int} \quad \Gamma \vdash \texttt{arg} : (\forall s.\texttt{ST}\ s\ \texttt{Int})}{\Gamma \vdash \texttt{app runST arg} : \texttt{Int}} \qquad \Gamma, (h{:}\texttt{Int}) \vdash h : \texttt{Int}}{\Gamma \vdash \texttt{let}\ h = \texttt{app runST arg in}\ h : \texttt{Int}}$$

The corresponding derivation in FPH is the same, except that all polymorphic instantiations are marked with boxes that are ignored when matching the types of arguments with the types that functions expect in applications:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{app} : (\boxed{\forall s\,.\,\texttt{ST}\ s\ \texttt{Int}} \to \texttt{Int}) \to \boxed{\forall s\,.\,\texttt{ST}\ s\ \texttt{Int}} \to \texttt{Int} \quad \Gamma \vdash \texttt{runST} : (\forall s\,.\,\texttt{ST}\ s\ \texttt{Int}) \to \texttt{Int} \quad \Gamma \vdash \texttt{arg} : (\forall s\,.\,\texttt{ST}\ s\ \texttt{Int})}{\Gamma \vdash \texttt{app runST arg} : \texttt{Int}} \qquad \Gamma, (h{:}\texttt{Int}) \vdash h : \texttt{Int}}{\Gamma \vdash \texttt{let}\ h = \texttt{app runST arg in}\ h : \texttt{Int}}$$

Notice that, since the type of the `let`-bound expression `app runST arg` is the box-free type `Int`, there is no need for placing any type annotation to make the program typeable.

The guideline implies that for a term consisting of applications and variables to be `let`-bound (without any type annotations), it *does not matter* what impredicative instantiations may happen to type it, provided that the result type is an ordinary Damas-Milner type. For example, the argument `choose id` to the function `f` below involves an impredicative instantiation (in fact for both `f` *and* `choose`), but no annotation is required whatsoever:

```
f :: forall a. (a -> a) -> [a] -> a
g = f (choose id) ids
```

In particular `choose id` gets type $\boxed{\forall a.a \to a} \to \boxed{\forall a.a \to a}$. However, `f`'s arguments types match (ignoring boxes), and its result type (again, ignoring boxes) is a Damas-Milner type ($\boxed{\forall a\,.\,a \to a}$), and hence no annotation is required for `g`.

Since the annotation guideline does not require the programmer to think about boxes at all, why does our specification use boxes? The answer is because the annotation guideline is conservative: it guarantees to make the program typeable, but it demands more annotations than are necessary. For example:

```
f' :: forall a. [a] -> [forall b. b -> b]
g' = f' ids
```

Notice that the rich result type `[forall b. b -> b]` is non-boxy, and hence no annotation is required for `g'`. In general, even if the type of a `let`-bound expression is rich, if that type does not result from impredicative instantiation (which is the common case), then no annotations are required. Boxes precisely specify what "that type does not result from impredicative instantiation" means. Nevertheless, a box-free specification is an attractive alternative design, as we discuss in Section 7.2.

Conservativity for a declarative specification Although the FPH system, as we have described it so far, is expressive, it is also somewhat conservative. It requires annotations in a few instances, even when there is only one type that can be assigned to a `let`-bound definition, as the following example demonstrates.

```
f :: forall a. a -> [a] -> [a]
ids :: [forall a. a -> a]

h1 = f (\x -> x) ids            -- Not typeable
h2 = f (\x -> x) ids :: [forall a. a->a]  -- OK
```

Here `f` is a function that accepts an element and a list and returns a list (for example, `f` could be `cons`). Definition `h1` is not typeable in FPH. We can attempt to instantiate `f` with $\boxed{\forall a.a \to a}$, but then the right hand side of `h1` has type $[\boxed{\forall a.a \to a}]$, and that type cannot enter the environment. The problem can of course be fixed by adding a type annotation, as `h2` shows.

One may think that it is overly conservative to require a type annotation in `h2`; after all, `h1` manifestly has only one possible type. But suppose that `f` had type $\forall ab.a \to b \to [a]$, which is a more general Damas-Milner type than the type above. This can happen if, for instance, `f` was actually defined:

```
f x y = single x
```

where `single` : $\forall a\,.\,a \to [a]$ creates a list with a single element. Now, the type signature we gave for `f` before is not its most general type; the latter is $\forall ab\,.\,a \to b \to [a]$. With this type for `f`, our example `h1` now has *two incomparable types*, namely $[\forall a.a \to a]$ as before, and $\forall a\,.[a \to a]$.

$$
\begin{array}{lllll}
\text{Types} & \sigma & ::= & \forall \overline{a}\,.\,\rho \\
 & \rho & ::= & \tau \mid \sigma \to \sigma \\
 & \tau & ::= & a \mid \tau \to \tau \\
 & & & \\
\text{Boxy Types} & \sigma' & ::= & \forall \overline{a}\,.\,\rho' \\
 & \rho' & ::= & \tau' \mid \sigma' \to \sigma' \\
 & \tau' & ::= & a \mid \boxed{\sigma} \mid \tau' \to \tau' \\
 & & & \\
\text{Environments} & \Gamma & ::= & \Gamma, (x{:}\sigma) \mid \cdot
\end{array}
$$

Figure 5.1: Syntax of FPH

Without any annotations we presumably have to choose the same type as the Damas-Milner type system would; and that might make occurrences of `h1` in its scope ill typed.

In short, making the type of `f` more general along the Damas-Milner instance relation has caused definitions in the scope of `h1` to become ill-typed. This threatens type inference completeness, and that is the reason that we reject `h1`, requiring an annotation as in `h2`. Requiring an annotation on `h2` may seem an annoyance to programmers, but it is this conservativity of FPH that results in a simple and declarative high-level specification. FPH allows `let`-bound definitions to enter environments with many different types, as is the case in the Damas-Milner type system.

We now turn our attention to a detailed account of the FPH type system, which supports higher-rank and impredicative types. We give here a declarative specification of the type system, present its properties, give a syntax-directed specification, and sketch the implementation of the syntax-directed specification.

5.2 Declarative specification of the FPH type system

For a systematic treatment of FPH, we begin with the syntax of types and environments in Figure 5.1. Types are divided into ordinary box-free types σ-, ρ-, and τ-types, and boxy types σ', ρ', and τ' types. Polymorphic types, σ and σ', may contain quantifiers at top-level, whereas ρ and ρ' types contain only nested quantifiers. For the examples we assume the existence of type constructors such as lists or pairs. The important difference between box-free and boxy types occurs at the

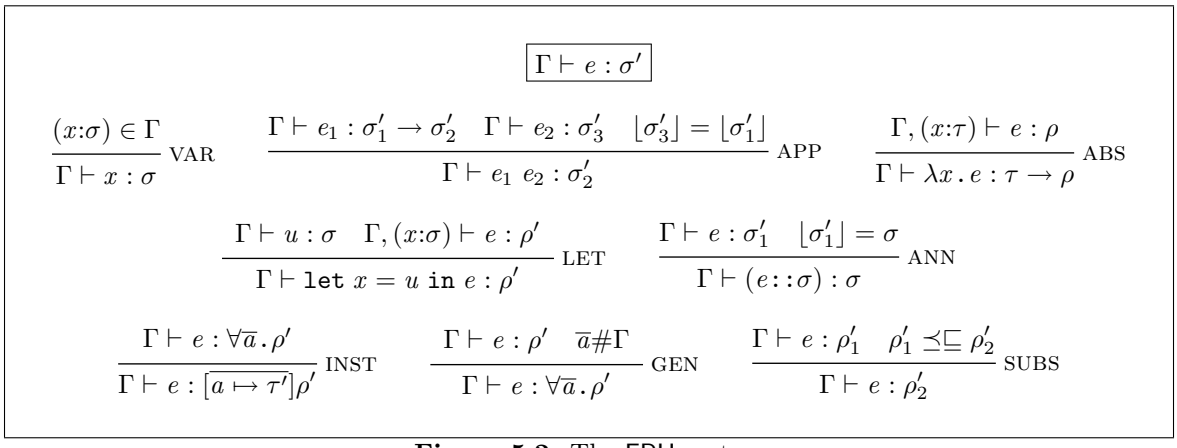

Figure 5.2: The FPH system

monotype level. Following previous work by Rémy *et al.* [6, 26], τ' may include boxes containing (box-free) polytypes. As we discussed in Section 5.1, these boxes represent the places where "guessed instantiations" take place. The syntax of type environments, Γ, does not allow types with boxes—all uncertainty about impredicative instantiations has to be resolved prior to pushing a variable into the environment. As a consequence, environments include only bindings of the form $(x{:}\sigma)$.

5.2.1 Typing rules

The declarative (i.e. not syntax-directed) specification of FPH is given in Figure 5.2. As usual, the judgement form $\Gamma \vdash e : \sigma'$ assigns the type σ' to the expression e in typing environment Γ. A non-syntactic invariant (i.e. can be proven as a separate lemma) of the typing relation is that, in the judgement $\Gamma \vdash e : \forall \overline{a}.\rho'$, no box may intervene between a variable quantified in ρ' and the occurrences of that variable. Thus, for example, ρ' cannot be of form $(\forall b.\boxed{b}) \to \texttt{Int}$, because the quantified variable b appears inside a box. The top-level quantified variables may, however, appear inside boxes.

The rules in Figure 5.2 are modest variants of the conventional Damas-Milner rules. Indeed rule VAR is precisely as usual, simply returning the type of a variable from the environment.

Rule APP infers a function type $\sigma'_1 \to \sigma'_2$ for e_1, infers a type σ'_3 for the argument e_2, and checks that the argument type matches the domain of the function type *modulo boxy structure*, as explained

in Section 5.1. This compatibility check is performed by *stripping* the boxes from σ'_1 and σ'_3, then comparing for equality. The notation $\lfloor \sigma' \rfloor$ denotes the non-boxy type obtained by discarding the boxes in σ':

Definition 5.2.1 (Stripping)**.** We define the strip function $\lfloor \cdot \rfloor$ on boxy types as follows:

$$\begin{array}{lcll} \lfloor a \rfloor & = & a & \\ \lfloor \boxed{\sigma} \rfloor & = & \sigma & \\ \lfloor \sigma'_1 \to \sigma'_2 \rfloor & = & \lfloor \sigma'_1 \rfloor \to \lfloor \sigma'_2 \rfloor & \\ \lfloor \forall \overline{a} \, . \, \rho' \rfloor & = & \forall \overline{a} \, \overline{b} \, . \, \rho & \text{where } \lfloor \rho' \rfloor = \forall \overline{b} \, . \, \rho \end{array}$$

The equality between types used in rule APP is ordinary α-equivalence. Stripping is also used in rule ANN, which handles expressions with explicit programmer-supplied type annotations. It infers a boxy type for the expression and checks that, modulo its boxy structure, it is equal to the type required by the annotation σ. In effect, this rule converts the boxy type σ'_1 that was inferred for the expression to a box-free type σ. If the annotated term is the right-hand side of a `let` binding, as in `let` $x = (e$ `::` $\sigma)$ `in` ..., this box-free type σ can now enter the environment as the type of `x` (whereas σ'_1 could not if it contained boxes).

Rule ABS infers types for λ-abstractions. It first extends the environment with a *monomorphic, box-free* binding $x{:}\tau$, and infers a ρ-type for the body of the abstraction. Notice that we insist (syntactically) that the result type ρ both (a) has no top-level quantifiers, and (b) is box-free. We exclude top-level quantifiers because we wish to attribute the same types as Damas-Milner for programs that are typeable by Damas-Milner; in the vocabulary of Chapter 3, we avoid *eager generalization*. Choice (b), that a λ-abstraction must return a box-free type, may require more programmer annotations, but turns out to permit a much simpler type inference algorithm. We return to this issue in Section 7.2.

Rule ABS is the main reason that the type system of Figure 5.2 cannot type all of System F, even with the addition of type annotations: ABS allows only abstractions of type $\tau \to \rho$, whereas System F allows λ-abstractions of type $\sigma_1 \to \sigma_2$. Rule ABS is however just enough to demonstrate our approach to impredicative instantiation, while previous work [41] has shown how to address this

limitation. It is easy to combine the two, as we show in Section 5.2.4.

Following the ideas outlined in Section 5.1, rule LET first infers a *box-free* type σ for the right-hand side expression u, and then checks the body pushing the binder x with type σ in the environment.

Generalization (GEN) takes the conventional form, where $\overline{a}\#\Gamma$ means that $\overline{a}$ is disjoint from the free type variables of Γ. In this rule, note that the generalized variables $\overline{a}$ may appear inside boxes in ρ', so that we might, for example, infer $\Gamma \vdash e : \forall a.\boxed{a} \to a$.

Instantiation (INST) is mostly conventional, but it allows us to instantiate a type with a *boxy* monotype τ' instead of just a box-free τ. However, we need to be a little careful with substitution in INST: since ρ' may contain $\overline{a}$ inside boxes, a naive substitution might leave us with nested boxes, which are syntactically ill-formed. Hence, we define a form of substitution that preserves the boxy structure of its argument.

Definition 5.2.2 (Monomorphic substitutions)**.** We use letter φ for *monomorphic substitutions*, that is, finite maps of the form $[\overline{a \mapsto \tau'}]$. We let $\mathit{ftv}(\varphi)$ be the set of the free variables in the range and domain of φ. We define the operation of applying φ to a type σ' as follows:

$$
\begin{array}{lcll}
\varphi(a) & = & \tau' & \text{where } [a \mapsto \tau'] \in \varphi \\
\varphi(\boxed{\sigma}) & = & \boxed{\lfloor \varphi(\sigma) \rfloor} & \\
\varphi(\sigma'_1 \to \sigma'_2) & = & \varphi(\sigma'_1) \to \varphi(\sigma'_2) & \\
\varphi(\forall \overline{a} \,.\, \rho') & = & \forall \overline{a} \,.\, \varphi(\rho') & \text{where } \overline{a}\#\mathit{ftv}(\varphi)
\end{array}
$$

We write $[\overline{a \mapsto \tau'}]\sigma'$ for the application of the $[\overline{a \mapsto \tau'}]$ to σ'.

For example, we have $[a \mapsto \boxed{\sigma}](a \to \boxed{a}) = \boxed{\sigma} \to \boxed{\sigma}$, and $[a \mapsto (\tau \to \boxed{\tau})](a \to \boxed{a}) = (\tau \to \boxed{\tau}) \to \boxed{\tau \to \tau}$.

5.2.2 The subsumption rule

The final rule, SUBS, has an important role. It ensures that boxes "don't get in the way" by allowing guessed (boxed) polymorphic types to get instantiated further, or guessed (boxed) function types to be used as ordinary function types. However, this is done without forgetting that the type has been

guessed (is boxed) in the first place, except in the case the guessed type is monomorphic. In that case it is safe for subsumption to completely remove the boxes off the type.

Consider the code fragment in Example 5.2.3.

Example 5.2.3 (Boxy instantiation)**.**

```
head :: forall a. [a] -> a
ids  :: [forall a. a -> a]
h = head ids 3
```

Temporarily ignoring rule SUBS, the application `head ids` can get type $\boxed{\forall a\,.\,a \to a}$, and only that type. Hence, the application `(head ids) 3` cannot be typed. This situation would be rather unfortunate as one would, in general, have to use type annotations to extract polymorphic expressions from polymorphic data structures. For example, programmers would have to write:

```
h = (head ids :: forall b. b -> b) 3
```

This situation would also imply that some expressions which consist only of applications of closed terms, and are typeable in System F, could not be typed in FPH.

Rule SUBS addresses these limitations. It modifies the types of expressions in two ways with the relation $\preceq\!\!\sqsubseteq$, which is the composition of two relations, $\preceq$ and $\sqsubseteq$ in Figure 5.3.

The relation $\preceq$, called *boxy instantiation*, simply instantiates (perhaps impredicatively) a polymorphic type within a box. Rule BI does exactly this, whereas rule BR ensures reflexivity of $\preceq$. As an example, it is derivable that $\boxed{\forall a\,.\,a \to a} \preceq \boxed{(\forall b\,.\,b \to b) \to (\forall b\,.\,b \to b)}$.

The relation $\sqsubseteq$, called *protected unboxing*, removes boxes around monomorphic types and pushes boxes congruently down the structure of types. The most important rules of this relation are TBOX and REFL. The first simply removes a box around a monomorphic type, while the second ensures reflexivity. If a ρ' type contains only boxes with monomorphic information, then these boxes can be completely dropped along the $\sqsubseteq$ relation to yield a box-free type. Notice that rule SUBS only uses ρ'-types—hence rule POLY of $\sqsubseteq$ may only be triggered for non-top-level quantified types. This

means that the quantified variables $\overline{a}$ are always going to be unboxed inside the bodies in ρ' in all use sites of the relation, but we chose to put the side-conditions there to emphasize this well-formedness invariant.

Because SUBS uses $\preceq\!\sqsubseteq$ instead of merely $\sqsubseteq$, `h` in Example 5.2.3 is typeable. When we infer a type for `head ids`, we may have the following derivation:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{head ids} : \boxed{\forall a\,.\,a \to a} \qquad \boxed{\forall a\,.\,a \to a} \preceq \boxed{a \to a} \sqsubseteq a \to a}{\Gamma \vdash \texttt{head ids} : a \to a}\,\text{SUBS}}{\Gamma \vdash \texttt{head ids} : \forall a\,.\,a \to a}\,\text{GEN}$$

Therefore, no annotation is required on `h` and, because the $\sqsubseteq$ relation can remove boxes around monomorphic types, it also follows that we can bind the `head ids` expression in a definition:

```
let h = (head ids) in ()
```

More generally, we have the following lemma.

Lemma 5.2.4. *If* $\Gamma \vdash e : \boxed{\forall \overline{a}\,.\,\tau}$ *then* $\Gamma \vdash e : \forall \overline{a}\,.\,\tau$.

Proof. By rule BI we have $\boxed{\forall \overline{a}\,.\,\tau} \preceq \boxed{\tau}$ where without loss of generality $\overline{a}\#\Gamma$, and by rule TBOX we get $\boxed{\tau} \sqsubseteq \tau$, hence $\boxed{\forall \overline{a}\,.\,\tau} \preceq\!\sqsubseteq \tau$. Applying rule SUBS to the derivation of $\Gamma \vdash e : \boxed{\forall \overline{a}\,.\,\tau}$ gives $\Gamma \vdash e : \tau$, and by rule GEN we get $\Gamma \vdash e : \forall \overline{a}\,.\,\tau$. □

Finally notice that it is not sound to combine the relations $\preceq$ and $\sqsubseteq$ by simply adding rule BI to the definition of the relation $\sqsubseteq$. The relation $\preceq$ may only be invoked at the top-level, and hence a usage of $\preceq$ may only be composed with $\sqsubseteq$. It would be unsound to use rule SUBS to derive type $\boxed{\tau \to \tau} \to \texttt{Int}$ from the type $\boxed{\forall a\,.\,a \to a} \to \texttt{Int}$.

5.2.3 Properties

The FPH system is type safe with respect to the semantics of the implicitly typed System F. One can observe that whenever $\sigma'_1 \preceq\!\sqsubseteq \sigma'_2$, it is the case that $\vdash^{\mathsf{F}} \lfloor \sigma'_1 \rfloor \leq \lfloor \sigma'_2 \rfloor$, where $\vdash^{\mathsf{F}}$ is the System

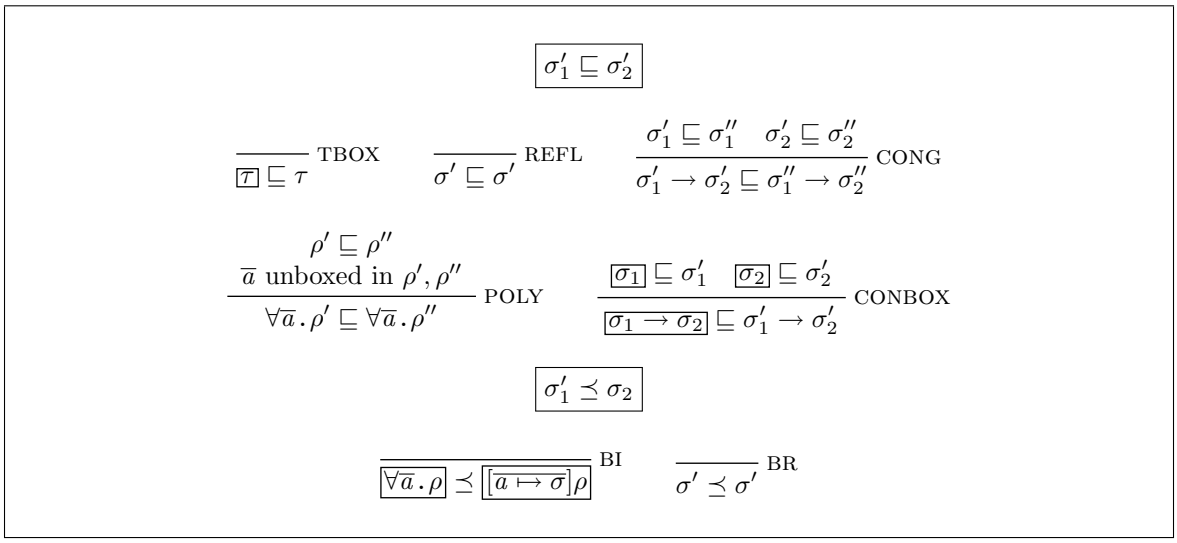

Figure 5.3: Protected unboxing and boxy instantiation relation

F type instance relation (Chapter 2). The (implicitly typed) System F typing relation is given in Figure 5.4. The figure includes `let`-bound definitions. We additionally define a function $(\cdot)^\flat$ on terms of FPH, that removes any annotations from terms. Its definition follows:

$$
\begin{array}{lcl}
x^\flat & = & x \\
(e :: \sigma)^\flat & = & e^\flat \\
(\lambda x \,.\, e)^\flat & = & \lambda x \,.\, e^\flat \\
(e_1 \; e_2)^\flat & = & e_1^\flat \; e_2^\flat \\
(\texttt{let } x = u \texttt{ in } e)^\flat & = & \texttt{let } x = u^\flat \texttt{ in } e^\flat
\end{array}
$$

The following lemma states formally the soundness of our typing relation with respect to System F.

Lemma 5.2.5. *If* $\Gamma \vdash e : \sigma'$ *then* $\Gamma \vdash^{\mathsf{F}} e^\flat : \lfloor \sigma' \rfloor$.

Proof. Straightforward induction. □

What this lemma shows is that if one ignores boxes and annotations on terms, a derivation in FPH is a derivation in implicitly typed System F, without further modification on the type or the expression. In contrast, the higher-rank type inference in Chapter 3 has to modify the actual terms by placing

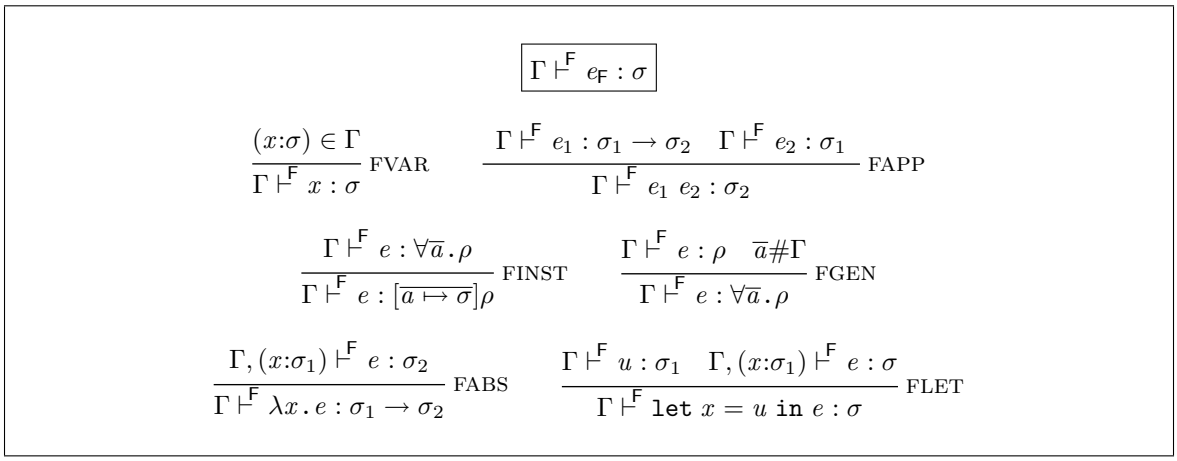

Figure 5.4: Implicitly typed System F (with local definitions)

coercion functions witnessing the Odersky-Läufer or deep-skolemization subsumption relations. In the case for FPH, no such subsumptions are used, and hence no need for term-level coercions exists.

Moreover, FPH is an extension of the Damas-Milner type system. The idea of the following lemma is that instantiation to τ' types always subsumes instantiation to τ types.

Lemma 5.2.6 (Extension of Damas-Milner)**.** *Assume that* Γ *contains types with only top-level quantifiers and* e *is annotation-free. Then* $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ *implies* $\Gamma \vdash e : \sigma$*.*

Proof. Straightforward induction. □

The converse direction is less interesting but we believe true: unannotated programs in environment that use only Damas-Milner types are typeable in Damas-Milner if they are typeable in FPH. Showing this result has to be done through the algorithmic implementation: in essence an unannotated program in a Damas-Milner environment gives rise to the same form of constraints (equalities between monomorphic types) as it would give in Hindley-Damas-Milner type inference. We do not proceed in formally proving this property.

5.2.4 Higher-rank types and System F

As we remarked in the discussion of rule ABS in Section 5.2.1, the system described so far deliberately does not support λ-abstractions with higher-rank types, and hence cannot yet express all of System

F. For example:

Example 5.2.7.

```
f   :: forall a. a -> [a] -> Int

foo :: [Int -> forall b.b->b]


bog = f (\x y -> y) foo
```

Here `bog` requires the λ-abstraction `\x y -> y` to be typed with type `Int` $\rightarrow \forall b\,.\,b \rightarrow b$, but no such type can be inferred for the λ-abstraction, as it is not of the form $\tau \rightarrow \rho$. We may resolve this issue by adding a new syntactic form, the annotated λ-abstraction, thus $(\lambda x\,.\,e :::\sigma_1 \rightarrow \sigma_2)$. This construct provides an annotation for both argument (σ_1, instead of a monotype τ) and result (σ_2 instead of ρ). Its typing rule is simple:

$$\frac{\Gamma, (x{:}\sigma_1) \vdash e : \sigma_2' \quad \lfloor \sigma_2' \rfloor = \sigma_2}{\Gamma \vdash (\lambda x\,.\,e :::\sigma_1 \rightarrow \sigma_2) : \sigma_1 \rightarrow \sigma_2}\ \text{ABS-ANN}$$

With this extra construct we can translate any implicitly-typed System F term into a well-typed term in FPH, using the translation of Figure 5.5. This type-directed translation of implicitly typed System F is specified as a judgement $\Gamma \vdash^{\mathsf{F}} e_{\mathsf{F}} : \sigma \rightsquigarrow e$ where e is a term that type checks in FPH. The translation is nothing more than the standard System F typing relation, instrumented with the term that is the result of the translation. Notice that the translation requires annotations *only* on λ-abstractions that have rich types (of course, it would be fine to annotate *every* λ-abstraction, but the translation we give generates terms with fewer annotations).

A subtle point is that the translation may generate *open* type annotations. For example, consider the implicitly typed System F term below:

$$\vdash \lambda x\,.\,e : \forall a\,.(\forall b\,.\,b \rightarrow a) \rightarrow a$$

Translating this term using Figure 5.5 gives $(\lambda x\,.\,e :::(\forall b\,.\,b \rightarrow a) \rightarrow a)$. Note that the type annotation mentions a which is nowhere bound. For the rest of this section we assume that FPH already accommodates such open annotations on abstractions.

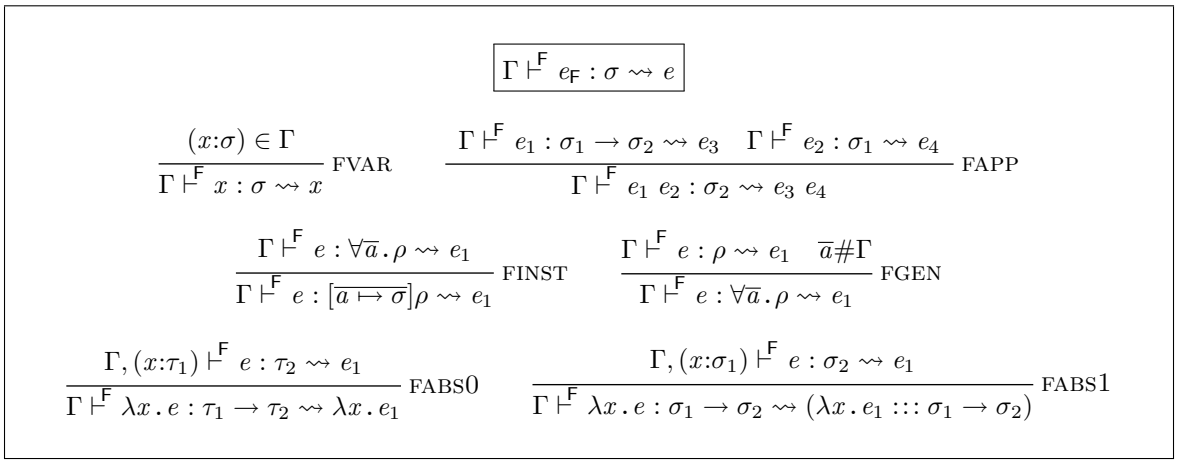

Figure 5.5: Type-directed translation of System F

The following theorem captures the essence of the translation.

Theorem 5.2.8. *If* $\Gamma \vdash^{\mathsf{F}} e : \sigma \rightsquigarrow e_1$ *then* $\Gamma \vdash e_1 : \sigma'$ *for some* σ' *with* $\lfloor \sigma' \rfloor = \sigma$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash^{\mathsf{F}} e : \sigma \rightsquigarrow e_1$. The case for FVAR is trivial. For FAPP we have that $\Gamma \vdash^{\mathsf{F}} e_1\ e_2 : \sigma_2 \rightsquigarrow e_3\ e_4$ given that $\Gamma \vdash^{\mathsf{F}} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_3$ and $\Gamma \vdash^{\mathsf{F}} e_2 : \sigma_1 \rightsquigarrow e_4$. By induction we have that $\Gamma \vdash e_1 : \sigma_0'$ with $\lfloor \sigma_1' \rfloor = \sigma_1 \to \sigma_2$. This implies that $\sigma_0' \sqsubseteq \sigma_1' \to \sigma_2'$ with $\lfloor \sigma_1' \rfloor = \sigma_1$ and $\lfloor \sigma_2' \rfloor = \sigma_2$. Hence, by rule SUBS and reflexivity of $\preceq$ it is $\Gamma \vdash e_1 : \sigma_1' \to \sigma_2'$. Also by induction we get $\Gamma \vdash e_2 : \sigma_1''$ with $\lfloor \sigma_1'' \rfloor = \sigma_1$. By applying rule APP we get $\Gamma \vdash e_1\ e_2 : \sigma_2'$ for which we know that $\lfloor \sigma_2' \rfloor = \sigma_2$, as required. For rule FINST we have that $\Gamma \vdash^{\mathsf{F}} e : [\overline{a \mapsto \sigma}]\rho \rightsquigarrow e_1$ where $\Gamma \vdash^{\mathsf{F}} e : \forall \overline{a}\,.\,\rho \rightsquigarrow e_1$. By induction $\Gamma \vdash e : \sigma'$ with $\lfloor \sigma' \rfloor = \forall \overline{a}\,.\,\rho$. Hence, by a sequence of INST and perhaps SUBS with BI we get the result. For rule FGEN the result follows by induction hypothesis and rule GEN. For rule FABS0 it is $\Gamma \vdash^{\mathsf{F}} \lambda x\,.\,e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x\,.\,e_1$, given that $\Gamma, (x{:}\tau_1) \vdash^{\mathsf{F}} e : \tau_2 \rightsquigarrow e_1$. By induction $\Gamma, (x{:}\tau_1) \vdash e_1 : \sigma_2'$ with $\lfloor \sigma_2' \rfloor = \tau_2$. Because τ_2 is monomorphic it must be that $\sigma_2' \preceq \sqsubseteq \tau_2$. Consequently, by rule SUBS, $\Gamma, (x{:}\tau_1) \vdash e_1 : \tau_2$, and by rule ABS $\Gamma \vdash \lambda x\,.\,e_1 : \tau_1 \to \tau_2$ as required. The case for rule FABS1 follows from induction hypothesis and rule ABS-ANN. □

Additionally, notice that our annotation form $e{::}\sigma$ is useless to this translation. In essence, in the presence of annotated abstractions, the form $e{::}\sigma$ has to be used only at `let`-bound expressions, where we may need to unbox the contents of boxes appearing in the type of the `let`-bound expression.

Since the implicit System F we address here contains no `let` expressions, there is no need for such annotation forms.

In practice, however, we do not recommend adding annotated λ-abstractions as a clunky new syntactic construct. Instead, with a bidirectional typing system we can get the same benefits (and more besides) from ordinary type annotations $e :: \sigma$, as we sketch in Section 7.1.

Having discussed expressiveness, for the rest of this document we assume that type annotations are closed, and we do not further address the implementation of rule ANN-ABS, assuming that the only form of annotations is the one that rule ANN offers. In fact, there are several ways to implement open type annotations; either using scoped type variables (as is currently the case in GHC), or by using partial type annotations (often referred to as existential type annotations), but this issue is orthogonal to the concerns raised in FPH.

5.2.5 Predictability and robustness

A key feature of FPH is that it is simple for the programmer to figure out when a type annotation is required. We gave intuitions in Section 5.1, but now we are in a position to give some specific results. The translation of System F to FPH of Section 5.2.4 shows that one needs only annotate `let`-bindings or λ-abstractions that must be typed with rich types. This is a result of combining Theorem 5.2.8 and Lemma 5.2.4.

For example, consider a System F applicative expression: one consisting only of variables, constants, and applications. If such an expression is typeable in System F then it is typeable in FPH without annotations. Hence typeability does not break by applications of polymorphic combinators (though annotations may be needed if we are to bind such an application).

Theorem 5.2.9. *If e is an applicative expression and $\Gamma \vdash^{\mathsf{F}} e : \sigma$ then $\Gamma \vdash e : \sigma'$ for some σ' with $\lfloor \sigma' \rfloor = \sigma$.*

Proof. This result follows from Theorem 5.2.8 by observing that whenever e is applicative and $\Gamma \vdash^{\mathsf{F}} e : \sigma \rightsquigarrow e_1$ then $e_1 = e$, that is, the translation never adds any annotations to applicative terms. □

Here's an example of the application of polymorphic combinators. Assume that $\Gamma \vdash e_1\ e_2 : \sigma_2'$ using rule APP. We therefore have:

$$\frac{\Gamma \vdash e_1 : \sigma_1' \to \sigma_2' \quad \Gamma \vdash e_2 : \sigma_3' \quad \lfloor \sigma_3' \rfloor = \lfloor \sigma_1' \rfloor}{\Gamma \vdash e_1\ e_2 : \sigma_2'}\ \text{APP}$$

Let us now consider the expressions $\texttt{app}\ e_1\ e_2$ and $\texttt{revapp}\ e_2\ e_1$, where $\texttt{app} : \forall ab\,.\,(a \to b) \to a \to b$ and $\texttt{revapp} : \forall ab\,.\,a \to (a \to b) \to b$. Then we may construct the following derivation:

$$\dfrac{\dfrac{\begin{array}{l}\Gamma \vdash \texttt{app} : (\boxed{\lfloor \sigma_1' \rfloor} \to \boxed{\lfloor \sigma_2' \rfloor}) \to \boxed{\lfloor \sigma_1' \rfloor} \to \boxed{\lfloor \sigma_2' \rfloor} \\ \Gamma \vdash e_1 : \sigma_1' \to \sigma_2' \\ \lfloor \sigma_1' \to \sigma_2' \rfloor = \lfloor \boxed{\lfloor \sigma_1' \rfloor} \to \boxed{\lfloor \sigma_2' \rfloor} \rfloor\end{array}}{\Gamma \vdash \texttt{app}\ e_1 : \boxed{\lfloor \sigma_1' \rfloor} \to \boxed{\lfloor \sigma_2' \rfloor}}\ \text{APP} \qquad \Gamma \vdash e_2 : \sigma_3' \quad \lfloor \sigma_3' \rfloor = \lfloor \boxed{\lfloor \sigma_1' \rfloor} \rfloor}{\Gamma \vdash \texttt{app}\ e_1\ e_2 : \boxed{\lfloor \sigma_2' \rfloor}}\ \text{APP}$$

A similar proof tree can be derived for the application of the `revapp` combinator. What this means is that there are cases where an annotation is needed around $\texttt{app}\ e_1\ e_2$ in order to unbox the polymorphism of σ_2'—for instance to `let`-bind the application. On the other hand, if σ_2' is monomorphic, or of the form $\forall \overline{a}\,.\,\tau$, no annotations are required to bind the application.

Additionally, a `let`-binding can always be inlined at its occurrence sites. More precisely if $\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \sigma'$, then $\Gamma \vdash [x \mapsto u]e : \sigma'$. This follows from the next lemma:

Lemma 5.2.10. *If* $\Gamma \vdash u : \sigma$ *and* $\Gamma, (x{:}\sigma) \vdash e : \sigma'$ *then* $\Gamma \vdash [x \mapsto u]e : \sigma'$.

Proof. Straightforward induction. □

The converse cannot be true in general (although it is true for ML and ML^F) because of the limited expressive power of System F types. Let $\sigma_1 = (\forall b\,.\,b \to b) \to (\forall b\,.\,b \to b)$, $\sigma_2 = \forall b\,.\,(b \to b) \to b \to b$, $f_1 : \sigma_1 \to \texttt{Int}$, and $f_2 : \sigma_2 \to \texttt{Int}$. One can imagine a program of the form:

$$\ldots (f_1\ (\texttt{choose id})) \ldots (f_2\ (\texttt{choose id})) \ldots$$

which may be typeable, but it cannot be the case that: $\texttt{let}\ x = \texttt{choose id}\ \texttt{in} \ldots (f_1\ x) \ldots (f_2\ x) \ldots$

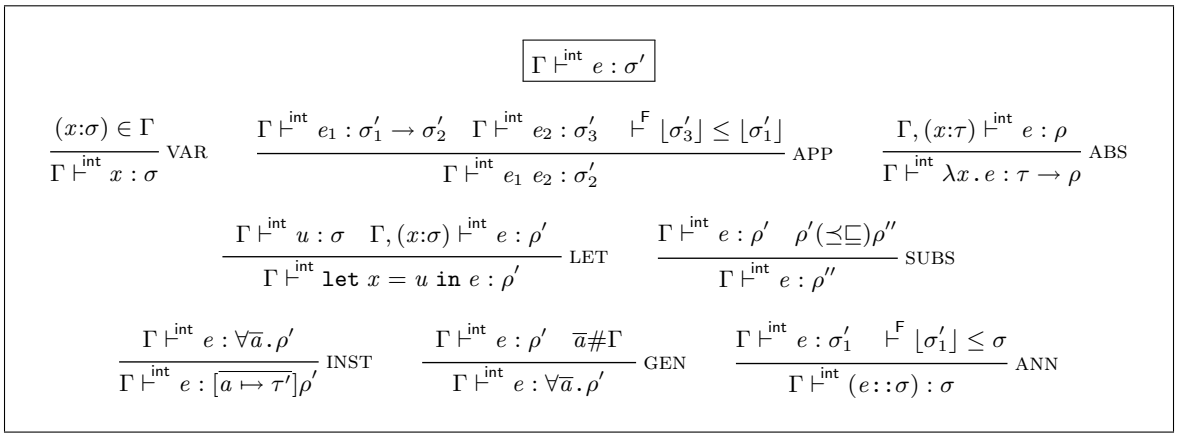

Figure 5.6: The type system with System F instance

is typeable, as x can be bound with only one of the two incomparable types (in fact only with $\forall b\,.\,(b \to b) \to b \to b$).

However, notice that if an expression is typed with a type that can be converted along $\preceq\sqsubseteq$ to a box-free type at each of its occurrences in a context, it may be `let`-bound out of the context. For example, since λ-abstractions are typed with box-free types, if $\mathcal{C}[\lambda x\,.\,e]$ is typeable, where $\mathcal{C}$ is a multi-hole context, then it is always the case that $\mathtt{let}\ f = (\lambda x\,.\,e)\ \mathtt{in}\ \mathcal{C}[f]$ is typeable. This is a corollary of type inference soundness and completeness, to be given in Chapter 6.

5.2.6 An equivalent declarative specification

It will be convenient to introduce at this point a slight variation of the basic type system of Figure 5.2 where we have pushed some instantiations and generalizations into the applications and type annotation nodes. The modified system is given in Figure 5.6, with the relation $\Gamma \vdash^{\mathsf{int}} e : \sigma'$ ($\vdash^{\mathsf{int}}$ for *intermediate*).

The differences with respect to the type system of Figure 5.2 are in rules APP and ANN, which instead of equality appeal to System F instance. However, the two type systems type exactly the same programs.

Theorem 5.2.11. *If* $\Gamma \vdash e : \sigma'$ *then* $\Gamma \vdash^{\mathsf{int}} e : \sigma'$.

Proof. Straightforward induction, observing that $\lfloor \sigma_1' \rfloor = \lfloor \sigma_2' \rfloor$ implies $\vdash^{\mathsf{F}} \lfloor \sigma_1' \rfloor \leq \lfloor \sigma_2' \rfloor$. □

Theorem 5.2.12. *If* $\Gamma \vdash^{\mathsf{int}} e : \sigma'$ *then* $\Gamma \vdash e : \sigma'$.

Proof. By induction on the derivation of $\Gamma \vdash^{\mathsf{int}} e : \sigma'$. The only interesting case is really the application case (the annotation case is similar), where we have that $\Gamma \vdash^{\mathsf{int}} e_1\ e_2 : \sigma_2'$ where $\Gamma \vdash^{\mathsf{int}} e_1 : \sigma_1' \to \sigma_2'$ and $\Gamma \vdash^{\mathsf{int}} e_2 : \sigma_3'$ with $\vdash^{\mathsf{F}} \lfloor \sigma_3' \rfloor \leq \lfloor \sigma_1' \rfloor$. By induction hypothesis it is $\Gamma \vdash e_1 : \sigma_1' \to \sigma_2'$ and $\Gamma \vdash^{\mathsf{int}} e_2 : \sigma_3'$. It must also be the case that $\lfloor \sigma_3' \rfloor = \forall \overline{a} \,.\, \rho$ and $\lfloor \sigma_1' \rfloor = \forall \overline{b} \,.\, [\overline{a \mapsto \sigma}]\rho$, where $\overline{b} \# ftv(\forall \overline{a} \,.\, \rho)$ and without loss of generality assume also that $\overline{b} \# \Gamma$. Let us consider two cases for σ_3'.

- $\sigma_3' = \forall \overline{a}_1 \,.\, \boxed{\forall \overline{a}_2 \,.\, \rho}$ such that $\overline{a} = \overline{a}_1 \overline{a}_2$, and assume without loss of generality that $\overline{a} \# \Gamma$. Then by rule INST we get $\Gamma \vdash e_2 : \boxed{\forall \overline{a}_2 \,.\, \rho}$ and by rule SUBS and rule BI we get $\Gamma \vdash e_2 : \boxed{\rho}$. By rule GEN we get $\Gamma \vdash e_2 : \forall \overline{a} \,.\, \boxed{\rho}$ and by INST we get $\Gamma \vdash e_2 : \boxed{[\overline{a \mapsto \sigma}]\rho}$. By rule GEN we get finally that $\Gamma \vdash e_2 : \forall \overline{b} \,.\, \boxed{[\overline{a \mapsto \sigma}]\rho}$, and rule APP is applicable to finish the case.
- $\sigma_3' = \forall \overline{a} \,.\, \rho'$ such that $\rho = \lfloor \rho' \rfloor$. Then by rule INST it is $\Gamma \vdash e_2 : [\overline{a \mapsto \boxed{\sigma}}]\rho'$ and by rule GEN we have $\Gamma \vdash e_2 : \forall \overline{b} \,.\, [\overline{a \mapsto \boxed{\sigma}}]\rho'$. Then, rule APP is applicable and finishes the case.

□

In fact, as the previous theorem suggests, System F instance can be simulated by subsumptions, instantiations and generalizations in our type system.

Lemma 5.2.13. *If* $\Gamma \vdash e : \sigma_1'$ *and* $\vdash^{\mathsf{F}} \lfloor \sigma_1' \rfloor \leq \lfloor \sigma_2' \rfloor$ *then* $\Gamma \vdash e : \sigma_3'$ *with* $\lfloor \sigma_3' \rfloor = \lfloor \sigma_2' \rfloor$.

Proof. Straightforward. □

Hence, the two declarative systems are equivalent. The type system of Figure 5.6 is more convenient to work with in the rest of this document, and hence we simply write $\vdash$ instead of $\vdash^{\mathsf{int}}$ below.

5.3 Syntax-directed specification

We now show how FPH may be implemented. The first step in establishing an algorithmic implementation is to specify a syntax-directed version of the type system of Figure 5.6, where uses of the non-syntax-directed rules (SUBS, INST, and GEN) have been pushed to appropriate nodes inside the

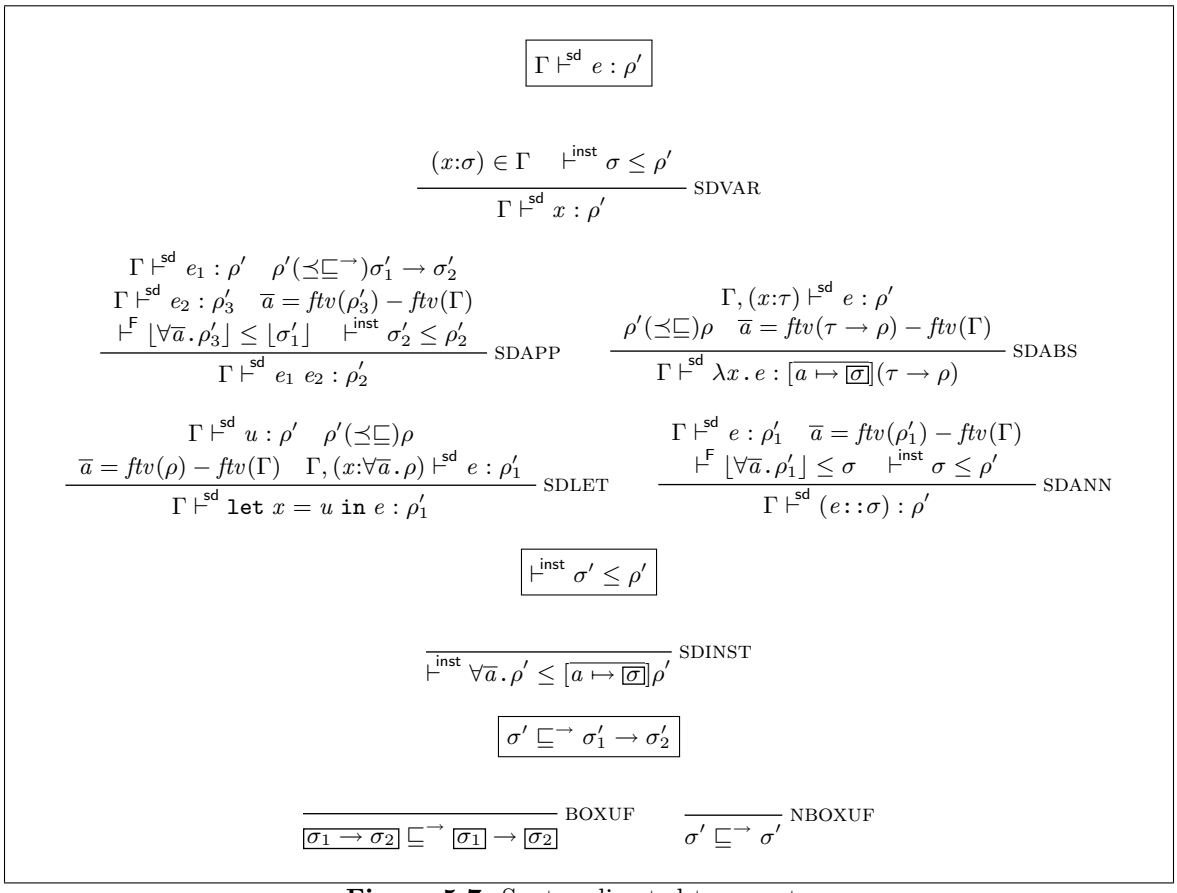

Figure 5.7: Syntax-directed type system

syntax-tree. Subsequently we may proceed with a low-level implementation of the syntax-directed system (Chapter 6). Our syntax-directed presentation appears in Figure 5.7.

Rule SDVAR instantiates the type of a variable bound in the environment, using the auxiliary judgement, $\vdash^{\mathsf{inst}} \sigma' \leq \rho'$. The latter instantiates the top-level quantifiers of σ' to yield a ρ' type. However, instead of instantiating with arbitrary τ' types, we instantiate only with types of the form $\boxed{\sigma}$, which is closer to the actual algorithm as boxes correspond to fresh constrained variables.

Rule SDAPP deals with applications. It infers a type ρ' for the function, and uses $\preceq$ (Figure 5.3) and $\sqsubseteq^{\rightarrow}$ (a subset of $\sqsubseteq$) to expose an arrow constructor. The latter step is called *arrow unification*. Then SDAPP infers a ρ_3' type for the argument of the application, generalizes over free variables that do not appear in the environment, and checks that the result is more polymorphic (along the System F type instance) than the required type. Finally SDAPP instantiates the return type.

Rule SDABS uses a τ type for the argument of the λ-abstraction, and then forces the returned type ρ' for the body to be unboxed to a ρ-type using $\rho' \preceq\sqsubseteq \rho$. Finally, we consider all the free variables of the abstraction type that do not appear in the environment, and substitute them with arbitrary boxes. The returned type for the λ-abstraction is $[\overline{a \mapsto \boxed{\sigma}}](\tau \to \rho)$.

This last step, of generalization and instantiation, is perhaps puzzling. After all, rule ABS (in the declarative specification of Figure 5.2) seems to only force λ-abstractions to have box-free types. Here is an example that shows why generalization and instantiation is needed:

Example 5.3.1 (Impredicative instantiations in λ-abstractions)**.** The following derivation holds: $\Gamma \vdash (\lambda x\,.\,x)\ \texttt{ids} : \boxed{[\forall a\,.\,a \to a]}$.

To construct a derivation for Example 5.3.1 observe that we can instantiate $\lambda x\,.\,x$ with a polymorphic argument type, as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma, (x{:}a) \vdash x : a}{\Gamma \vdash \lambda x\,.\,x : a \to a}\ \text{ABS}}{\Gamma \vdash \lambda x\,.\,x : \forall a\,.\,a \to a}\ \text{GEN}}{\Gamma \vdash \lambda x\,.\,x : \boxed{[\forall a\,.\,a \to a]} \to \boxed{[\forall a\,.\,a \to a]}}\ \text{INST}$$

The use of GEN and INST are essential to make the term applicable to $\texttt{ids} : [\forall a\,.\,a \to a]$. The generalization and instantiation in SDABS ensure that GEN and INST are performed at each λ-abstraction, much as SDLET ensures that GEN is performed at each `let`-binding.

Rule SDLET is straightforward; after inferring a type for u which may contain boxes, we check that the boxes can be removed by $\preceq\sqsubseteq$ to get a ρ-type, which can subsequently be generalized and pushed into the environment.

Finally rule SDANN infers a type ρ_1' for the expression e, and generalizes over its free variables that do not appear in the environment. It subsequently checks that the generalized type is more polymorphic than the annotation type. As the final step, the annotation type is instantiated.

We can now establish the soundness of the syntax-directed system with respect to the declarative one (of Figure 5.6):

Theorem 5.3.2 (Soundness of $\vdash^{\mathsf{sd}}$)**.** *If* $\Gamma \vdash^{\mathsf{sd}} e : \rho'$ *then* $\Gamma \vdash e : \rho'$.

Proof. Straightforward induction. □

To prove the converse we need some additional machinery. First, recall the predicative restriction of the $\vdash^{\mathsf{F}}$ relation, $\vdash^{\mathsf{DM}}$, from Chapter 2, where ρ-types may contain nested polymorphism. We write $\vdash^{\mathsf{DM}} \Gamma_2 \leq \Gamma_1$ if for every $(x{:}\sigma_1) \in \Gamma_1$, there exists a σ_2 such that $(x{:}\sigma_2) \in \Gamma_2$, and $\vdash^{\mathsf{DM}} \sigma_2 \leq \sigma_1$.

We have the following lemmas about the System F and the Damas-Milner instance relations.

Lemma 5.3.3 (Transitivity of $\vdash^{\mathsf{DM}}$ and $\vdash^{\mathsf{F}}$)**.** *If* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$ *and* $\vdash^{\mathsf{DM}} \sigma_2 \leq \sigma_3$ *then* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_3$*. If* $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_2$ *and* $\vdash^{\mathsf{F}} \sigma_2 \leq \sigma_3$ *then* $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_3$*.*

Proof. Easy inductions. □

Lemma 5.3.4 ($\vdash^{\mathsf{F}}$ substitution stability)**.** *If* $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_2$ *then* $\vdash^{\mathsf{F}} \lfloor \varphi\sigma_1 \rfloor \leq \lfloor \varphi\sigma_2 \rfloor$*.*

Proof. Easy induction. □

Moreover, $\sqsubseteq$ is transitive.

The next lemma is particularly useful for establishing type inference completeness, and in essence is a commutation property between substitution and generalization.

Lemma 5.3.5 (Gen-subs commutation for $\vdash^{\mathsf{DM}}$)**.** *Assume that* φ *is box-free. Let* $\overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma)$ *and* $\overline{b} = \mathit{ftv}(\varphi\rho) - \mathit{ftv}(\varphi\Gamma)$*. It follows that* $\vdash^{\mathsf{DM}} \varphi(\forall\overline{a}\,.\,\rho) \leq \forall\overline{b}\,.\,\varphi\rho$*.*

Proof. Easy unfolding of the definitions. □

Lemma 5.3.6. *If* $\vdash^{\mathsf{DM}} \sigma_1 \leq \sigma_2$ *then* $\mathit{ftv}(\sigma_1) \subseteq \mathit{ftv}(\sigma_2)$*.*

Proof. Easy check. □

Lemma 5.3.7. *If* $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_2$ *then* $\mathit{ftv}(\sigma_1) \subseteq \mathit{ftv}(\sigma_2)$*.*

Proof. Easy check. □

Finally, we show a sequence of substitution properties.

Lemma 5.3.8 ($\vdash^{\mathsf{inst}}$ substitution stability)**.** *If* $\vdash^{\mathsf{inst}} \sigma' \leq \rho'$ *then* $\vdash^{\mathsf{inst}} \varphi\sigma' \leq \varphi\rho'$*.*

Proof. Assume $\sigma' = \forall \overline{a} \,.\, \rho_1'$ such that $\overline{a} \# ftv(\varphi)$. Then we know that $\rho' = \overline{[a \mapsto \boxed{\sigma}]}\rho_1'$ and hence $\varphi\rho' = \varphi\overline{[a \mapsto \boxed{\sigma}]}\rho_1' = \overline{[a \mapsto \boxed{\lfloor \varphi\sigma \rfloor}]}\varphi\rho_1'$ as required. □

Lemma 5.3.9 ($\sqsubseteq$ substitution stability)**.** *If $\sigma_1' \sqsubseteq \sigma_2'$ then $\varphi\sigma_1' \sqsubseteq \varphi\sigma_2'$.*

Proof. Easy induction. □

Corollary 5.3.10 ($\preceq\sqsubseteq$ substitution stability)**.** *If $\sigma_1' \preceq\sqsubseteq \sigma_2'$ then $\varphi\sigma_1' \preceq\sqsubseteq \varphi\sigma_2'$.*

Proof. Follows by Lemma 5.3.4 and Lemma 5.3.9. □

Lemma 5.3.11 (Bijection substitution)**.** *If $\Gamma \vdash^{\mathsf{sd}} e : \rho'$ and φ is a variable bijection then $\varphi\Gamma \vdash^{\mathsf{sd}} e : \varphi\rho'$ and the new derivation has the same height.*

Proof. Easy induction. □

Theorem 5.3.12 (Substitution)**.** *If $\Gamma \vdash^{\mathsf{sd}} e : \rho'$ and $dom(\varphi) \# ftv(\Gamma)$ then $\Gamma \vdash^{\mathsf{sd}} e : \varphi\rho'$.*

Proof. By induction on the height of the derivation of $\Gamma \vdash^{\mathsf{sd}} e : \rho'$. We have the following cases to consider:

- Case SDVAR. We have that $\Gamma \vdash^{\mathsf{sd}} x : \rho'$ given that $(x{:}\sigma) \in \Gamma$ and $\vdash^{\mathsf{inst}} \sigma \leq \rho'$. By inverting $\vdash^{\mathsf{inst}}$ we know that $\sigma = \forall \overline{a} \,.\, \rho$ and $\rho' = \overline{[a \mapsto \boxed{\sigma}]}\rho$. But $ftv(\forall \overline{a} \,.\, \rho) \# dom(\varphi)$ and assuming without loss of generality that $\overline{a} \# ftv(\varphi)$ we only need show that: $\Gamma \vdash^{\mathsf{sd}} x : \varphi\overline{[a \mapsto \boxed{\sigma}]}\rho$ But then:

  $$\varphi\overline{[a \mapsto \boxed{\sigma}]}\rho = \overline{[a \mapsto \boxed{\lfloor \varphi\sigma \rfloor}]}\varphi\rho = \overline{[a \mapsto \boxed{\lfloor \varphi\sigma \rfloor}]}\rho$$

  To finish the case by rule SDVAR we only need show that $\vdash^{\mathsf{inst}} \forall \overline{a} \,.\, \rho \leq \overline{[a \mapsto \boxed{\lfloor \varphi\sigma \rfloor}]}\rho$, which is true.

- Case SDABS. In this case we have that: $\Gamma \vdash^{\mathsf{sd}} \lambda x \,.\, e : \overline{[a \mapsto \boxed{\sigma}]}(\tau \to \rho)$, given that $\Gamma, (x{:}\tau) \vdash^{\mathsf{sd}} e : \rho'$, $\rho' \preceq\sqsubseteq \rho$, and $\overline{a} = ftv(\tau \to \rho) - ftv(\Gamma)$. Let us assume using Lemma 5.3.11 that $\overline{a} \# ftv(\varphi)$. Then we need to show that $\Gamma \vdash^{\mathsf{sd}} \lambda x \,.\, e : \varphi\overline{[a \mapsto \boxed{\sigma}]}(\tau \to \rho)$. But $\varphi\overline{[a \mapsto \boxed{\sigma}]}(\tau \to \rho) = \overline{[a \mapsto \boxed{\lfloor \varphi\sigma \rfloor}]}\varphi(\tau \to \rho)$. But notice that $ftv(\tau \to \rho) = \overline{a} \cup (ftv(\tau \to \rho) \cap ftv(\Gamma))$. Hence

$\varphi(\tau \to \rho) = \tau \to \rho$. Hence we only need to show that $\Gamma \vdash^{\mathsf{sd}} \lambda x\,.\,e : [\overline{a \mapsto \boxed{\lfloor \varphi\sigma \rfloor}}](\tau \to \rho)$, which follows by applying directly rule SDABS.

- Case SDAPP. We have in this case that: $\Gamma \vdash^{\mathsf{sd}} e_1\ e_2 : \rho_2'$, given that $\Gamma \vdash^{\mathsf{sd}} e_1 : \rho'$, $\rho' \preceq \sqsubseteq^{\to} \sigma_1' \to \sigma_2'$, $\Gamma \vdash^{\mathsf{sd}} e_2 : \rho_3'$, $\overline{a} = ftv(\rho_3') - ftv(\Gamma)$, $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho_3' \rfloor \leq \lfloor \sigma_1' \rfloor$, and finally $\vdash^{\mathsf{inst}} \sigma_2' \leq \rho_2'$. By induction hypothesis we have that: $\Gamma \vdash^{\mathsf{sd}} e_1 : \varphi\rho'$, and we can observe that $\varphi\rho' \preceq \sqsubseteq^{\to} \varphi\sigma_1' \to \varphi\sigma_2'$. Moreover, let us assume that $\overline{a} \# ftv(\varphi)$ by appealing to Lemma 5.3.11. Hence $\varphi(\lfloor \forall \overline{a}\,.\,\rho_3' \rfloor) = \lfloor \forall \overline{a}\,.\,\rho_3' \rfloor$. Then, we know that: $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho_3' \rfloor \leq \lfloor \sigma_1' \rfloor$ and hence: $\vdash^{\mathsf{F}} \lfloor \varphi(\forall \overline{a}\,.\,\rho_3') \rfloor \leq \lfloor \varphi(\sigma_1') \rfloor$ by appealing to the $\vdash^{\mathsf{F}}$ substitution property. The case is finished with rule SDAPP if we show that $\vdash^{\mathsf{inst}} \varphi\sigma_2' \leq \varphi\rho_2'$, which holds by Lemma 5.3.8.
- Case SDLET. In this case we have that $\Gamma \vdash^{\mathsf{sd}} \mathtt{let}\ x = u\ \mathtt{in}\ e : \rho_1'$ given that $\Gamma \vdash^{\mathsf{sd}} u : \rho'$, $\rho' \preceq \sqsubseteq \rho$, $\overline{a} = ftv(\rho) - ftv(\Gamma)$. and $\Gamma, (x{:}\forall \overline{a}\,.\,\rho) \vdash^{\mathsf{sd}} e : \rho_1'$. But we know that $\Gamma \vdash^{\mathsf{sd}} u : \rho'$. We can apply the induction hypothesis then to get $\Gamma, (x{:}\forall \overline{a}\,.\,\rho) \vdash^{\mathsf{sd}} e : \varphi\rho_1'$, since $ftv(\forall \overline{a}\,.\,\rho) \subseteq ftv(\Gamma)$, and hence also $dom(\varphi) \# ftv(\forall \overline{a}\,.\,\rho)$, and the case is finished.
- Case SDANN. In this case we have that $\Gamma \vdash^{\mathsf{sd}} (e :: \sigma) : \rho'$ given that $\Gamma \vdash^{\mathsf{sd}} e : \rho_1'$, $\overline{a} = ftv(\rho_1') - ftv(\Gamma)$, and $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho_1' \rfloor \leq \sigma$. Let us assume by appealing to Lemma 5.3.11 that $\overline{a} \# ftv(\varphi)$. Then we know that $\varphi \lfloor \forall \overline{a}\,.\,\rho_1' \rfloor = \lfloor \forall \overline{a}\,.\,\rho_1' \rfloor$. By stability of System F instance under substitution it is $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho_1' \rfloor \leq \varphi\sigma$, and applying Lemma 5.3.8 and rule SDANN finishes the case.

□

The following is an inversion property for $\sqsubseteq$.

Lemma 5.3.13 (Arrow unification)**.** *If $\sigma' \sqsubseteq \sigma_1' \to \sigma_2'$ then*

- *$\sigma' = \boxed{\sigma_1 \to \sigma_2}$ with $\boxed{\sigma_1} \sqsubseteq \sigma_1'$ and $\boxed{\sigma_2} \sqsubseteq \sigma_2'$, or*
- *$\sigma' = \sigma_1'' \to \sigma_2''$ with $\sigma_1'' \sqsubseteq \sigma_1'$ and $\sigma_2'' \sqsubseteq \sigma_2'$.*

Proof. Easy induction on $\sqsubseteq$. □

The next two lemmas establish transitivity of $\preceq\sqsubseteq$.

Lemma 5.3.14 ($\sqsubseteq$ transitivity)**.** *If $\sigma_1' \sqsubseteq \sigma_2'$ and $\sigma_2' \sqsubseteq \sigma_3'$ then $\sigma_1' \sqsubseteq \sigma_3'$.*

Proof. Easy induction on the sums of the heights of the two derivations. □

Lemma 5.3.15 (Transitivity of $\preceq\sqsubseteq$)**.** *If* $\sigma_1' \preceq\sqsubseteq \sigma_2'$ *and* $\sigma_2' \preceq\sqsubseteq \sigma_3'$ *then* $\sigma_1' \preceq\sqsubseteq \sigma_3'$.

Proof. Assume that $\sigma_1' \preceq\sqsubseteq \sigma_2'$ and $\sigma_2' \preceq\sqsubseteq \sigma_3'$. We consider cases on σ_2'.

- Assume that $\sigma_2' = \boxed{\sigma}$. By inversion on $\sqsubseteq$, since $\sigma_1' \preceq\sqsubseteq \boxed{\sigma}$, it must be that $\sigma_1' \preceq \boxed{\sigma}$, and hence we have two cases by inversion on $\preceq$:
  - Case $\sigma_1' = \boxed{\sigma}$ (using rule BR). Then, we know by assumption that $\boxed{\sigma} = \sigma_2' \preceq\sqsubseteq \sigma_3'$, and therefore the case is finished.
  - Case $\sigma_1' = \boxed{\forall \overline{a}\,.\,\rho}$ and $\boxed{\sigma} = \boxed{\overline{[a \mapsto \sigma_a]}\rho}$ (using rule BI). Moreover we know by assumptions that $\boxed{\overline{[a \mapsto \sigma_a]}\rho} \preceq\sqsubseteq \sigma_3'$. We consider cases on whether $\rho = a \in \overline{a}$ or not. If $\rho = a$ then assume that $\sigma_a = \forall \overline{b}\,.\,\rho_b$ in which case we know that $\boxed{\sigma_a} \preceq \boxed{\overline{[b \mapsto \sigma_b]}\rho_b} \sqsubseteq \sigma_3'$. But in that case, also $\sigma_1' = \boxed{\forall \overline{a}\,.\,\rho} = \boxed{\forall \overline{a}\,.\,a} \preceq \boxed{\overline{[b \mapsto \sigma_b]}\rho_b} \sqsubseteq \sigma_3'$. Consequently, $\sigma_1' \preceq\sqsubseteq \sigma_3'$ as required. If on the other hand $\rho \neq a$, this means that $\boxed{\overline{[a \mapsto \sigma_a]}\rho} \sqsubseteq \sigma_3'$. Hence, $\sigma_1' = \boxed{\forall \overline{a}\,.\,\rho} \preceq \boxed{\overline{[a \mapsto \sigma_a]}\rho} \sqsubseteq \sigma_3'$, that is $\sigma_1' \preceq\sqsubseteq \sigma_3'$.
- Assume that $\sigma_2' \neq \boxed{\sigma}$ for any σ. It follows by inversion on $\preceq$ that $\sigma_2' \sqsubseteq \sigma_3'$. Therefore we have $\sigma_1' \preceq\sqsubseteq \sigma_2' \sqsubseteq \sigma_3'$, and by transitivity of $\sqsubseteq$ we get $\sigma_1' \preceq\sqsubseteq \sigma_3'$.

□

With all the auxiliary lemmas in place, we can now state and prove the completeness of the syntax-directed specification with respect to the declarative one.

Theorem 5.3.16 (Completeness of syntax-directed system)**.** *Assume that* $\Gamma_1 \vdash e : \forall \overline{a}\,.\,\rho'$. *Then, for all* Γ_2 *with* $\vdash^{\mathsf{DM}} \Gamma_2 \leq \Gamma_1$ *and for all* $\overline{\sigma}$ *there exists a* ρ_0' *with* $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho_0'$ *and* $\rho_0' \preceq\sqsubseteq \overline{[a \mapsto \boxed{\sigma}]}\rho'$.

Proof. By induction on the derivation of $\Gamma_1 \vdash e : \forall \overline{a}\,.\,\rho'$. We consider the following cases:

- Case VAR. We have in this case that $(x{:}\sigma_1) \in \Gamma_1$ and $\Gamma_1 \vdash x : \sigma_1$. Then $(x{:}\sigma_2) \in \Gamma_2$ and moreover $\vdash^{\mathsf{DM}} \sigma_2 \leq \sigma_1$. Assume that $\sigma_2 = \forall \overline{a}\,.\,\rho$ and $\sigma_1 = \forall \overline{b}\,.\,\overline{[a \mapsto \tau]}\rho$ given that $\overline{b} \# \mathit{ftv}(\forall \overline{a}\,.\,\rho)$. Then, let us pick $\overline{\sigma}$. We need to find $\overline{\sigma}_a$ such that $\overline{[a \mapsto \boxed{\sigma_a}]}\rho \preceq\sqsubseteq \overline{[b \mapsto \boxed{\sigma}]}\overline{[a \mapsto \tau]}\rho$. For

this we can simply pick each σ_a to be the type $\lfloor[\overline{b \mapsto \boxed{\sigma}}]\tau\rfloor$, and it is actually $[\overline{a \mapsto \boxed{\sigma_a}}]\rho \sqsubseteq [\overline{b \mapsto \boxed{\sigma}}][\overline{a \mapsto \tau}]\rho$. With reflexivity of $\preceq$, we get $[\overline{a \mapsto \boxed{\sigma_a}}]\rho \preceq\sqsubseteq [\overline{b \mapsto \boxed{\sigma}}][\overline{a \mapsto \tau}]\rho$ as required.

- Case ABS. We have that $\Gamma_1 \vdash \lambda x\,.\,e : \tau \to \rho$ given that $\Gamma_1, (x{:}\tau) \vdash e : \rho$. By induction hypothesis we get $\Gamma_2, (x{:}\tau) \vdash^{\mathsf{sd}} e : \rho'_0$ with $\rho'_0 \preceq\sqsubseteq \rho$. Then, consider all $\overline{a} \in ftv(\tau, \rho) - ftv(\Gamma_2)$. Consider the substitution: $[\overline{a \mapsto \boxed{a}}]$ and $[\overline{a \mapsto \boxed{a}}](\tau \to \rho) \preceq\sqsubseteq \tau \to \rho$, as required if we apply rule SDABS to get $\Gamma_2 \vdash^{\mathsf{sd}} \lambda x\,.\,e : [\overline{a \mapsto \boxed{a}}](\tau \to \rho)$.
- Case INST. We have in this case that $\Gamma_1 \vdash e : [\overline{a \mapsto \tau'}]\rho'$ given that $\Gamma_1 \vdash e : \forall \overline{a}\,.\,\rho'$. By induction hypothesis we have that for all vectors $\overline{\sigma}$, we have $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho'_0$ with $\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \boxed{\sigma}}]\rho'$. We need to show that $\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \tau'}]\rho'$. By picking $\overline{\sigma} = \lfloor\overline{\tau'}\rfloor$, we know that:

$$\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \boxed{\lfloor\tau'\rfloor}}]\rho' \sqsubseteq [\overline{a \mapsto \tau'}]\rho$$

  and by transitivity of $\sqsubseteq$, we get $\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \tau'}]\rho$ as required.
- Case GEN. We have that $\Gamma_1 \vdash e : \forall \overline{a}\,.\,\rho'$ given that $\Gamma_1 \vdash e : \rho'$ and $\overline{a}\#\Gamma_1$. By induction hypothesis $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho'_0$ with $\rho'_0 \preceq\sqsubseteq \rho'$. We need to find a ρ''_0 such that $\rho''_0 \preceq\sqsubseteq [\overline{a \mapsto \boxed{\sigma}}]\rho'$. Because $\vdash^{\mathsf{DM}} \Gamma_2 \leq \Gamma_1$ it is easy to prove that also $\overline{a}\#\Gamma_2$. Then, by Theorem 5.3.12, it is $\Gamma_2 \vdash^{\mathsf{sd}} e : [\overline{a \mapsto \boxed{\sigma}}]\rho'_0$. Let $\rho''_0 = [\overline{a \mapsto \boxed{\sigma}}]\rho'_0$. Because $\rho'_0 \preceq\sqsubseteq \rho'$, by substitution stability for $\preceq\sqsubseteq$ we can get $[\overline{a \mapsto \boxed{\sigma}}]\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \boxed{\sigma}}]\rho'$ as required.
- Case SUBS. We have that $\Gamma_1 \vdash e : \rho''$ given that $\Gamma_1 \vdash e : \rho'$ and $\rho' \preceq\sqsubseteq \rho''$. By induction hypothesis we get $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho'_0$ with $\rho'_0 \preceq\sqsubseteq \rho'$ and by transitivity of $\preceq\sqsubseteq$ (Lemma 5.3.15) we get $\rho'_0 \preceq\sqsubseteq \rho''$ as required.
- Case LET. In this case we have that: $\Gamma_1 \vdash \mathtt{let}\ x = u\ \mathtt{in}\ e : \rho'$, given that $\Gamma_1 \vdash u : \forall \overline{a}\,.\,\rho$ and $\Gamma_1, (x{:}\forall \overline{a}\,.\,\rho) \vdash e : \rho'$. By induction hypothesis we get that for all vectors σ there exists a ρ'_0 with: $\Gamma_2 \vdash^{\mathsf{sd}} u : \rho'_0$ and $\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \boxed{\sigma}}]\rho$. Pick in particular the ρ'_0 with $\rho'_0 \preceq\sqsubseteq [\overline{a \mapsto \boxed{a}}]\rho$. Then, it is easy to see that $[\overline{a \mapsto \boxed{a}}]\rho \sqsubseteq \rho$ and hence $\rho'_0 \preceq\sqsubseteq \rho$ by transitivity of $\sqsubseteq$. Since $\vdash^{\mathsf{DM}} \Gamma_2 \leq \Gamma_1$ it is the case that $ftv(\Gamma_2) \subseteq ftv(\Gamma_1)$ (by Lemma 5.3.6), and hence for $\overline{b} = ftv(\rho) - ftv(\Gamma_2)$ we get that $\vdash^{\mathsf{DM}} \forall \overline{b}\,.\,\rho \leq \forall \overline{a}\,.\,\rho$. By induction hypothesis then we get that $\Gamma_2, (x{:}\forall \overline{b}\,.\,\rho) \vdash^{\mathsf{sd}} e : \rho'_1$ such that $\rho'_1 \preceq\sqsubseteq \rho'$. Applying rule SDLET finishes the case.
- Case ANN. We have that $\Gamma_1 \vdash (e :: \sigma) : \sigma$, given that $\Gamma_1 \vdash e : \sigma'_1$ and $\vdash^{\mathsf{F}} \lfloor\sigma'_1\rfloor \leq \sigma$. Assume that

$\sigma'_1 = \forall \overline{a}\,.\,\rho'_1$. Then, by induction we get that for all $\overline{\sigma}$ there exists a ρ'_0 with $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho'_0$ and $\rho'_0 \preceq\sqsubseteq \lceil \overline{a \mapsto \boxed{\sigma}} \rceil \rho'_1$. Pick in particular the ρ'_0 that makes true that $\rho'_0 \preceq\sqsubseteq \lceil \overline{a \mapsto \boxed{a}} \rceil \rho'_1$. Then let $\overline{b} = ftv(\rho'_0) - ftv(\Gamma_2)$. If $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho'_1 \rfloor \leq \sigma$ then it must also be $\vdash^{\mathsf{F}} \lfloor \forall \overline{b}\,.\,\rho'_0 \rfloor \leq \sigma$, by using transitivity of $\vdash^{\mathsf{F}}$ since, $\vdash^{\mathsf{F}} \lfloor \forall \overline{b}\,.\,\rho'_0 \rfloor \leq \lfloor \forall \overline{a}\,.\,\rho'_1 \rfloor$. The case concludes by appealing to rule SDANN in the same way as in the case of rule VAR.

- Case APP. We have that $\Gamma_1 \vdash e_1\ e_2 : \sigma'_2$ given that $\Gamma_1 \vdash e_1 : \sigma'_1 \to \sigma'_2$, $\Gamma_1 \vdash e_2 : \sigma'_3$, and $\vdash^{\mathsf{F}} \lfloor \sigma'_3 \rfloor \leq \lfloor \sigma'_1 \rfloor$. By induction hypothesis we have that $\Gamma_2 \vdash^{\mathsf{sd}} e_1 : \rho'_1$ with $\rho'_1 \preceq\sqsubseteq \sigma'_1 \to \sigma'_2$. Let ρ''_1 be the particular type such that $\rho'_1 \preceq \rho''_1 \sqsubseteq \sigma'_1 \to \sigma'_2$. Then, by Lemma 5.3.13 we have two cases:

  - $\rho''_1 = \boxed{\sigma_1 \to \sigma_2}$ with $\boxed{\sigma_1} \sqsubseteq \sigma'_1$ and $\boxed{\sigma_2} \sqsubseteq \sigma'_2$, or
  - $\rho''_1 = \sigma''_1 \to \sigma''_2$ with $\sigma''_1 \sqsubseteq \sigma'_1$ and $\sigma''_2 \sqsubseteq \sigma'_2$.

  Let σ'_{11} be either σ''_1 or $\boxed{\sigma_1}$. In any case it is the case that $\lfloor \sigma'_{11} \rfloor = \lfloor \sigma'_1 \rfloor$. And let σ'_{22} be either σ''_2 or $\boxed{\sigma_2}$. Moreover, let us assume that $\sigma'_3 = \forall \overline{b}\,.\,\rho'_3$. By induction hypothesis there exists a ρ''_3 such that $\Gamma_2 \vdash^{\mathsf{sd}} e_3 : \rho''_3$ and $\rho''_3 \preceq\sqsubseteq \lceil \overline{b \mapsto \boxed{b}} \rceil \rho'_3$. Moreover let $\overline{a} = ftv(\rho''_3) - ftv(\Gamma_2)$. It must be the case that: $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho''_3 \rfloor \leq \lfloor \forall \overline{b}\,.\,\rho'_3 \rfloor$ and we know that $\vdash^{\mathsf{F}} \lfloor \sigma'_3 \rfloor \leq \lfloor \sigma'_1 \rfloor$. By transitivity of $\vdash^{\mathsf{F}}$ then it is $\vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho''_3 \rfloor \leq \lfloor \sigma'_1 \rfloor = \lfloor \sigma'_{11} \rfloor$. To conclude the case using rule SDAPP we have the following. Assume that $\sigma'_2 = \forall \overline{c}\,.\,\rho'_2$. Pick any vector $\overline{\sigma}$. We need to find an instantiation of σ'_{22}, ρ'_{22} with $\rho'_{22} \preceq\sqsubseteq \lceil \overline{c \mapsto \boxed{\sigma}} \rceil \rho'_2$. We take cases according to the form of σ'_{22}:

  - Case $\sigma'_{22} = \boxed{\sigma_2}$. Then it is the case that $\boxed{\sigma_2} \sqsubseteq \forall \overline{c}\,.\,\rho'_2$, which means that $\overline{c} = \emptyset$ and σ_2 is a ρ-type, by inversion on the $\sqsubseteq$ relation. In which case we have the required result because σ'_{22} cannot be further instantiated.
  - Case $\sigma'_{22} = \sigma''_2 \sqsubseteq \forall \overline{c}\,.\,\rho'_2 = \sigma'_2$. By inversion again we either get that $\sigma''_2 = \forall \overline{c}\,.\,\rho'_2$, or that $\sigma''_2 = \forall \overline{c}\,.\,\rho''_2$ with $\rho''_2 \sqsubseteq \rho'_2$ and by the substitution lemma for $\sqsubseteq$ we get $\lceil \overline{c \mapsto \boxed{\sigma}} \rceil \rho''_2 \sqsubseteq \lceil \overline{c \mapsto \boxed{\sigma}} \rceil \rho'_2$, i.e. by reflexivity of $\preceq$, $\lceil \overline{c \mapsto \boxed{\sigma}} \rceil \rho''_2 \preceq\sqsubseteq \lceil \overline{c \mapsto \boxed{\sigma}} \rceil \rho'_2$ and $\vdash^{\mathsf{inst}} \sigma'_{22} \leq \lceil \overline{c \mapsto \boxed{\sigma}} \rceil \rho''_2$, as required.

□

We have the more general completeness corollary below.

Corollary 5.3.17. *If* $\Gamma \vdash e : \rho'$ *then* $\Gamma \vdash^{\mathsf{sd}} e : \rho'_0$ *with* $\rho'_0 \preceq\sqsubseteq \rho'$.

Proof. The result follows by Theorem 5.3.16, observing that $\vdash^{\mathsf{DM}}$ is reflexive. □

To see an example of this, consider the environment $\Gamma = \{\texttt{head}{:}\forall a\,.\,[a] \to a, \texttt{ids}{:}[\forall b\,.\,b \to b]\}$. Now, one can see that $\Gamma \vdash (\texttt{head ids}) : \boxed{\forall c\,.\,c \to c} \to \boxed{\forall c\,.\,c \to c}$ but in the syntax-directed system we only get $\Gamma \vdash^{\mathsf{sd}} (\texttt{head ids}) : \boxed{\forall a\,.\,a \to a}$, and it is the case that $\boxed{\forall a\,.\,a \to a} \preceq\sqsubseteq \boxed{\forall c\,.\,c \to c} \to \boxed{\forall c\,.\,c \to c}$.

We also state one further corollary, which is a key ingredient to showing the implementability of the syntax-directed system by a low-level algorithm (to be described in Chapter 6).

Corollary 5.3.18 ($\vdash^{\mathsf{DM}}$-saturation of typing)**.** *If* $\Gamma_1 \vdash^{\mathsf{sd}} e : \rho'_1$ *and* $\vdash^{\mathsf{DM}} \Gamma_2 \leq \Gamma_1$ *then* $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho'_2$ *with* $\rho'_2 \preceq\sqsubseteq \rho'_1$.

Proof. The corollary follows by appealing to Theorem 5.3.2 and Theorem 5.3.16. □

Corollary 5.3.18 means that if we change the types of expressions in the environments to be the most general according to the predicative $\vdash^{\mathsf{DM}}$, typeability is not affected. This property is important for type inference completeness for the following reason: All types that are pushed in the environment are box-free and hence can only be related to the most general one along the $\vdash^{\mathsf{DM}}$ relation—their polymorphic parts are determined by annotations. In fact the algorithm will choose the most general of them according to $\vdash^{\mathsf{DM}}$. Therefore, if an expression is typeable in the declarative type system with bindings in the environments that *do not have their most general types*, the above corollary shows that the expression will also be typeable if these bindings are assigned their most general types, that is, the types that the algorithm infers for them.

5.4 Summary

We have presented the declarative specification of FPH, a type system that supports impredicative polymorphism and is based on System F types. It is reminiscent of Damas-Milner as it allows instantiation and generalization anywhere in a typing derivation. It is expressive and economic in annotations as programmers need only annotate `let`-bindings or λ-abstractions with rich types. In

fact only those `let`-bindings with rich types that result from some impredicative instantiation (i.e. contain boxes) need be annotated. We additionally gave a syntax-directed specification for FPH and the next step is to give an algorithm that implements that specification.

Chapter 6

FPH algorithmic implementation

The syntax-directed specification of FPH in Figure 5.7 can be implemented by a low-level constraint-based algorithm, which resembles the algorithm of ML^{F} [26]. To ease the transition into the detailed low-level description of the algorithm, we present below a high-level description of the basic ideas behind the implementation and we elaborate in the rest of the chapter. The metatheory of the implementation can be found in Appendix A.

Like Hindley-Damas-Milner type inference [5, 35], the FPH algorithm creates fresh unification variables to instantiate polymorphic types and to use as the argument types of abstractions. In Hindley-Damas-Milner type inference these variables are unified with other types. Hence, a Hindley-Damas-Milner type inference engine maintains a set of *equality constraints* that map each unification variable to some type, updating the constraints as type inference proceeds.

Our algorithm uses a similar structure to Hindley-Damas-Milner type inference, but maintains both equality and *instance constraints* during type inference, so we use the term *constrained variable* instead of unification variable. A constrained variable in the algorithm corresponds to a box in the high-level specification. To distinguish between constrained variables and (rigid) quantified variables, we use greek letters α, β, for the former. Therefore, the algorithm uses types with the following

syntax:

$$\begin{array}{lcl} \tau & ::= & a \mid \tau \to \tau \mid \alpha \\ \rho & ::= & \tau \mid \sigma \to \sigma \\ \sigma & ::= & \forall \overline{a}\,.\,\rho \end{array}$$

We call such types *algorithmic*. An algorithmic type with no constrained variables can be viewed as an ordinary System F (box-free) type, and we omit explicit coercions when we need to do such a conversion.

The need for instance constraints can be motivated by the (algorithmic) typing of `choose id`, where `choose` has type $\forall a\,.\,a \to a \to a$ and `id` has type $\forall b.b \to b$. First, since `choose` has type $\forall a\,.\,a \to a \to a$, we may instantiate the quantified variable a with a fresh constrained variable α. However, when we meet the argument `id`, it becomes unclear whether α should be equal to $\beta \to \beta$ (that would arise from instantiating the type of `id`), or $\forall b\,.\,b \to b$ (if we do not instantiate `id`). In the high-level specification we can clairvoyantly make a (potentially boxed) choice that suits us. The algorithm does not have the luxury of clairvoyance, so rather than making a choice, it must instead simply record an *instance constraint*. In this case, the instance constraint specifies that α can be *any instance* of $\forall b\,.\,b \to b$. To express this, at first approximation, we need constraints of the form $\alpha \geq \sigma$.

However, we need to go slightly beyond this constraint form. Consider the program `f (choose id)` where `f` has type $\forall c\,.\,c \to c$. After we instantiate the quantified variable c with a fresh variable γ, we must constrain γ by the type of `choose id`, thus

$$\gamma \geq (\text{principal type of } \texttt{choose id})$$

But, the principal type of `choose id` must be a type that is quantified *and* constrained at the same time: $[\alpha \geq \forall b\,.\,b \to b] \Rightarrow \alpha \to \alpha$. Following ML^F [26], this *scheme* captures the *set* of all types for `choose id`, such as $\forall d\,.(d \to d) \to (d \to d)$ or $(\forall b\,.\,b \to b) \to (\forall b\,.\,b \to b)$. We hence extend the bounds of constrained variables to include $\gamma \geq \varsigma$, where ς is a scheme. Schemes ς are of the form $[c_1, \ldots, c_n] \Rightarrow \rho$, where each constraint c_i is of the form $(\alpha \geq \varsigma)$, or $(\alpha = \sigma)$, or $(\alpha\ \bot)$. To be technically precise, the syntax of constraints involves also special flags on variables, but as a first

approximation we may ignore this detail. The constraint $(\alpha\ \bot)$ means that α is unconstrained. Ordinary System F types can be viewed as schemes whose quantified variables are unconstrained, and hence the type $\forall b\,.\,b \to b$ can be written as $[\beta\ \bot] \Rightarrow \beta \to \beta$. The meaning of the constraint $\gamma \geq \varsigma$ is that γ belongs in the *set of System F types that* ς *represents*, which we write $[\![\varsigma]\!]$. For example, if $\varsigma = [\alpha \geq ([\beta\ \bot] \Rightarrow \beta \to \beta)] \Rightarrow (\alpha \to \alpha)$, then we have:

$$\begin{aligned} (\forall b\,.\,b \to b) \to (\forall b\,.\,b \to b) &\in [\![\varsigma]\!] \\ \forall c\,.\,(c \to c) \to c \to c &\in [\![\varsigma]\!] \\ \forall c\,.\,([c] \to [c]) \to [c] \to [c] &\in [\![\varsigma]\!] \end{aligned}$$

The algorithmic implementation of FPH manipulates such sets of System F types that may be assigned to expressions, by syntactically describing them as schemes.

We now proceed to the detailed description of the low-level algorithmic implementation of the syntax-directed type system of Figure 5.7. We give the precise syntax and semantics of constraints, and explain how the algorithm manipulates them, and conclude with formal statements of the most important properties that connect the implementation to the type system. Proofs can be found in Appendix A.

6.1 Types and constraints, formally

The formal language of types and constraints that are used in the implementation is given in Figure 6.1. Monomorphic types τ include "rigid" variables written with lowercase latin characters (such as quantified variables), and constrained (unification) variables written with lowercase greek characters. As outlined above, constrained variables represent the boxes of the high-level type system. A box that is filled in with some type, algorithmically corresponds to a constrained variable and a substitution that satisfies the constraint that is attached to the variable—and which maps the variable to the contents of the box.

Constraints C are finite maps of the form $\overline{\alpha_\mu\ bnd}$. Every variable α is mapped to a flag μ and a bound bnd. The domain of a constraint C, written $dom(C)$, is the set $\{\alpha \mid (\alpha_\mu\ bnd) \in C\}$. Bounds

Algorithmic types	σ	$::=$	$\forall \overline{a}\,.\,\rho$
	ρ	$::=$	$\tau \mid \sigma \to \sigma$
	τ	$::=$	$a \mid \alpha \mid \tau \to \tau$
Constraints	C, D, E	$::=$	$\overline{\alpha_\mu\ bnd}$
Flags	μ	$::=$	$\mathsf{m} \mid \star$
Schemes	ς	$::=$	$[C] \Rightarrow \rho$
Bounds	bnd	$::=$	$(\geq \varsigma) \mid (= \sigma) \mid \bot$
Flagsets	Δ	$::=$	$\overline{\alpha_\mu}$

Figure 6.1: Algorithmic types, schemes, and constraints

bnd specify either flexible constraints, rigid constraints, or the fact that a constrained variable is yet completely unconstrained.

The new bit compared to our introductory description at the beginning of this chater is the presence of the flags μ. These flags are used to ensure that the variables that enter the environments are never unified with types that contain quantifiers, and that these variables remain hence monomorphic. This is important to ensure that abstraction argument types are kept monomorphic, as the high-level specification requires. Those constrained variables that must be kept monomorphic are flagged with m, whereas variables with no restrictions are flagged with $\star$.

Flagsets Δ are a notational convenience and can be viewed as special constraints that only give information about the flags of the variables in their domain. For a constraint C, we write Δ_C for the flagset of the domain of C. We write $\Delta(\beta) = \mu$ whenever $\beta_\mu \in \Delta$, and $dom(\Delta)$ for the domain of Δ. We write $\Delta_1\Delta_2$ for the disjoint union of the two flagsets with respect the variables in their domains. Finally, and overloading notation, we write $fcv(\cdot)$ (for *free constrained variables*) for the set of constrained variables of the argument. We write $fcv(C)$ for the set of all constrained variables appearing anywhere in C. We come back to the precise definition of $fcv(bnd)$ in the next section.

6.2 Bounds and the meaning of constraints

We next turn to the description of the semantics of constraints and schemes. We first need, however, some technical definitions. Because schemes $[D]\Rightarrow\rho$ bind the constrained variables in the domain of D inside the body ρ we can assume without loss of generality that D can be concatenated with any external constraint C, to form the constraint $C\bullet D$. This notation ensures freshness of the constrained variables in the domain of D.

Definition 6.2.1 (Constraint disjoint concatenation)**.** For two constraints C and D, we write $C\bullet D$ to denote their union whenever $dom(D)\#fcv(C)$.

We now define the set of *reachable constrained variables* of types and schemes *through a constraint*, with the (overloaded) function $\widehat{C}(\cdot)$, below:

$$
\begin{array}{lcl}
\widehat{C}(\alpha) \mid \alpha \notin dom(C) & = & \{\alpha\} \\
\widehat{C}(\alpha) \mid (\alpha_\mu \perp) \in C & = & \{\alpha\} \\
\widehat{C}(\alpha) \mid (\alpha_\mu = \sigma) \in C & = & \{\alpha\} \cup \widehat{C}(\sigma) \\
\widehat{C}(\alpha) \mid (\alpha_\mu \geq \varsigma) \in C & = & \{\alpha\} \cup \widehat{C}(\varsigma) \\
\widehat{C}(a) & = & \emptyset \\
\widehat{C}(\sigma_1 \rightarrow \sigma_2) & = & \widehat{C}(\sigma_1) \cup \widehat{C}(\sigma_2) \\
\widehat{C}(\forall \overline{a}\,.\,\rho) & = & \widehat{C}(\rho) \\
\widehat{C}([D]\Rightarrow\rho) & = & \widehat{C\bullet D}(\rho, dom(D)) - dom(D)
\end{array}
$$

In the case for schemes, $\widehat{C}([D]\Rightarrow\rho)$, we gather all variables reachable from ρ and the domain of D through $C\bullet D$ except those quantified by the scheme ($dom(D)$). However, an invariant of our algorithm will be that there exist no useless quantified variables in $dom(D)$ in any scheme of the form $[D]\Rightarrow\rho$ —hence in reality it will suffice to only consider $\widehat{C\bullet D}(\rho) - dom(D)$.

Finally, the function $\widehat{C}(\cdot)$ exists since any constraint involves a finite set of constrained variables, and we can compute it by an iteration process that adds new variables from the constraint C at each step. Although formally this corresponds to a guaranteed-to-exist fixpoint, we will simply be using the above set of equalities as the actual definition of $\widehat{C}(\sigma)$. We can now define $fcv(\cdot)$ using the reachable variables through the empty constraint. In particular $fcv(\varsigma) = \widehat{\emptyset}(\varsigma)$. With this definition in place, the overloaded notation $fcv(bnd)$ can be defined: $fcv(\perp) = \emptyset$, and moreover $fcv(\geq \varsigma) = fcv(\varsigma)$

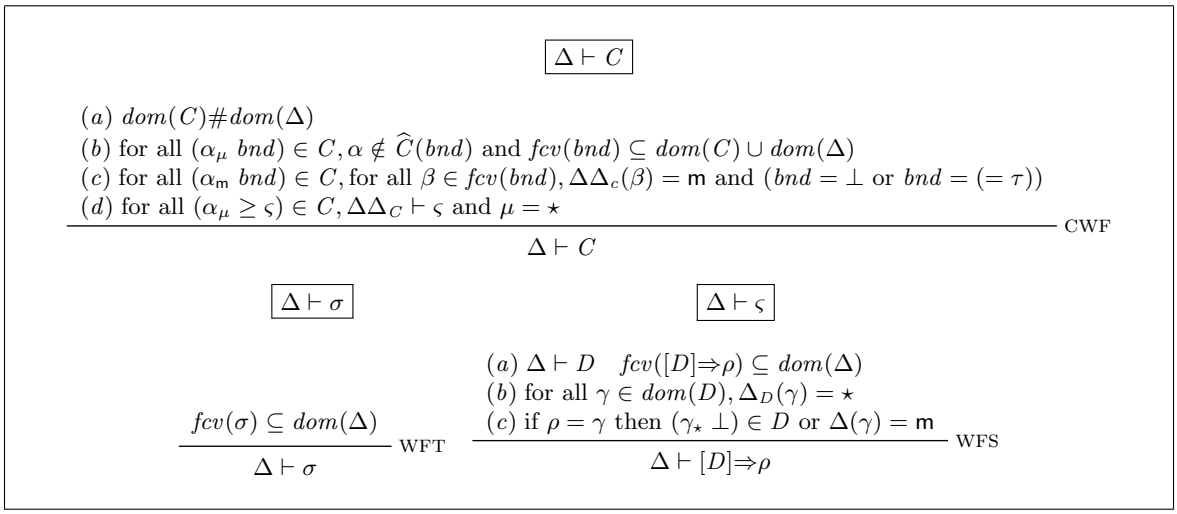

Figure 6.2: Well formed constraints, types, and schemes

and $fcv(= \sigma) = fcv(\sigma)$.

6.2.1 Semantics of constraints

We turn now to the meaning of constraints and bounds. Before presenting the actual definition, we give conditions on the constraints that will help get insights about their meaning. Figure 6.2 presents conditions on constraints, types, and schemes, with the notations $\Delta \vdash C$, $\Delta \vdash \sigma$, and $\Delta \vdash \varsigma$ respectively. These conditions are preserved during type inference.

The judgement $\Delta \vdash C$ ensures that the constraint C is well-formed in a given flagset Δ. In rule CWF, condition (a) ensures that the domain of C consists of fresh variables with respect to Δ. Condition (b) ensures that there exist no cycles in C. No variable should be reachable by itself through the constraint. The reason is that, if there exists a variable that is reachable from itself through the constraint, then there can be no substitution of that variable to a System F type that satisfies the constraint, unless all types appearing in the cycle are single constrained variables. As we shall see, such "benign" cycles can always be eliminated and hence our algorithm produces constraints with no cycles at all. Condition (b) also ensures that all constrained variables in the bounds of C belong either in the domain of C or in the external $dom(\Delta)$. Condition (c) ensures that if a variable is flagged as monomorphic with m then it must be either unrestricted (bound to $\bot$) or mapped rigidly

to some type τ whose constrained variables must be themselves flagged as m either in Δ or Δ_C. Condition (d) ensures that all flags of flexibly bound variables are $\star$. It makes no sense for a variable that must be monomorphic to be flexibly bound, since a monomorphic variable can only be equal to a monomorphic type that belongs in the interpretation of the scheme. Finally, condition (d) also asserts that in all bounds of the form $\alpha_\star \geq \varsigma$, the constraint ς must itself be well-formed in the extended flagset $\Delta\Delta_C$.

The judgement $\Delta \vdash \sigma$ simply ensures that all constrained variables appear in the flagset Δ. The judgement $\Delta \vdash \varsigma$ ensures that the scheme ς is well-formed in the flagset Δ. Let us examine in detail the conditions of rule WFS, which asserts that $\Delta \vdash [D]\Rightarrow\rho$. Condition (a) ensures that the constraint D of the scheme is well-formed in Δ, and that all the free constrained variables of $[D]\Rightarrow\rho$ belong in the domain of Δ. Condition (b) states that all constrained variables in $dom(D)$ cannot be flagged as m. The reason is the following: If such a variable were flagged with m then it must have been the case that some variable *from the environment* has been mapped to it—in order to force its flagging. It follows that such a variable could not have been quantified in the scheme anyway, because schemes are constructed as results of generalization. Condition (c) ensures that the scheme $[D]\Rightarrow\rho$ is *normal*. That is, if the body is a single variable, then it must either be bound to $\bot$ in $dom(D)$ or it must be monomorphically flagged in Δ. During inference, a normalization procedure ensures that schemes are normal, and our unification algorithm preserves normal schemes. We return to this condition in Section 6.3.

We are particularly interested in constraints that are *well-formed with no assumptions*, i.e. constraints C where $\vdash C$. Our type inference algorithm produces and manipulates such constraints. Well-formed constraints correspond to sets of substitutions from constrained variables to constrained-variable-free types, generalizing the interpretation of schemes as sets of constrained-variable-free (System F) types. The idea exists in the ML^F line of work, and we take it a step further, to formalize type inference using this interpretation of constraints, instead of employing a syntactic instance relation that syntactically encodes the set semantics. In order to present the interpretation of constraints and schemes we need to introduce the *quantifier rank of a constraint*.

Definition 6.2.2 (Constraint quantifier rank)**.** The quantifier rank of a constraint C, written $q(C)$,

is defined as:

$$q(C) = |dom(C)| + \sum_{(\alpha_\mu \geq [D] \Rightarrow \rho) \in C} q(D)$$

Intuitively the metric $q(C)$ sums up the number of all constrained variables in C, including the constrained variables of all schemes appearing in C.

Definition 6.2.3 (Well-formed constraint interpretation)**.** A substitution θ from constrained variables to constrained-variable-free (System F) types satisfies C, written $\theta \models C$, iff:

1. for all $(\alpha_{\mathsf{m}}\ bnd) \in C$, it is the case that $\theta\alpha$ is quantifier-free,
2. for all $(\alpha_\mu = \sigma) \in C$, it is the case that $\theta\alpha = \theta\sigma$, and
3. for all $(\alpha_\mu \geq \varsigma) \in C$, it is the case that $\theta\alpha \in [\![\theta\varsigma]\!]$,

where:

$$[\![[D] \Rightarrow \rho]\!] = \{\forall \overline{b}\,.\,\theta_D\rho \mid \overline{b} \# ftv([D] \Rightarrow \rho) \text{ and } \theta_D \models D\}$$

Of course we will be interested mainly in the interpretation of well-formed constraints, but the definition is valid for arbitrary constraints, so long as θ maps all constrained variables to System F types.

Condition (1) ensures that all monomorphic variables are indeed mapped to monomorphic types. Condition (2) ensures that equalities are respected. Condition (3) realizes our set interpretation of schemes. We observe that the domain of θ must be a superset of the domain of C and hence in clause (3), $\theta\varsigma$ contains no free constrained variables. The interpretation $[\![[D] \Rightarrow \rho]\!]$ is nothing more than a generalization of System F instance. It requires the existence of a substitution θ_D that satisfies D, and we can generalize over type variables ($\overline{b}$) that θ_D introduced (condition $\overline{b} \# ftv([D] \Rightarrow \rho)$). The free type variables of a scheme are all the rigid variables appearing free anywhere in the scheme. Intuitively, if a variable b belongs in the set $ftv([D] \Rightarrow \rho)$ then it must belong in *any* type contained in the interpretation of the scheme, and hence it cannot be quantified. We ensure this condition by requiring $\overline{b} \# ftv([D] \Rightarrow \rho)$, which is really only a generalization of the corresponding condition in the Damas-Milner instance. Notice that, because of the well-formedness condition (c) in rule WFS, $\theta_D\rho$ may add no top-level quantifiers on ρ. Finally, the mutual definition of satisfying substitutions and

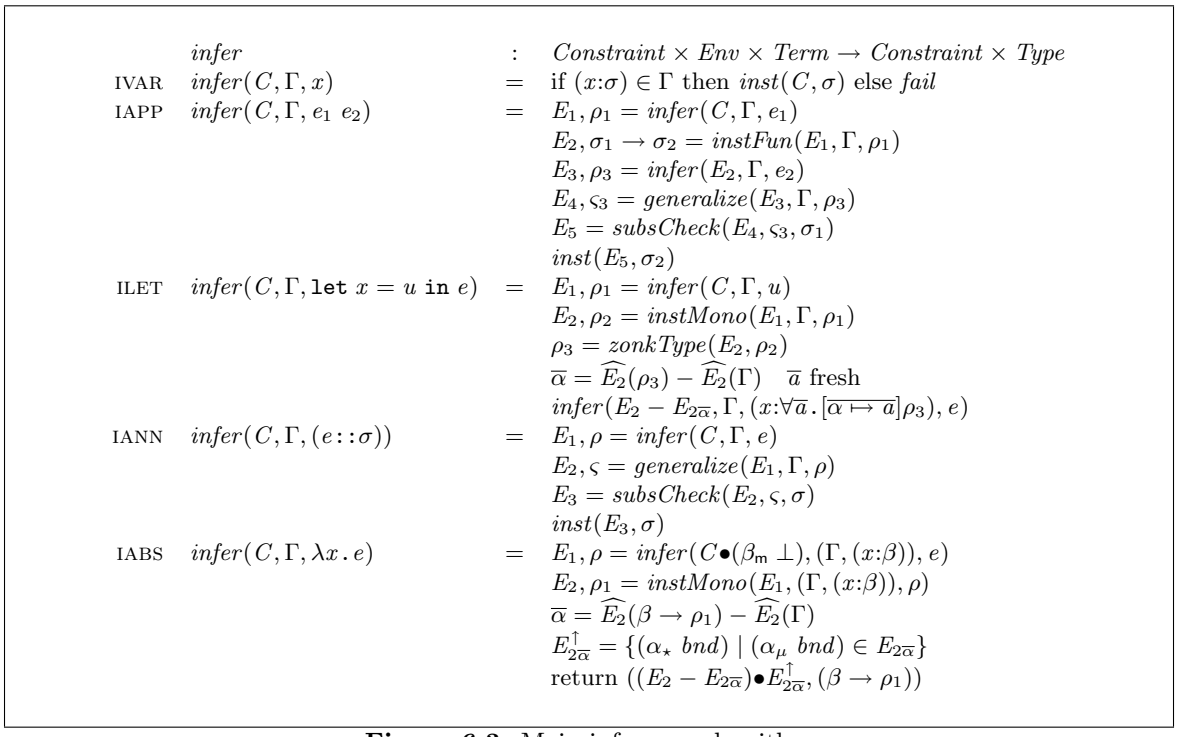

Figure 6.3: Main inference algorithm

scheme interpretations is well-formed by using as metric the quantifier rank $q(C)$ for $\theta \models C$, and $q(D)$ for the interpretation of $[D]\Rightarrow\rho$.

As an example, let us see how we can derive that

$$\forall b\,.\,(b \to b) \to (b \to b) \in [\![[\alpha_\star \geq ([\beta_\star \bot]\Rightarrow\beta \to \beta)]\Rightarrow(\alpha \to \alpha)]\!]$$

Consider the substitution $\theta_\alpha = [\alpha \mapsto (b \to b)]$. Since $(b \to b) \to (b \to b) = \theta_\alpha(\alpha \to \alpha)$ it suffices to show that $\theta_\alpha \models (\alpha_\star \geq [\beta_\star \bot]\Rightarrow\beta \to \beta)$. This will be the case if $\theta_\alpha\alpha \in [\![[\beta_\star \bot]\Rightarrow\beta \to \beta]\!]$, that is, if $b \to b \in [\![[\beta_\star \bot]\Rightarrow\beta \to \beta]\!]$. This follows by picking $\theta_\beta = [\beta \mapsto b]$.

6.3 Inference implementation

We proceed with explaining the basic ideas behind the reference implementation. This implementation is described in Figure 6.3, and some auxiliary functions are given in Figures 6.4 and 6.5.

The top-level inference algorithm is given with the function *infer*, which follows the syntax-directed presentation of Figure 5.7, and has signature:

$$\mathit{infer} : \mathit{Constraint} \times \mathit{Env} \times \mathit{Term} \to \mathit{Constraint} \times \mathit{Type}$$

The function *infer* accepts a constraint C_1, an environment Γ, and a term e. A call to $\mathit{infer}(C_1, \Gamma, e)$ either fails with *fail* or returns an updated constraint C_2 and a type ρ.

The clause IVAR deals with variables. If a variable is bound in the environment, its type is returned and instantiated using a call to $\mathit{inst}(C, \sigma)$, from Figure 6.4. A call to $\mathit{inst}(C, \sigma)$ merely instantiates the top-level quantified variables of σ, replacing them with fresh unrestricted and $\star$-flagged variables. This function effectively implements the $\vdash^{\mathsf{inst}}$ judgement of the syntax-directed presentation.

The most interesting case, which demonstrates the power of schemes, is the clause corresponding to applications (IAPP). In a call to $\mathit{infer}(C, \Gamma, e_1\ e_2)$ we perform the following steps:

- We first infer a type ρ_1 for e_1 and an updated constraint E_1, by calling $\mathit{infer}(C, \Gamma, e_1)$.
- However, type ρ_1 may itself be a constrained type variable, that is, it may correspond to a single box in the syntax-directed specification. The function $\mathit{instFun}(E_1, \Gamma, \rho_1)$ in Figure 6.4 implements the relation $\preceq\!\sqsubseteq^{\rightarrow}$. Intuitively, if ρ_1 is a variable α that is not monomorphic in E_1, that is if $(\alpha_\star = \forall \overline{a}\,.\,\rho) \in E_1$ or $(\alpha_\star \geq [D] \Rightarrow \rho) \in E_1$, it is replaced by an instantiation of its bound (that is, with $[\overline{a \mapsto \alpha}]\rho$ for fresh $\overline{\alpha}$, or ρ respectively). This is achieved with a function called *normalize* (called from *generalize*, again described in Figure 6.4), that accepts a constraint and a type and inlines/instantiates all constraint bounds if the type is a single variable. For example

$$\begin{array}{lcl} \mathit{normalize}((\alpha_\star = \forall b\,.\,b \to b), \alpha) & = & [\beta_\star\ \bot] \Rightarrow \beta \to \beta \\ \mathit{normalize}((\alpha_\star \geq [\beta_\star\ \bot] \Rightarrow \beta), \alpha) & = & [\beta_\star\ \bot] \Rightarrow \beta \\ \mathit{normalize}((\alpha_\star = \sigma), \alpha \to \alpha) & = & [\alpha_\star = \sigma] \Rightarrow \alpha \to \alpha \end{array}$$

  With this step we have effectively performed an instantiation along $\preceq$. To push boxes down along $\sqsubseteq^{\rightarrow}$ we consider two cases: if the inlined and instantiated type was a function type we

simply return it, otherwise we attempt to unify the inlined type with some function type $\beta \to \gamma$ consisting of fresh unification variables β and γ. We return the resulting type as $\sigma_1 \to \sigma_2$.

- Next, we infer a type ρ_3 and an updated constraint E_3 for the argument e_2, with $E_3, \rho_3 = \mathit{infer}(E_2, \Gamma, e_2)$.
- At this point we need to compare the function argument type σ_1 to the type that we have inferred for the argument. However, we do not know the precise type of the argument at this point and hence we call $\mathit{generalize}(E_3, \Gamma, \rho_3)$ to get back a new constraint E_4 and a scheme ς_3. The scheme ς_3 expresses the set of *all possible types* of the argument e_2. We return to *generalize* later in this section.
- Now that we have a scheme ς_3 expressing all possible types of the argument e_2, we must check that the required type σ_1 belongs in the set that ς_3 expresses. This is achieved with the call to $\mathit{subsCheck}(E_4, \varsigma_3, \sigma_1)$, which simply returns an updated constraint E_5. We describe the function *subsCheck* in Section 6.3.1.
- Finally, type σ_2 may be equal to some type $\forall \overline{a} \,.\, \rho_2$, which we instantiate to $[\overline{a \mapsto \alpha}]\rho_2$ for fresh $\overline{\alpha}$. This is achieved with the call to $\mathit{inst}(E_5, \sigma_2)$.

We now turn to *generalize* in Figure 6.4, with the following signature:

$$\mathit{generalize} : \mathit{Constraint} \times \mathit{Env} \times \mathit{Type} \to \mathit{Constraint} \times \mathit{Scheme}$$

The *generalize* function accepts a constraint, an environment, and a type ρ, which is typically the result of a prior call to *infer*. It returns a scheme which is well-formed and expresses the set of all possible System F types that can be generalized from ρ. The first step in the definition of *generalize* is to locate which variables can be generalized. These are the $\overline{\beta}$ which are the reachable variables of ρ through C, except for those that are reachable from the environment Γ through C. Notice that if the reachable variables of Γ through the constraint C were monomorphic this implies that there are *no monomorphic variables* in $\overline{\beta}$. Hence we could at this point return $[C_{\overline{\beta}}] \Rightarrow \rho$ ($C_{\overline{\beta}}$ is the restriction of C to $\overline{\beta}$) and condition (b) of rule CWF would not be violated. However condition (c) could be violated, and hence we must *normalize* ρ in order to inline any polymorphism of ρ, if ρ is a plain variable. After the call to *normalize*, the returned scheme ς is a normal scheme that satisfies

$$
\begin{array}{lllll}
 & \mathit{inst} & : & \mathit{Constraint} \times \mathit{Type} \to \mathit{Constraint} \times \mathit{Type} \\
\textsc{inst} & \mathit{inst}(C, \forall \overline{a}\,.\,\rho) & = & \text{return } (C \bullet (\overline{\alpha}_\star\ \bot), [\overline{a \mapsto \alpha}]\rho) \\
 & & & \\
 & \mathit{normalize} & : & \mathit{Constraint} \times \mathit{Type} \to \mathit{Scheme} \\
\textsc{n}1 & \mathit{normalize}(C, \alpha) \mid (\alpha_\star \geq [D] \Rightarrow \rho) \in C & = & \text{return } ([C \bullet D - \alpha] \Rightarrow \rho) \\
\textsc{n}2 & \mathit{normalize}(C, \alpha) \mid (\alpha_\star = \sigma) \in C & = & \mathit{normalize}(C - \alpha, \sigma) \\
\textsc{n}3 & \mathit{normalize}(C, \forall \overline{a}\,.\,\rho) & = & \text{return } ([C \bullet (\overline{\alpha}.\ \bot)] \Rightarrow [\overline{a \mapsto \alpha}]\rho) \\
\textsc{n}4 & \mathit{normalize}(C, \rho) & = & \text{return } ([C] \Rightarrow \rho) \\
 & & & \\
\textsc{gen} & \mathit{generalize}(C, \Gamma, \rho) & = & \overline{\beta} = \widehat{C}(\rho) - \widehat{C}(\Gamma) \\
 & & & [E] \Rightarrow \rho_1 = \mathit{normalize}(C_{\overline{\beta}}, \rho) \\
 & & & \text{return } (C - \overline{\beta}, [E] \Rightarrow \rho_1) \\
 & & & \\
 & \mathit{instFun} & : & \mathit{Constraint} \times \mathit{Env} \times \mathit{Type} \to \mathit{Constraint} \times \mathit{Type} \\
\textsc{if}1 & \mathit{instFun}(C, \Gamma, \alpha) & = & E_1, ([E_g] \Rightarrow \rho_g) = \mathit{generalize}(C, \Gamma, \alpha) \\
 & & & E_2 = \mathit{eqCheck}(E_1 \bullet E_g \bullet (\beta_\star\ \bot) \bullet (\gamma_\star\ \bot), \rho_g, \beta \to \gamma) \\
 & & & \text{return } (E_2, \beta \to \gamma) \\
\textsc{if}2 & \mathit{instFun}(C, \Gamma, \sigma_1 \to \sigma_2) & = & \text{return } (C, \sigma_1 \to \sigma_2) \\
\textsc{if}3 & \mathit{instFun}(C, \Gamma, \_) & = & \mathit{fail} \\
 & & & \\
 & \mathit{instMono} & : & \mathit{Constraint} \times \mathit{Env} \times \mathit{Type} \to \mathit{Constraint} \times \mathit{Type} \\
\textsc{im}1 & \mathit{instMono}(C, \Gamma, \alpha) & = & E_1, ([E_g] \Rightarrow \rho_g) = \mathit{generalize}(C, \Gamma, \alpha) \\
 & & & E_2 = \mathit{mkMono}(E_1 \bullet E_g, \mathit{True}, \rho_g) \\
 & & & \text{return } (E_2, \rho_g) \\
\textsc{im}2 & \mathit{instMono}(C, \Gamma, \rho) & = & E = \mathit{mkMono}(C, \mathit{False}, \rho)\,;\text{return } (E, \rho)
\end{array}
$$

Figure 6.4: Auxiliary functions

condition (c). Finally, in the returned constraint we can safely discard all the bounds of $\overline{\beta}$.

To type `let`-bindings, clause ILET is applicable. In this case, we infer a type ρ_1 and an updated constraint E_1 for the expression to be bound. Subsequently, we must implement $\preceq\!\sqsubseteq$, so that the result type is box-free. As in the case for applications (where we had to implement $\preceq\!\sqsubseteq^{\to}$), we call the function *instMono* in Figure 6.4. This function, after inlining all bounds if the type is a single variable, proceeds with calling the *monomorphization* function *mkMono*, in Figure 6.5. Monomorphization ensures that all constrained variables of a type are mapped to quantifier types and are flagged (recursively) as m. The result is an updated constraint E_2 and a type ρ_2. However, ρ_2 now only contains constrained variables that are mapped to quantifier-free types. With the function *zonkType* (Figure 6.5) we create a type all of whose constrained variables (which must necessarily be m-flagged), are bound to $\bot$. Just as in Hindley-Damas-Milner type inference, we may now generalize over those that do not appear in the environment by replacing them with fresh rigid variables $\overline{a}$.

	$mkMono$	:	$Bool \times Constraint \times Type \rightarrow Constraint$
M1	$mkMono(f, C, \forall \overline{a}\,.\,\rho)$	=	if $(f \wedge \overline{a} \neq \emptyset)$ then *fail* else $mkMono(f, C, \rho)$
M2	$mkMono(f, C, \sigma_1 \rightarrow \sigma_2)$	=	$E = mkMono(f, C, \sigma_1)$ $mkMono(f, E, \sigma_2)$
M3	$mkMono(f, C, a)$	=	C
M4	$mkMono(f, C, \alpha) \mid (\alpha_{\mathsf{m}}\ bnd) \in C$	=	C
M5	$mkMono(f, C, \alpha) \mid (\alpha_\mu \geq [D] \Rightarrow \rho) \in C)$	=	$E = mkMono(True, C \bullet D, \rho)$ return $(E \leftarrow (\alpha_{\mathsf{m}} = \rho))$
M6	$mkMono(f, C, \alpha) \mid (\alpha_\mu = \sigma) \in C$	=	$E = mkMono(True, C, \sigma)$ return $(E \leftarrow (\alpha_{\mathsf{m}} = \sigma))$
M7	$mkMono(f, C, \alpha) \mid (\alpha_\star\ \bot) \in C$	=	return $(C \leftarrow (\alpha_{\mathsf{m}}\ \bot))$
Z1	$zonkType(C, \alpha) \mid (\alpha_\mu = \sigma) \in C$	=	$zonkType(\sigma)$
Z2	$zonkType(C, \alpha) \mid (\alpha_\mu\ \bot) \in C$	=	α
Z3	$zonkType(C, a)$	=	a
Z4	$zonkType(C, \sigma_1 \rightarrow \sigma_2)$	=	$zonkType(C, \sigma_1) \rightarrow zonkType(C, \sigma_2)$
Z5	$zonkType(C, \forall \overline{a}\,.\,\rho)$	=	$\forall \overline{a}\,.\,zonkType(C, \rho)$

Figure 6.5: Monomorphization and zonking

To infer a type for an annotated expression, clause IANN is applicable. In much the same way as for applications, a scheme is inferred for the expression and it is subsequently checked with *subsCheck* that the annotation belongs in the set that the scheme expresses.

Finally, λ-abstractions are typed (clause IABS) by creating a fresh m-flagged variable for the argument type, inferring a type for the body, forcing it to be monomorphic via *instMono*, and lifting the m flags from the variables that can be generalized.

Importance of normal schemes The case for applications illustrates why it is important for schemes to be normal. Consider the following code fragment:

```
head :: forall d. [d] -> d

foo :: (forall b. b -> b) -> Int


h = foo (head ids)
```

The application is certainly typeable in the specification. First we instantiate the type of `head` with $[\boxed{\forall b\,.\,b \rightarrow b}] \rightarrow \boxed{\forall b\,.\,b \rightarrow b}$ and the application `head ids` can be typed with $\boxed{\forall b\,.\,b \rightarrow b}$. Subsequently it is straightforward to check that $\vdash^{\mathsf{F}} \forall b\,.\,b \rightarrow b \leq \forall b\,.\,b \rightarrow b$ as rule SDAPP requires. However, let us

consider the algorithm operation. The type that will be inferred for `head ids` will simply be γ where $(\gamma_\star = \forall b\,.\,b \to b)$ is bound in the constraint. The non-normalized scheme is $[\gamma_\star = \forall b\,.\,b \to b] \Rightarrow \gamma$ whereas the normalized one would be $[\beta_\star\ \bot] \Rightarrow \beta \to \beta$. But now notice that according to Definition 6.2.3 it *is not* the case that $(\forall b\,.\,b \to b) \in [\![[\gamma_\star = \forall b\,.\,b \to b] \Rightarrow \gamma]\!]$. However, with the normal scheme, $[\beta_\star\ \bot] \Rightarrow \beta \to \beta$, *it is* the case that $(\forall b\,.\,b \to b) \in [\![[\beta_\star\ \bot] \Rightarrow \beta \to \beta]\!]$.

In short, non-normal schemes, when interpreted with Definition 6.2.3, do not capture all of the desired System F types. Either the interpretation and the implementation of instance checking should be different, or we should be using normal schemes. We chose to use normal schemes.

6.3.1 Instance checking and unification

The main type inference algorithm relies on the *subsCheck* function, which checks whether a type belongs in the interpretation of a scheme. The *subsCheck* function is mutually defined with several other functions; these together constitute our unification algorithm. The signatures of these functions are given below:

$$
\begin{array}{lcl}
\mathit{eqCheck} & : & \mathit{Constraint} \times \mathit{Type} \times \mathit{Type} \to \mathit{Constraint} \\
\mathit{subsCheck} & : & \mathit{Constraint} \times \mathit{Scheme} \times \mathit{Type} \to \mathit{Constraint} \\
\mathit{updateRigid} & : & \mathit{Constraint} \times \mathit{Var} \times \mathit{Type} \to \mathit{Constraint} \\
\mathit{updateFlexi} & : & \mathit{Constraint} \times \mathit{Var} \times \mathit{Scheme} \to \mathit{Constraint} \\
\mathit{join} & : & \mathit{Constraint} \times \mathit{Scheme} \times \mathit{Scheme} \to \mathit{Constraint} \times \mathit{Scheme}
\end{array}
$$

The definitions of these functions can be found in Figure 6.6 and Figure 6.7.

With the call to $\mathit{eqCheck}(C, \sigma_1, \sigma_2)$ we check that σ_1 can be equated to σ_2 and we produce an updated constraint or fail. Clause E1 checks rigid variables. Clauses E2 and E3 check the case where we need to equate a constrained variable α to make it equal to a type σ. In this case we must appeal to the function *updateRigid*, which updates the bound of a constrained variable so that it is equal to a type. Clause E4 equates two function types, whereas clause E5 equates polymorphic types, requiring that none of the quantified variables escapes in any constrained variable in the environment. In all other cases (clause E6) the algorithm fails.

$$
\begin{array}{lllll}
 & \mathit{eqCheck} & : & \mathit{Constraint} \times \mathit{Type} \times \mathit{Type} \to \mathit{Constraint} \\
\textsc{e}1 & \mathit{eqCheck}(C, a, a) & = & C \\
\textsc{e}2 & \mathit{eqCheck}(C, \alpha, \sigma) & = & \mathit{updateRigid}(C, \alpha, \sigma) \\
\textsc{e}3 & \mathit{eqCheck}(C, \sigma, \alpha) & = & \mathit{updateRigid}(C, \alpha, \sigma) \\
\textsc{e}4 & \mathit{eqCheck}(C, \sigma_1 \to \sigma_2, \sigma_3 \to \sigma_4) & = & E = \mathit{eqCheck}(C, \sigma_1, \sigma_3) \\
 & & & \mathit{eqCheck}(E, \sigma_2, \sigma_4) \\
\textsc{e}5 & \mathit{eqCheck}(C, \forall \overline{a}\,.\,\rho_1, \forall \overline{a}\,.\,\rho_2) & = & E = \mathit{eqCheck}(C, [\overline{a \mapsto b}]\rho_1, [\overline{a \mapsto b}]\rho_2) \\
 & & & \text{if } \overline{b} \# E \text{ then return } E \text{ else } \mathit{fail} \\
\textsc{e}6 & \mathit{eqCheck}(C, \_, \_) & = & \mathit{fail} \\
 & & & \\
 & \mathit{subsCheck} & : & \mathit{Constraint} \times \mathit{Scheme} \times \mathit{Type} \to \mathit{Constraint} \\
\textsc{s}1 & \mathit{subsCheck}(C, \varsigma, \beta) & = & \mathit{updateFlexi}(C, \beta, \varsigma) \\
\textsc{s}2 & \mathit{subsCheck}(C, [D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2) & = & \rho_3 = [\overline{c \mapsto b}]\rho_2 \\
 & & & E = \mathit{eqCheck}(C \bullet D, \rho_1, \rho_3) \text{ and } \overline{\gamma} = \widehat{E}(\mathit{dom}(C)) \\
 & & & \text{if } \overline{b} \# E_{\overline{\gamma}} \text{ then return } E_{\overline{\gamma}} \text{ else } \mathit{fail}
\end{array}
$$

Figure 6.6: Equivalence and instance checking

$$
\begin{array}{lllll}
\textsc{ur}1 & \mathit{updateRigid}(C, \alpha, \alpha) & = & \text{return } C \\
\textsc{ur}2 & \mathit{updateRigid}(C, \alpha, \beta) \mid ((\beta_\mu = \gamma) \in C) & = & \mathit{updateRigid}(C, \alpha, \gamma) \\
\textsc{ur}3 & \mathit{updateRigid}(C, \alpha, \beta) \mid & & \\
 & \quad ((\beta_\mu \geq [D] \Rightarrow \gamma) \in C) \wedge (\gamma \notin \mathit{dom}(D)) & = & \mathit{updateRigid}(C, \alpha, \gamma) \\
\textsc{ur}4 & \mathit{updateRigid}(C, \alpha, \sigma) & = & \text{if } \alpha \in \widehat{C}(\sigma) \text{ then } \mathit{fail} \text{ else } \mathit{doUpdateR}(C, \alpha, \sigma) \\
\textsc{dr}1 & \mathit{doUpdateR}(C, \alpha, \sigma) \mid ((\alpha_{\mathsf{m}} \perp) \in C) & = & E = \mathit{mkMono}(\mathit{True}, C, \sigma) \\
 & & & \text{return } (E \leftarrow (\alpha_{\mathsf{m}} = \sigma)) \\
\textsc{dr}2 & \mathit{doUpdateR}(C, \alpha, \sigma) \mid ((\alpha_\star \perp) \in C) & = & \text{return } (C \leftarrow (\alpha_\star = \sigma)) \\
\textsc{dr}3 & \mathit{doUpdateR}(C, \alpha, \sigma) \mid ((\alpha_\mu \geq \varsigma_a) \in C) & = & E = \mathit{subsCheck}(C, \varsigma_a, \sigma) \\
 & & & \text{return } (E \leftarrow (\alpha_\mu = \sigma)) \\
\textsc{dr}4 & \mathit{doUpdateR}(C, \alpha, \sigma) \mid ((\alpha_\mu = \sigma_a) \in C) & = & \mathit{eqCheck}(C, \sigma_a, \sigma) \\
 & & & \\
\textsc{uf}1 & \mathit{updateFlexi}(C, \alpha, [D] \Rightarrow \gamma) \mid \gamma \notin \mathit{dom}(D) & = & \mathit{updateRigid}(C, \alpha, \gamma) \\
\textsc{uf}2 & \mathit{updateFlexi}(C, \alpha, \varsigma) & = & \text{if } \alpha \in \widehat{C}(\varsigma) \text{ then } \mathit{fail} \text{ else } \mathit{doUpdateF}(C, \alpha, \varsigma) \\
\textsc{df}1 & \mathit{doUpdateF}(C, \alpha, [D] \Rightarrow \rho) \mid ((\alpha_{\mathsf{m}} \perp) \in C) & = & E = \mathit{mkMono}(\mathit{True}, C \bullet D, \rho) \\
 & & & \text{return } (E \leftarrow (\alpha_{\mathsf{m}} = \rho)) \\
\textsc{df}2 & \mathit{doUpdateF}(C, \alpha, \varsigma) \mid ((\alpha_\star \perp) \in C) & = & \text{return } (C \leftarrow (\alpha_\star \geq \varsigma)) \\
\textsc{df}3 & \mathit{doUpdateF}(C, \alpha, \varsigma) \mid ((\alpha_\mu = \sigma_a) \in C) & = & \mathit{subsCheck}(C, \varsigma, \sigma_a) \\
\textsc{df}4 & \mathit{doUpdateF}(C, \alpha, \varsigma) \mid ((\alpha_\mu \geq \varsigma_a) \in C) & = & E, \varsigma_r = \mathit{join}(C, \varsigma_a, \varsigma) \\
 & & & \text{return } (E \leftarrow (\alpha_\mu \geq \varsigma_r)) \\
 & \mathit{join} & : & \mathit{Constraint} \times \mathit{Scheme} \times \mathit{Scheme} \to \\
 & & & \qquad \mathit{Constraint} \times \mathit{Scheme} \\
\textsc{j}1 & \mathit{join}(C, [D] \Rightarrow \alpha, \varsigma_2) \mid (\alpha_\mu \perp) \in D & = & (C, \varsigma_2) \\
\textsc{j}2 & \mathit{join}(C, \varsigma_1, [D] \Rightarrow \alpha) \mid (\alpha_\mu \perp) \in D & = & (C, \varsigma_1) \\
\textsc{j}3 & \mathit{join}(C, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) & = & E = \mathit{eqCheck}(C \bullet D_1 \bullet D_2, \rho_1, \rho_2) \\
 & & & \overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(\mathit{dom}(C)) \\
 & & & \text{return } (E_{\widehat{E}(\mathit{dom}(C))}, [E_{\overline{\delta}}] \Rightarrow \rho_1)
\end{array}
$$

Figure 6.7: Unification and instance checking

Instance checking is given with the call $subsCheck(C, \varsigma, \sigma)$ that checks whether σ belongs in the interpretation of the scheme ς and produces an updated constraint, or fails. The function is defined by pattern matching on the type σ. If σ is a plain variable (S1) we simply update its bound with a call to *updateFlexi*. Otherwise (S2), we skolemize any top-level quantified variables the type may have, instantiate the scheme and check for equality. Finally, we are interested only in the resulting constraint that is reachable from the original one, $E_{\overline{\gamma}}$. The rest represents "dead variables". In a real implementation we actually do not have to perform any restriction on the constraint as the "dead part" is not reachable anyway. However we do have to check that $\overline{b} \# E_{\overline{\gamma}}$, which corresponds to an escape check condition.

The call to $updateRigid(C, \alpha, \sigma)$ updates the bound α so that it is equal to σ, and either fails or produces an updated constraint. Notice that we need to be careful for the case where σ is itself a single variable α. In this case σ may be actually equated through the constraint to α and such a condition *is solvable*! Therefore, the function *updateRigid* first ensures that the type σ does not reach the variable α, except only by a chain of constraints that only involve single variables. Clause UR1 makes sure we do not update the constraint if we are equating a variable with itself. Clauses UR2 and UR3 take care of the cases where the type σ is a single variable that is mapped to a single variable through the constraint, in which case we have to recursively call *updateRigid*. Finally, once clause UR4 is called, we are certain that there exist no single-variable chains, and hence we may perform an occur check that $\alpha \notin \widehat{C}(\sigma)$. If $\alpha \in \widehat{C}(\sigma)$ we fail, as no substitution can equate α and σ. On the other hand, if the occur check succeeds then we may call the core of the function, *doUpdateR*.

The call to $doUpdateR(C, \alpha, \sigma)$ performs the actual updating of the bound of α, assuming that α does not belong in the reachable variables of σ through C. If α is monomorphic and unbound in C (clause DR1) then we monomorphize σ, to ensure that all its reachable variables are now flagged as monomorphic, and update the bound of α. This is written in clause DR1 with $E \leftarrow (\alpha_{\mathsf{m}} = \sigma)$. If α is unrestricted and unbound in C (clause DR2) then we may directly update the constraint. If the variable is flexibly bound in C (clause DR3) we must ensure that the bound we want to assign to the variable is actually contained in the set of types allowed by the flexible bound of the variable. This is achieved with the call to *subsCheck*. Finally, if the variable is rigidly bound in C (clause DR4) we must check that its bound can be equated to the type σ. The call to $updateFlexi(C, \alpha, \varsigma)$ is

similar in operation to *updateRigid*. It updates the bound of α so that the new bound of α allows the intersection of its old bound and ς. But we need to be careful again to take care of single-variable chains. Rule UF1 does exactly this. Notice that because of well-formedness conditions on schemes we know that whenever we meet a scheme $[D]\Rightarrow\gamma$ where $\gamma \notin dom(D)$, it must be that γ is monomorphic and hence cannot represent a set of types, but can only be substituted for a monomorphic type. Hence, rule UF1 directly calls *updateRigid*. Rule UF2 performs the usual occur check and calls the core of the function, *doUpdateF*.

The call to $doUpdateF(C, \alpha, \varsigma)$ performs the actual updating of the bound of α, assuming that α does not belong in the reachable variables of ς through C. If α is monomorphic and unbound in C (clause DF1) then we instantiate the scheme ς and monomorphize it, and update the bound of α. If α is unrestricted and unbound in C (clause DF2) then we may directly update the constraint. If the variable is rigidly bound we check that its rigid bound is contained in the set of types denoted by ς with clause DF3). Finally, if the variable is flexibly bound we compute the *join* of its bound and ς, which should be interpreted as a scheme expressing the intersection of the sets denoted by the bound and ς.

Finally, a call to $join(C, \varsigma_1, \varsigma_2)$ is in the core of the algorithmic implementation. If it succeeds, it returns an updated constraint and a scheme expressing the intersection of (the interpretations of) ς_1 and ς_2. Because we wish to produce well-formed schemes we first have to check with clauses J1 and J2 the cases where one of the schemes is actually $[\alpha_\star \bot]\Rightarrow\alpha$. Otherwise clause J3 applies: we instantiate both argument schemes, check their bodies for equivalence, and return this part of the new constraint that is reachable through the original while we quantify over the rest in the returned scheme.

6.4 Summary of the algorithmic implementation properties

In this section we outline the most important properties of the algorithm; detailed metatheory can be found in Appendix A.

The first observation is that the functions that constitute our instance checking and unification terminate; and as an easy corollary the overall algorithm terminates.

Proposition (see Theorem A.1.20): Assume that $\vdash C_1$ and $\Delta_C \vdash \varsigma$ and $\Delta_C \vdash \sigma$. Then the call to $\mathit{subsCheck}(C_1, \varsigma, \sigma)$ either returns *fail* or terminates and returns a constraint C_2.

Unsurprisingly, the proof of termination is very similar to the termination proof for the ML^F unification algorithm. It uses a metric that is a lexicographic triple. The first component is the sum of the quantifier rank of the argument C_1 and of the scheme ς_1. Its second component is the sum of the sizes of ς and σ, and the third component involves the sizes of the sets of reachable variables of ς and σ through C_1.

The following proposition states soundness of instance checking and unification.

Proposition (see Section A.2): Assume in all cases that $\vdash C_1$ and all the schemes and types below are well-formed in Δ_{C_1}. Then:

1. If $\mathit{eqCheck}(C_1, \sigma_1, \sigma_2) = C_2$ and $\theta \models C_2$ then $\theta \models C_1$ and moreover $\theta\sigma_1 = \theta\sigma_2$.
2. If $\mathit{subsCheck}(C_1, \varsigma, \sigma) = C_2$ and $\theta \models C_2$ then $\theta \models C_1$ and moreover $\theta\sigma \in [\![\theta\varsigma]\!]$.
3. If $\mathit{updateRigid}(C_1, \alpha, \sigma) = C_2$ and $\theta \models C_2$ then $\theta \models C_1$ and moreover $\theta\alpha = \theta\sigma$.
4. If $\mathit{updateRigid}(C_1, \alpha, \varsigma) = C_2$ and $\theta \models C_2$ then $\theta \models C_1$ and moreover $\theta\alpha \in [\![\theta\varsigma]\!]$.
5. If $\mathit{join}(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$ and $\theta \models C_2$ then $\theta \models C_1$ and moreover $[\![\theta\varsigma]\!] \subseteq [\![\theta\varsigma_1]\!] \cap [\![\theta\varsigma_2]\!]$.

The following proposition states the completeness of unification with respect to the set semantics.

Proposition (see Section A.3): Assume in all cases that $\vdash C_1$ and all the schemes and types are well formed in Δ_{C_1} and that $\mathit{dom}(\theta) = \mathit{dom}(C_1)$. Then, the following are true:

1. If $\theta\sigma_1 = \theta\sigma_2$ then $\mathit{eqCheck}(C_1, \sigma_1, \sigma_2) = C_2$ and there exists a θ_r such that $\theta\theta_r \models C_2$.
2. If $\theta\sigma \in [\![\theta\varsigma]\!]$ then $\mathit{subsCheck}(C_1, \varsigma, \sigma) = C_2$ and there exists a θ_r such that $\theta\theta_r \models C_2$.
3. If $\theta\alpha = \theta\sigma$ then $\mathit{updateRigid}(C_1, \alpha, \sigma) = C_2$ and there exists a θ_r such that $\theta\theta_r \models C_2$.
4. If $\theta\alpha \in [\![\theta\varsigma]\!]$ then $\mathit{updateFlexi}(C_1, \alpha, \varsigma) = C_2$ and there exists a θ_r such that $\theta\theta_r \models C_2$.
5. If $\sigma \in [\![\theta\varsigma_1]\!] \cap [\![\theta\varsigma_2]\!]$ then $\mathit{join}(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$ and there exists a θ_r such that $\theta\theta_r \models C_2$ and $\sigma \in [\![\theta\theta_r\varsigma]\!]$.

It is particularly illuminating to observe the unification soundness and completeness for the *join* function. Soundness and completeness of unification show that *join* computes precisely a scheme that expresses the intersection of the interpretation of the argument schemes.

We now turn our focus to the main inference function. In order to state the soundness and completeness properties of the algorithm we have to define the notion of a boxy substitution.

Definition 6.4.1 (Boxy substitution)**.** Given a substitution θ from constrained variables to System F types (i.e. constrained-variable-free types), we define the boxy substitution of θ on σ, denoted with $\theta[\sigma]$, that substitutes the range of θ in a boxed and capture-avoiding fashion inside σ.

For example, if $\theta = [\alpha \mapsto \sigma]$ then $\theta[\alpha \to \alpha] = \boxed{\sigma} \to \boxed{\sigma}$. We may use boxy substitutions to recover specification types from algorithmic types, provided that all their constrained variables appear in the domains of the substitutions. Additionally we write $C \vdash \Gamma$ when for all $(x{:}\sigma) \in \Gamma$ it is the case that $\Delta_C \vdash \sigma$, and additionally it is the case that $\alpha \in \widehat{C}(\Gamma)$ iff $\Delta_C(\alpha) = \mathsf{m}$. Intuitively, the notation $C \vdash \Gamma$ means that all variables of Γ are in C and monomorphically flagged, and the reachable variables from Γ are the only m-flagged variables in C. We are now ready to state soundness:

Proposition (see Lemma A.4.10): If $\vdash C_1$ and $C_1 \vdash \Gamma$ and $\mathit{infer}(C_1, \Gamma, e) = C_2, \rho$ then for all $\theta \models C_2$ it is the case that $\theta \models C_1$ and $\theta\Gamma \vdash^{\mathsf{sd}} e : \theta[\rho]$.

Notice that we need not apply θ in a boxy fashion to Γ, since all the variables of Γ will be monomorphic in C_1, and θ can only map them to monomorphic types. A corollary is soundness with respect to the declarative specification:

Corollary 6.4.2. *If* $\vdash C_1$ *and* $C_1 \vdash \Gamma$ *and* $\mathit{infer}(C_1, \Gamma, e) = C_2, \rho$ *then for all* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $\theta\Gamma \vdash e : \theta[\rho]$.

We are now ready to state the main completeness proposition.

Proposition (see Lemma A.5.7): If $\vdash C_1$ and $C_1 \vdash \Gamma$ and $\theta \models C_1$ and $\mathit{dom}(\theta) = \mathit{dom}(C_1)$ and $\theta\Gamma \vdash^{\mathsf{sd}} e : \rho'$ then $\mathit{infer}(C_1, \Gamma, e) = C_2, \rho$ and there exists a θ_r such that $\theta\theta_r \models C_2$ and $\theta[\rho] \preceq\sqsubseteq \rho'$.

The proof is an induction on the size of e in the derivation of $\theta\Gamma \vdash^{\mathsf{sd}} e : \rho'$. The cases for `let`-bound expressions and abstractions rely on Lemma 5.3.5 and the Damas-Milner strengthening property (Corollary 5.3.18), and are similar to the corresponding cases for ordinary Hindley-Damas-Milner

inference. However the cases for applications and annotations crucially rely on the two propositions below:

Proposition (see Lemma A.5.2): If $\vdash \varsigma$ and $\sigma_1 \in [\![\varsigma]\!]$ and $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_2$ then $\sigma_2 \in [\![\varsigma]\!]$.

Proposition (See Lemma A.5.5): Assume that $\vdash C_1$ and $C_1 \vdash \Gamma$ and $C_1 \vdash \rho$ and $\theta \models C_1$ and $ftv(\rho) = \emptyset$ and $E, \varsigma = generalize(C_1, \Gamma, \rho)$ and $\overline{a} = ftv(\theta\rho) - ftv(\theta\Gamma)$. Then $\lfloor \forall \overline{a} \,.\, \theta[\rho] \rfloor \in [\![\theta\varsigma]\!]$.

Notice that the last proposition can be viewed as a generalization of Lemma 5.3.5 that involves schemes. The following corollary is then true, by observing that $\preceq\!\sqsubseteq$ is transitive:

Corollary 6.4.3. *If* $\vdash C_1$ *and* $C_1 \vdash \Gamma$ *and* $\theta \models C_1$ *and* $dom(\theta) = dom(C_1)$ *and* $\theta\Gamma \vdash e : \rho'$ *then* $infer(C_1, \Gamma, e) = C_2, \rho$ *and there exists a* θ_r *such that* $\theta\theta_r \models C_2$ *and* $\theta[\rho] \preceq\!\sqsubseteq \rho'$.

Together Corollary 6.4.2 and Corollary 6.4.3 ensure the implementability of the declarative specification of Figure 5.2.

6.5 Optimizations and complexity considerations

One observation about the reference implementation is the non-negligible amount of (somewhat complicated) book-keeping. For example, let us examine the *normalize* function in Figure 6.4. Clause N1 reads:

$$\text{N1} \quad normalize(C, \alpha) \mid (\alpha_\star \geq [D] \Rightarrow \rho) \in C \quad = \quad \text{return } ([C \bullet D - \alpha] \Rightarrow \rho)$$

Intuitively, the binding for α is no longer needed and hence we can remove it from the constraint $C \bullet D$. Consequently the returned constraint is $C \bullet D - \alpha$. But, would it hurt to not remove it? After all α can no longer be reached from any reachable variable of ρ.

Another instance of the bureaucracy is the *join* function in Figure 6.7. Clause J3 reads:

$$\begin{array}{lll} \text{J3} \quad join(C, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) & = & E = eqCheck(C \bullet D_1 \bullet D_2, \rho_1, \rho_2) \\ & & \overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(dom(C)) \\ & & \text{return } (E_{\widehat{E}(dom(C))}, [E_{\overline{\delta}}] \Rightarrow \rho_1) \end{array}$$

We first traverse ρ_1 through the constraint E to gather its variables. Then we get the reachable variables through E of the domain of C. Then we compute their difference $\overline{\delta}$. Finally, we *restrict* E to $\overline{\delta}$ to be the actual returned constraint, whereas we restrict again E to $\widehat{E}(dom(C))$ to return it. All this seems terribly complicated and bureaucratic—and furthermore such a naïve implementation of *join* is rather inefficient.

Fortunately we can do much better. Observe that schemes arise from quantification over constrained variables that do not appear in the environment via calls of *generalize*. Since the environment-reachable variables are always monomorphic, they are always flagged with m. What this means is that in a scheme we actually *never need* to record its constraint: its local constraint is implicitly determined by the part of the global constraint that can be spliced out by considering all the reachable variables from the type that are not m-flagged. For example, if we have inferred a type $\beta \to \alpha$ and a constraint $\{\alpha_{\mathsf{m}} = \tau, \beta_\star \geq \varsigma\}$ wen *know* how we can generalize the type $\beta \to \alpha$ to the scheme $[\beta_\star \geq \varsigma] \Rightarrow \beta \to \alpha$ without even having to look at which constrained variables occur in the environment and which not! The reason is that α appears flagged with m, and hence we know that it must be reachable from the typing environment and we cannot generalize over it.

This suggests a huge improvement. Since we always carry the global constraint, we no longer need to carry the local constraint of a scheme along with the scheme. The syntax of schemes can be simplified to:

$$\text{Schemes} \quad \varsigma \quad ::= \quad [\_] \Rightarrow \rho$$

without having to keep track of the precise constraint of the scheme in the scheme itself. Let us see how *normalize*, *join*, and *subsCheck* can be modified using this notation.

The corresponding clause of normalization is given below:

$$\text{N1} \quad \mathit{normalize}(C, \alpha) \mid (\alpha_\star \geq [\_] \Rightarrow \rho) \in C \quad = \quad \text{return } ([\_] \Rightarrow \rho)$$

Clause S2 of instance checking is given below:

$$\begin{array}{lll} \text{S2} \quad \mathit{subsCheck}(C, [\_]{\Rightarrow}\rho_1, \forall \overline{c}\,.\,\rho_2) & = & \rho_3 = [\overline{c \mapsto b}]\rho_2 \\ & & E = \mathit{eqCheck}(C, \rho_1, \rho_3) \\ & & \overline{\delta} = \{\delta \in \mathit{dom}(E) \mid \Delta_E(\delta) = \mathsf{m}\} \\ & & \text{if } \overline{b} \# E_{\overline{\delta}} \text{ then return } E \text{ else } \mathit{fail} \end{array}$$

There are two modifications compared to the original version of S2: The first modification is that now C contains the implicitly quantified variables of the scheme $[\_]{\Rightarrow}\rho_1$. Hence we only need to check that the skolem constants $\overline{b}$ did not escape in the part of the domain of E that *does not* contain the implicitly quantified variables of $[\_]{\Rightarrow}\rho_1$. But that part can only be the monomorphic part of E. The second modification simply returns E instead of restricting it to the reachable part of C as before. The reason is that the rest of constraint will be unreachable.[1]

The J3 clause of the *join* function can also be greatly simplified:

$$\begin{array}{lll} \text{J3} \quad \mathit{join}(C, [\_]{\Rightarrow}\rho_1, [\_]{\Rightarrow}\rho_2) & = & E = \mathit{eqCheck}(C, \rho_1, \rho_2) \\ & & \text{return } (E, [\_]{\Rightarrow}\rho_1) \end{array}$$

Since we do not restrict the returned constraint to eliminate its dead part, and we don't need to explicitly quantify over a constraint, this version of J3 needs not find the reachable variables of types or constraints, nor need it restrict the returned constraints at all.

It is worth noting that, in the light of these simplifications, *generalize* becomes merely a wrapper for *normalize*:

$$\begin{array}{lll} \mathit{generalize}(C, \Gamma, \rho) & = & \varsigma = \mathit{normalize}(C, \rho) \\ & & \text{return } (C, \varsigma) \end{array}$$

In conclusion, flagging the environment variables (and the reachable variables from the environment) as monomorphic results in eliminating many superfluous traversals of types through the constraints, and is the preferred way that a real-world implementation should be coded.

[1] This restriction is however technically useful in the proofs of type inference soundness and completeness; this is why the reference implementation includes it.

With respect to complexity, the termination metric of the unification/instance checking algorithm is cubic in the size of the constraints and types involved (Appendix A). This suggests a polynomial bound for unification—full type inference has to be worst-case exponential in the size of programs, as it subsumes ML type inference (which is exponential-time complete). Recent work by Rémy and Yakobowski [47, 49] gives evidence that the type inference algorithm in FPH can be improved to achieve almost linear complexity under reasonable assumptions (such as constant left-nesting of `let`-bindings).

6.6 Summary

In this chapter we have given the details of the algorithmic implementation of FPH, stated its formal properties, and presented possible optimizations. The implementation is more complicated than the traditional Hindley-Damas-Milner algorithm, but in exchange it enjoys a simple specification that only uses System F types. We have a Haskell implementation of a bidirectional version of the algorithm described in this chapter, along the lines of Section 7.1. The implementation is available from `www.cis.upenn.edu/~dimitriv/fph/` and consists of 1613 lines of Haskell code. In contrast, the simpler implementation of the bidirectional higher-rank type system consists of 977 lines of code—but the FPH implementation also supports user-defined datatypes contrary to the bidirectional higher-rank implementation and does not implement some of the simplifications in Section 6.5.

Chapter 7

Discussion of FPH

We proceed with a discussion of various FPH features and extensions. We show how to add bidirectionality, substantiating thus our claim that FPH is compatible with local type inference. We give some technical reasons that prevent FPH from using more expressive instance relations, such as the ones used in the predicative higher-rank type systems. These technical difficulties result in differences between the predicative higher-rank work of Chapter 3 and FPH. Finally, for the reader who has a good understanding of ML^F we sketch an encoding of FPH in a variant of ML^F that illustrates some of the more technical connections between the two works.

7.1 Bidirectionality

Bidirectional propagation of type annotations may further reduce the amount of required type annotations in FPH. It is relatively straightforward to add bidirectional annotation propagation to the specification of FPH in the style of Chapter 3. This bidirectional annotation propagation procedure can be implemented as a separate preprocessing pass, provided that we support open type annotations, and annotated λ-abstractions. Alternatively, this procedure can be implemented by weaving an inference judgement of the form $\Gamma \vdash^{\mathsf{sd}} e :\Uparrow \rho'$ and a checking judgement $\Gamma \vdash^{\mathsf{sd}} e :\Downarrow \rho'$. Because type annotation propagation depends on the shape of terms, this bidirectional system is

syntax-directed. A special-top level judgement $\Gamma \vdash^{\mathsf{sd}}_{\star} e :^{\Downarrow} \sigma'$ checks that the expression e can be assigned type σ' as follows:

$$\frac{\Gamma \vdash^{\mathsf{sd}} e :^{\Downarrow} \rho' \quad \overline{a} \# \Gamma}{\Gamma \vdash^{\mathsf{sd}}_{\star} e : \forall \overline{a} \,.\, \rho'} \textsc{skol} \qquad \frac{\Gamma \vdash^{\mathsf{sd}} e :^{\Uparrow} \rho' \quad \overline{a} \# \Gamma \quad \vdash^{\mathsf{F}} \lfloor \forall \overline{a} \,.\, \rho' \rfloor \leq \sigma}{\Gamma \vdash^{\mathsf{sd}}_{\star} e :^{\Downarrow} \boxed{\sigma}} \textsc{cbox}$$

The judgement $\Gamma \vdash^{\mathsf{sd}}_{\star} e :^{\Downarrow} \sigma'$ is defined by pattern matching on the structure of the expected type σ' that is pushed inwards. Rule SKOL simply removes the top-level quantifiers and checks the expression against the body of the type. Rule CBOX checks an expression against a single box. In this case, we must simply infer a type for the expression, as we cannot use its contents as a type annotation. Consequently, in our main checking judgement $\Gamma \vdash^{\mathsf{sd}} e :^{\Downarrow} \rho'$, we can always assume that ρ' is never a single box. The rule for checking applications then becomes:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{sd}} e_1 :^{\Uparrow} \sigma' \quad \sigma' \preceq \sqsubseteq^{\rightarrow} \sigma'_1 \to \sigma'_2 \\ \Gamma \vdash^{\mathsf{sd}}_{\star} e_2 :^{\Downarrow} \sigma'_1 \quad \vdash^{\mathsf{inst}} \sigma'_2 \leq \rho'_2 \quad \lfloor \rho'_2 \rfloor = \lfloor \rho' \rfloor\end{array}}{\Gamma \vdash^{\mathsf{sd}} e_1 \; e_2 :^{\Downarrow} \rho'} \textsc{app-check}$$

Following previous work, we first infer a type for the function e_1. Notice that e_2 is *checked* against the type σ'_1, which eliminates the need to generalize its type, as happens in the inference-only syntax-directed specification. Since ρ' cannot be a single box, we may simply instantiate σ'_2 to ρ'_2 and check that this type is equal modulo the boxes to the ρ' type that the context requires.

Annotations no longer reveal polymorphism locally, but rather propagate the annotation down the term structure. The rule ANN-INF below infers a type for an annotated expression $e :: \sigma$ by first *checking* e against the annotation σ:

$$\frac{\Gamma \vdash^{\mathsf{sd}}_{\star} e :^{\Downarrow} \sigma \quad \vdash^{\mathsf{inst}} \sigma \leq \rho'}{\Gamma \vdash^{\mathsf{sd}} (e :: \sigma) :^{\Uparrow} \rho'} \textsc{ann-inf}$$

The rule for inferring types for λ-abstractions is similar to rule SDABS, but the rule for *checking* λ-abstractions allows us now to check a function against a type of the form $\sigma'_1 \to \sigma'_2$:

$$\frac{\sigma'_1 \sqsubseteq \sigma_1 \quad \Gamma, (x{:}\sigma_1) \vdash^{\mathsf{sd}}_{\star} e :^{\Downarrow} \sigma'_2}{\Gamma \vdash^{\mathsf{sd}} \lambda x \,.\, e :^{\Downarrow} \sigma'_1 \to \sigma'_2} \textsc{abs-check}$$

Notice that σ_1' must be made box-free before entering the environment, to preserve our invariant that environments are box-free.

With these additions, and assuming that support for open type annotations, we can type functions with more elaborate types than simply $\tau \to \rho$ types, as the FPH original system does. Recall, for instance, Example 5.2.7 from Section 5.2.4.

```
f   :: forall a. a -> [a] -> Int
foo :: [Int -> forall b.b->b]


bog = f (\x y -> y) foo
```

Though `bog` is untypeable (even in a bidirectional system), we can recover it with the (ordinary) annotation:

```
bog = f (\x y -> y :: Int -> forall b. b -> b) foo
```

Special forms for annotated λ-abstractions (Section 5.2.4) are consequently not necessary in a bidirectional system.

7.2 Alternative design choices

The design choices in FPH are a compromise between simplicity and expressiveness. In this section we sketch several other ideas and their feasibility.

7.2.1 Removing the $\preceq\sqsubseteq$ relation

Our use of $\preceq\sqsubseteq$ in the type system of Figure 5.2 is motivated by examples where we need to extract and use some polymorphic value out of a data structure, as in `head ids 3`. If we are willing to sacrifice this convenience, the relation $\preceq\sqsubseteq$ need not be present in our specification, and rule

SUBS is unnecessary, at the cost of more type annotations. For instance, we may have to write `(head ids :: forall a. a -> a) 3`. The implementation becomes simpler as well. Nevertheless, we believe the extra complexity is worth it because it saves many annotations, and perhaps more importantly, it allows us to type all terms of System F that consist only of applications and variables (Section 5.2.5) without need for any type annotations.

7.2.2 Typing abstractions with more expressive types

Recall that λ-abstractions are typed with box-free types only. This implies that certain transformations, such as *thunking*, may break typeability. For example, consider the following code:

```
f1 :: forall a. (a -> a) -> [a] -> Int
g1 = f‘ (choose id) ids   -- OK

f2 :: forall a b. (b -> a -> a) -> [a] -> Int
g2 = f2 (\ _ -> choose id) ids -- fails!
```

In the example, while `g1` type checks, simply thunking the application `choose id` breaks typeability, because the type $\boxed{\forall a\,.\,a \to a} \to \boxed{\forall a\,.\,a \to a}$ cannot be unboxed.

An obvious alternative would be to allow arbitrary ρ' types as results of λ-abstractions, and lift our invariant that environments are box-free to allow τ' types as the arguments of abstractions. Though such a modification allows for even fewer type annotations (the bodies of abstractions could use impredicative instantiations and no annotations would be necessary), we do not know of a sound and complete algorithm that could implement such an extension, for at least two reasons.

1. One complication has to do with boxy instantiation $\preceq$. If an argument x enters the environment with type $\boxed{\sigma}$, then in the body of the abstraction boxy instantiation may allow x to be used at two different types—something that algorithmically cannot be implemented. For example

we may have the following:

$$\frac{(g{:}\boxed{\forall a\,.\,a \to a}) \vdash g\ 3 : \texttt{Int} \quad (g{:}\boxed{\forall a\,.\,a \to a}) \vdash g\ \texttt{True} : \texttt{Bool}}{\vdash \lambda g\,.\,(g\ 3, g\ \texttt{True}) : \boxed{\forall a\,.\,a \to a} \to (\texttt{Int}, \texttt{Bool})}$$

where both the premises are derivable through uses of SUBS with boxy instantiation. For instance:

$$\frac{\begin{array}{c}(g{:}\boxed{\forall a\,.\,a \to a}) \vdash g : \boxed{\forall a\,.\,a \to a} \\ \boxed{\forall a\,.\,a \to a} \preceq\sqsubseteq \texttt{Int} \to \texttt{Int}\end{array}}{(g{:}\boxed{\forall a\,.\,a \to a}) \vdash g : \texttt{Int} \to \texttt{Int}}$$

This is bad because it threatens implementability. Imagine the following program fragment:

```
h = (\lambda x @.@ 42)\;(\lambda g @.@ (g\;3, g\;@True@))
```

How is the type checker going to figure out the type of the higher-ranked function? Actually, as the proof of Wells indicates [61], the presence of guessed polymorphic function arguments in the environment that can be arbitrarily instantiated leads to undecidability of type reconstruction. Perhaps an alternative would be to introduce a different type of *rigid boxes* that do not admit $\preceq$, but only $\sqsubseteq$.

2. A second complication has to do with uses of boxy instantiation for the body types of abstractions. If a λ-abstraction is typed with $\tau \to \boxed{\forall a\,.\,\rho}$ then by using $\preceq$ when typing the body of the abstraction it can also be typed with $\tau \to \boxed{\rho}$. Algorithmically, this means that probably we would have to "flexify" all bounds to the right of arrows. If for a function we have inferred type $\texttt{Int} \to \alpha$ where $\alpha = \forall a\,.\,\rho$, the algorithm would probably have to modify this type to $\texttt{Int} \to \alpha$ where $\alpha \geq [\alpha_\star\ \bot] \Rightarrow \rho$, in order to capture the possibility of $\preceq$ used in the typing derivation of the body of the abstraction. Currently we have not fully studied the consequences of this modification.

In general, it is not immediate that the programming benefits justify the implementation, specification, and formal verification costs—after all in many cases λ-abstractions are `let`-bound, and hence they are forced by the rules for `let`-bound expressions to be box free anyway.

7.2.3 A box-free specification

A safe approximation of where type annotations are necessary is at `let`-bindings or λ-abstractions that have to use rich types. Perhaps surprisingly, taking this guideline one step further, if we *always* require annotations in bindings with rich types then we no longer need boxes in the specification *at all*. Consider the basic type system of Figure 5.2 with the following modifications:

1. Drop all boxy structure from all typing rules, that is, replace all ρ', σ', types with ρ and σ types, and completely remove SUBS and $\preceq\sqsubseteq$. Instantiate with arbitrary σ types in rule INST.
2. Replace rule LET and ABS with their corresponding versions for Damas-Milner types

$$\frac{\Gamma \vdash u : \forall \overline{a}\,.\,\tau \qquad \Gamma, (x{:}\forall \overline{a}\,.\,\tau) \vdash e : \rho}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \rho}\ \text{LET} \qquad \frac{\Gamma, (x{:}\tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x\,.\,e : \tau_1 \to \tau_2}\ \text{ABS}$$

3. Add provision for annotated `let`-bindings and λ-abstractions:

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, (x{:}\sigma) \vdash e : \rho}{\Gamma \vdash \texttt{let}\ x :: \sigma = u\ \texttt{in}\ e : \rho}\ \text{LET-ANN} \qquad \frac{\Gamma, (x{:}\sigma_1) \vdash e : \sigma_2}{\Gamma \vdash (\lambda x\,.\,e ::: \sigma_1 \to \sigma_2) : \sigma_1 \to \sigma_2}\ \text{ABS-ANN}$$

The resulting type system enjoys sound and complete type inference, by using essentially the same algorithm as the FPH type system. However, this variation is more demanding in type annotations than the box-based FPH. For instance, one has to annotate *every* `let`-binding that uses rich types, even if its type did not involve any impredicative instantiations. For example:

```
f :: Int -> (forall a. a -> a) -> (forall a. a -> a)
h = f 42      -- fails!
```

The binding for `h` has a rich type an hence must be annotated, although no impredicative instantiation took place.

Of course the algorithm of FPH will be essentially the same, even in this box-free specification. There is a certain gain in simplicity in this approach: No boxes are needed anywhere and programmers need not understand boxy instantiation $\preceq$, or protected unboxing $\sqsubseteq$.

On the other hand, this simplification is more demanding in type annotations. Many programs that GHC programmers are used to writing without type annotations (such as the last example) and that perfectly type check in the type systems of Chapter 3 are now untypeable without annotations. Hence, we believe that the box-free specification is less suitable for a real-world implementation.

7.2.4 Removing the strip functions

Finally, recall that the rule for typing applications makes use of a strip function that returns a non-boxed System F type from a boxy type by discarding all boxes. Alternatively, we could have extended the $\sqsubseteq$ relation to also *expand* boxes arbitrarily around types. Hence, the function type $\sigma'_1 \to \sigma'_2$ and the argument type σ'_3 could be converted along $\sqsubseteq$ to $\sigma'_4 \to \sigma'_2$ and σ'_4 respectively. Consequently, the condition $\lfloor \sigma'_1 \rfloor = \lfloor \sigma'_3 \rfloor$ could be eliminated, and we would not need the strip function in the typing rules. This change however would be mostly esthetic and we are not certain that it would make the system easier to use.

7.3 On η-conversions and deep instance relations

The FPH system does not preserve typeability under η-expansions, contrary to System F and ML^F. In particular, if the binding $f : \sigma \to \texttt{Int}$ exists in the environment then it is not necessarily the case that $\lambda x\,.\,f\ x$ is typeable—since x can only be assigned a τ-type.

Unsurprisingly, since FPH is based on System F, it is not stable under η-reductions. If an expression $\lambda x\,.\,e\ x$ makes a context $\mathcal{C}[(\lambda x\,.\,e\ x)]$ typeable, then it is not necessarily the case that $\mathcal{C}[e]$ is typeable. Consider the code below:

```
f   :: Int -> forall a. a -> a
g   :: forall a. a -> [a] -> a
lst :: [forall a. Int -> a -> a]


g1 = g (\x -> f x) lst   -- ok
```

```
g2 = g f lst              -- fails!
```

The application in `g2` (untypeable in the implicitly typed System F) is ill-typed since `lst` requires the instantiation of `g` with type $\forall a\,.\,\texttt{Int} \to a \to a$, whereas `f` has type $\texttt{Int} \to \forall a\,.\,a \to a$. The FPH system, which is based on System F, is not powerful enough to understand that these two types are isomorphic.

Although such conversions are easier to support in predicative systems [41], the presence of impredicativity complicates our ability to support them. For example, our *join* function would have to be extended to compute joins between $\forall a\,.\,\texttt{Int} \to a \to a$ and $\texttt{Int} \to \forall a\,.\,a \to a$. We have to modify our interpretation of schemes and our instance checking algorithm.

We are additionally not sure how such conversions may cross the boundaries of boxes that represent unknown perhaps polymorphic instantiations. For example, a conversion like deep skolemization should not cross the boundaries of boxes. While it may well be that $\vdash \forall a\,.\,\texttt{Int} \to a \to a \leq \texttt{Int} \to \forall b\,.\,b \to b$, it should not be derivable that $\vdash \forall a\,.\,\texttt{Int} \to a \to a \leq \texttt{Int} \to \boxed{\forall b\,.\,b \to b}$. To see why consider the code below:

```
f :: forall c. (Int -> c) -> [c] -> Bool


h = f (\k x -> x) ids
```

Now, the type checker has to check that the inferred type of `\k x -> x`, presumably $\forall a\,.\,\texttt{Int} \to a \to a$ (without eager generalization), is more general than $\texttt{Int} \to \gamma$ where γ is the constrained variable created by the instantiation of c. Of course at this point the type checker has no way of determining that $\gamma = \forall b\,.\,b \to b$ but only that $\gamma = \beta \to \beta$, which will cause the implementation to fail when it will check the second argument `ids`.

Another point is that deep instance relations—even without deep skolemization—give rise to constraints that are "going the opposite way" from our instance constraints. For example

$$\forall c\,.\,\texttt{Int} \to c \leq \texttt{Int} \to \forall b\,.\,b \to b$$

gives rise to a constraint $\gamma \leq \forall b\,.\,b \to b$. How can we deal with such constraints? Is decidability threatened? Should we replace them perhaps with equalities? If yes, what is then a clean specification of this algorithmic operation? Otherwise, our unification algorithm has to be extended with bounds of this form and we need to assign interpretation to such constraints.

In fact, no type inference system with impredicative instantiations proposed to date fully supports η-reductions because those proposals are based on System F. On the other hand, ML^F does actually support η-expansions, since all that is required to perform an η-expansion is the ability to abstract over a term variable that is only going to be used at a single (perhaps polymorphic) type. Since ML^F requires an annotations only if an argument variable are used at two or more types, no annotations are needed, and typeability is preserved under an η-expansion.

We are currently seeking ways to extend our instance relation to some "deep" version that treats quantifiers to the right of arrows as if they were top-level, but combining that with impredicative instantiations remains a subject of future research.

7.3.1 A comparative discussion of the higher-rank type systems and FPH

In what follows we give a comparison of the behaviour of the syntax-directed higher-rank type system from Chapter 3, Figure 3.1 (we use $\vdash_{\mathsf{sd}}$ and $\vdash^{\mathrm{poly}}_{\mathsf{sd}}$ for that typing relation), the bidirectional system from Chapter 3, Figure 3.3 (we use $\vdash_{\Uparrow}$, $\vdash^{\mathrm{poly}}_{\Uparrow}$, and $\vdash_{\Downarrow}$, $\vdash^{\mathrm{poly}}_{\Downarrow}$ for that typing relation), and the uni-directional version of FPH from Chapter 5, Figure 5.2 (we use $\vdash^{\mathsf{FPH}}$ for that typing relation). We compare the various type systems with respect to stability under η-conversions and applications of polymorphic combinators.

Stability under η-expansions None of the relations $\vdash_{\Uparrow}$, $\vdash^{\mathsf{FPH}}$, or $\vdash_{\mathsf{sd}}$ preserve typeability under an η-expansion. We already gave the reason: an η-expansion may require that an argument gets pushed into the environment with a polymorphic type. For a more subtle reason, the property is *also* false for the checking relation $\vdash_{\Downarrow}$. To see why, assume that $\Gamma \vdash_{\Downarrow} e : \sigma_1 \to \sigma_2$. We would like to check whether $\Gamma \vdash_{\Downarrow} \lambda x\,.\,e\ x : \sigma_1 \to \sigma_2$. Using rule ABS2, it would be adequate to show that $\Gamma, (x{:}\sigma_1) \vdash^{\mathrm{poly}}_{\Downarrow} e\ x : \sigma_2$. Assume that $\forall \overline{c}\,.\,\rho_2 = pr(\sigma)$. Hence, it is would

be enough to show that $\Gamma, (x{:}\sigma_1) \vdash_{\Downarrow} e\ x : \rho_2$. To show this we would have to show that we can *infer* a type for e, since rule APP requires that in application nodes we infer the function type while we check the argument type. But—and here is where the proof breaks—the fact that $\Gamma \vdash_{\Downarrow} e : \sigma_1 \to \sigma_2$ *does not* imply that there exists a type σ such that $\Gamma \vdash_{\Uparrow} e : \sigma$. It may well be the case that e is typeable only in checking mode.

Stability under η-reductions The higher-rank type systems actually preserve typeability of expressions under η-expansions in inference mode, as the next two lemmas show.

Lemma 7.3.1. *If* $\Gamma \vdash^{\text{poly}}_{\mathsf{sd}} \lambda x\,.\,e\ x : \sigma_1$ *then there exists a* σ_2 *with* $\Gamma \vdash^{\text{poly}}_{\mathsf{sd}} e : \sigma_2$ *and* $\vdash^{\mathsf{dsk}} \sigma_2 \leq \sigma_1$.

Proof. By inversion it must be that $\sigma_1 = \forall \overline{a}\,.\,\rho_1$ with $\overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma)$ and $\Gamma \vdash_{\mathsf{sd}} \lambda x\,.\,e\ x : \rho_1$. By inversion again it must be that $\rho_1 = \tau_1 \to \rho_2$ such that $\Gamma, (x{:}\tau_1) \vdash_{\mathsf{sd}} e\ x : \rho_2$. Hence, it must be that $\Gamma \vdash_{\mathsf{sd}} e : \tau_1 \to \sigma_2$ such that $\vdash^{\mathsf{inst}} \sigma_2 \leq \rho_2$ (with an intermediate inversion on the $\vdash^{\mathsf{dsk}}$ relation). Then it is straightforward to show that $\vdash^{\mathsf{dsk}} \forall \overline{b}\,.\,\tau_1 \to \sigma_2 \leq \forall \overline{a}\,.\,\tau_1 \to \rho_2$, where $\overline{b} = \mathit{ftv}(\tau_1 \to \sigma_2) - \mathit{ftv}(\Gamma)$. □

Lemma 7.3.2. *If* $\Gamma \vdash^{\text{poly}}_{\Uparrow} \lambda x\,.\,e\ x : \sigma_1$ *then there exists a* σ_2 *with* $\Gamma \vdash^{\text{poly}}_{\Uparrow} e : \sigma_2$ *and* $\vdash^{\mathsf{dsk}} \sigma_2 \leq \sigma_1$.

Proof. The proof follows the same structure as the proof of Lemma 7.3.1. □

Moreover, typeability is preserved in checking mode.

Lemma 7.3.3. *If* $\Gamma \vdash^{\text{poly}}_{\Downarrow} \lambda x\,.\,e\ x : \sigma_1$ *then* $\Gamma \vdash^{\text{poly}}_{\Downarrow} e : \sigma_1$.

Proof. The proof follows the same structure as the proof of Lemma 7.3.1 and uses Theorem 3.4.10 and 3.4.16. □

The FPH type system is not stable under η-reduction on the other hand, as the example at the beginning of this section demonstrates. The reduct is well typed, but maybe with a type that is not related according to the System F type instance to the original one.

Stability under polymorphic application combinators Since the FPH type system is based on System F, if $\Gamma \vdash^{\mathsf{FPH}} e_1\ e_2 : \sigma'$ then also $\Gamma \vdash^{\mathsf{FPH}} \mathtt{app}\ e_1\ e_2 : \sigma'_1$ such that $\lfloor \sigma' \rfloor = \lfloor \sigma'_1 \rfloor$.

But in any of the predicative higher-rank type systems, the application of the `app` combinator may require an impredicative instantiation, and hence typeability is not, in general, preserved under such applications.

7.4 On encoding FPH in an ML^F-like specification

Because of the many technical parallels between FPH and ML^F in the specification and the implementation one may wonder whether an encoding of FPH in a variation of ML^F is possible. It turns out that such an encoding is possible, but requires certain (non-invasive) modifications in the ML^F type system. The first has to do with restricting ML^F equivalence following Leijen [29] because ML^F type equivalence is larger than System F equivalence. In particular, it allows reordering of quantifiers arbitrarily deep in the type structure. For instance, in ML^F the types $\forall(a = \forall bc\,.\,b \to c \to b)\,.\,[a]$ and $\forall(a = \forall cb\,.\,b \to c \to b)\,.\,[a]$ are equivalent, whereas in System F and FPH the types $[\forall bc\,.\,b \to c \to b]$ and $[\forall cb\,.\,b \to c \to b]$ are not equal.

The second modification is to replace all "known" polymorphism with special "equivalence bounds" $\equiv$ that admit both sharing and unsharing. For example, if a variable in the environment has type $\sigma \to \mathtt{Int}$, its type becomes $\forall(a \equiv \sigma)\,.\,a \to \mathtt{Int}$. Since these bounds admit sharing and unsharing, it does not matter whether these bounds are "maximally" or "minimally" shared. What we mean by this is that, if a function has type $\sigma \to \sigma \to \mathtt{Int}$ in the environment, it does not matter whether we encode it as $\forall(a \equiv \sigma)\,.\,a \to a \to \mathtt{Int}$ or $\forall(a \equiv \sigma, b \equiv \sigma)\,.\,a \to b \to \mathtt{Int}$. ML^F instance now allows conversions of flexible bounds to $=$-bounds, but *never* to $\equiv$-bounds, because the latter are known polymorphism. On the other hand, an equivalence bound $\equiv$ can be converted through the ML^F abstraction relation to a rigid bound (which no longer admits ML^F unsharing). But not conversely. In fact revelation (the inverse of abstraction, which will be triggered at type annotation nodes) will convert rigid bounds to equivalence bounds, which will subsequently allow any sharing and unsharing (because they represent known information from some programmer-supplied user annotation). In a type annotation node, a type with perhaps rigid bounds meets a *programmer supplied type* that contains *only equivalence bounds* (i.e. all polymorphism is known). Hence, all the guessed $=$-bounds are converted to use $\equiv$-bounds.

Then our boxes correspond to ML^F rigid bounds, in a specification which completely lacks flexible bounds; since we never need to quantify over flexible types, such types need not appear in the specification.

Usages of boxy instantiation (rule BI) correspond to the ML^F equivalence $(Q)\ \forall(a = \sigma)\,.\,a \equiv \sigma$ which must of course be extended to $(Q)\ \forall(a \equiv \sigma)\,.\,a \equiv \sigma$. For example here is how to type `head ids` with type `Int` $\to$ `Int` in this hypothetical version of ML^F:

$$\dfrac{\dfrac{\dfrac{(\emptyset)\ \Gamma \vdash \texttt{ids} : \forall(b \equiv \sigma_{id})\,.\,[b]}{\forall(b \equiv \sigma_{id})\,.\,[b] \sqsubseteq \forall(b = \sigma_{id})\,.\,[b]}}{(\emptyset)\ \Gamma \vdash \texttt{ids} : \forall(b = \sigma_{id})\,.\,[b]} \qquad \dfrac{\ldots}{(b = \sigma_{id})\ \Gamma \vdash \texttt{head} : [b] \to b}}{(b = \sigma_{id})\ \Gamma \vdash \texttt{head ids} : b}$$

Now we may continue the derivation as follows:

$$\dfrac{\dfrac{(b = \sigma_{id})\ \Gamma \vdash \texttt{head ids} : b}{(\emptyset)\ \Gamma \vdash \texttt{head ids} : \forall(b = \sigma_{id})\,.\,b} \qquad (\emptyset)\ \forall(b = \sigma_{id})\,.\,b \equiv \sigma_{id}}{(\emptyset)\ \Gamma \vdash \texttt{head ids} : \forall(a \geq \bot)\,.\,a \to a}$$
$$(\emptyset)\ \Gamma \vdash \texttt{head ids} : \texttt{Int} \to \texttt{Int}$$

Notice that the usage of $\equiv$ in this last derivation corresponds precisely to boxy instantiation in FPH.

In this modified version of ML^F, the corresponding rule for `let`-bindings will simply require that the type we attribute to the expression to be `let`-bound does not contain any guessed polymorphism, that is, it does not contain any rigid bounds—but may well contain $\equiv$-bounds.

$$\dfrac{(Q)\ \Gamma \vdash u : \sigma \quad \sigma \text{ only contains } \equiv\text{-bounds} \quad (Q)\ \Gamma, (x{:}\sigma) \vdash e : \sigma_1}{(Q)\ \Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e}\ \text{LET}$$

What we require then with this rule is that all the $=$ bounds in σ be inlined, which effectively forces them to have been monomorphic, *precisely* as our $\sqsubseteq$ relation strips the boxes when their contents are monomorphic.

The ideas behind this encoding, which essentially replaces boxes with named rigid constraints and known polymorphism with named $\equiv$-bounds, suggest several improvements over FPH:

- They possibly allow the lifting the restriction that argument variables enter environments with only monomorphic types. Now they can enter environments with monomorphic types that contain variables with rigid bounds, and hence annotations will be required only if these arguments have to be used at two different types.
- They allow the lifting of the restriction that the returned types of abstractions have to be box-free.
- The encoding opens a possibility of directly using the new and efficient ML^F unification algorithm [47] and its metatheory. Flexible bounds will still exist internally, as in FPH or (the original) ML^F.
- The fact that abstraction converts $\equiv$-bounds to $=$-bounds suggests that the instance relation in FPH could be extended so that $\sqsubseteq$ is allowed to also expand boxes around known polymorphism, as outlined in Section 7.2.4. Conversion of a $\equiv$-bound to a $=$-bound precisely corresponds to expanding a box around the $\equiv$-bound.
- If we allow quantification in the environment over $=$-bounds as well (instead of only over $\equiv$-bounds) then even fewer type annotations may be needed in `let`-bound expressions. For example there should be no annotation on `let` $h =$ `id ids in` $\ldots$ as FPH currently requires, because h can be assigned type $\forall(a = [\forall b\,.\,b \to b])\,.\,a$ in the environment.

In exchange for these simplifications, one must enter the world of named boxes and rigid or equivalence bounds, which may or may not be desirable. Additionally one must keep in mind that, in the absence of flexible bounds in the specification, even the ML^F-based system will not enjoy the `let` expansion property, contrary to full ML^F with flexible bounds.

Chapter 8

Closely connected works

There are several proposals for higher-rank and impredicative polymorphism of varying expressiveness, complexity, and practicality. The most relevant related research for this dissertation can be subject to a coarse classification: works that are based on explicitly marking the introduction and elimination of polymorphism, works that are based on local type inference, and works that are based on global type inference.

8.1 Marking polymorphism

There is some work on explicitly marking the introduction and elimination of polymorphism. If this approach is taken, the algorithmic implementation is usually simple with the cost of many annotations on programs.

Semi-explicit first-class polymorphism Garrigue and Rémy's extension of ML with higher-rank polymorphism [6] embeds polytypes inside monotypes, like the boxy monotypes of FPH. Their types mark whether polytypes are annotated or inferred. Other than type signatures, that system does not use any contextual information arising from annotation propagation. Only annotated polytypes are allowed to be instantiated, at only marked locations, and all polymorphic information

has to be explicitly revealed. In contrast, in FPH polymorphic information can be implicitly revealed without any special term-level constructs as long as this does not lead to ambiguities (e.g. when the type in or out of a box is an ordinary Damas-Milner type).

First-class polymorphism via type constructors Mark Jones proposed FCP [16], standing for *first-class polymorphism with type inference*, which is a system that allows polymorphic values to be packaged as values of datatypes with higher-rank constructors. Pattern matching against a value of such a constructor reveals the polymorphic contents and amounts to elimination of polymorphism, whereas packaging up a polymorphic value via applying it to such a constructor amounts to introducing polymorphism. Hence, a modest type inference system with predicative instantiations that allows constructors (constants) with higher-rank types is all that is required, provided that programmers are ready to perform manually all coercions for polymorphic values that they want to use as first-class.

8.2 The local type inference approach

Local type inference [45] suggests that we *synthesize* the type instantiation of a polymorphic expression locally, for example by looking at the application of a polymorphic function to an argument, and determining what type the argument requires the function to be instantiated with. Since the Damas-Milner type system supports a more global type inference approach, combining local type inference for polymorphic instantiations with global type inference for ordinary monomorphic instantiations is somewhat delicate to implement and specify.

Boxy Types Like Colored Local Type Inference [40], Boxy Types *combine* the two judgement forms of bidirectional type inference, $\vdash_{\Downarrow}$ and $\vdash_{\Uparrow}$ into a single judgment, $\Gamma \vdash e : \sigma'$, where σ' is like an ordinary type, except that it may contain "holes" that correspond to the locally inferred part of the type. These "holes" are written with boxes—but these boxes do not only identify impredicative instantiations as in FPH. For example, $\Gamma \vdash e : \boxed{\texttt{Bool}} \rightarrow \texttt{Int}$ checks that e has a type of the

form $\ldots \to \texttt{Int}$; and infers that this unknown argument type is `Bool`. The inference and checking judgements of bidirectional type inference become now just special cases:

$$\Gamma \vdash_{\Downarrow} e : \sigma \quad \text{is written} \quad \Gamma \vdash e : \sigma$$
$$\Gamma \vdash_{\Uparrow} e : \sigma \quad \text{is written} \quad \Gamma \vdash e : \boxed{\sigma}$$

The purpose of the boxes in the specification is to make sure that all derivations in the Boxy Types system do have a corresponding algorithmic interpretation (i.e. boxes ensure *implementability*).

To demonstrate the Boxy Types approach to higher-rank types, let us examine several examples. The judgement

$$\vdash \lambda g\,.\,(g\ 3, g\ \texttt{True}) : (\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$$

is derivable in the Boxy Types system, as the interpretation of the box-free type is that the abstraction is *checked*, and hence the type of the argument g is known. Similarly, it is derivable that $\vdash \lambda g\,.\,(g\ 3, g\ \texttt{True}) : (\forall a\,.\,a \to a) \to \boxed{(\texttt{Int}, \texttt{Bool})}$, as the type indicates that the argument g has a known type but we are inferring the resulting type. On the contrary $\nvdash \lambda g\,.\,(g\ 3, g\ \texttt{True}) : \boxed{\forall a\,.\,a \to a} \to \boxed{(\texttt{Int}, \texttt{Bool})}$. The reason is that, algorithmically, such a typing derivation for this λ-abstraction would be impossible to implement because we would have to guess the type of the polymorphic argument g—and there are many types that would result in the abstraction being typeable. Similarly it is $\nvdash \lambda x\,.\,x : \boxed{\forall a\,.\,a \to a} \to \boxed{\forall a\,.\,a \to a}$ but only $\vdash \lambda x\,.\,x : \boxed{\tau} \to \boxed{\tau}$. In other words, when no contextual information is available, the boxes can only possess monomorphic contents. Of course, type annotations augment the contextual information to determine the contents of boxes, and hence it is derivable that:

$$\vdash \lambda(g :: \forall a\,.\,a \to a)\,.\,(g\ 3, g\ \texttt{True}) : \boxed{\forall a\,.\,a \to a} \to \boxed{(\texttt{Int}, \texttt{Bool})}$$

For impredicative instantiation of type variables, the story is very similar; instantiations of type variables are boxes which are filled in locally by annotations or contextual information. For example, if f has type $\forall a\,.\,a \to a$, then it is derivable that: $\vdash f : (\forall a\,.\,a \to a) \to (\forall a\,.\,a \to a)$ but $\nvdash f : \boxed{\forall a\,.\,a \to a} \to \boxed{\forall a\,.\,a \to a}$. Instead, only $\vdash f : \boxed{\tau} \to \boxed{\tau}$ is derivable, as there is no local information available to determine the instantiation of f.

The particular contribution of Boxy Types is a type system that supports type annotations and higher-rank and impredicative types, that include the aforementioned notion of a boxy type, which expresses the direction of information flow in the type inference algorithm in the type system specification. The Boxy Types system has been shown a conservative extension of the Damas-Milner type system, expressive enough to encapsulate all of System F via the addition of type annotations to System F programs, and to admit a sound and complete type inference algorithm which is an extension of the Hindley-Damas-Milner algorithm—and (relatively) simple to implement in a real-world compiler such as GHC.

On the other hand, the local decisions that Boxy Types must make have a price. In particular, Boxy Types often require programs to unbox the contents of the boxes too early. For type inference completeness, if information about the contents of a box is not available locally (i.e. cannot be discovered within the premises of the typing rule that applies), that box must contain a monomorphic type. As a result, the vanilla Boxy Types system requires many type annotations. Ad-hoc heuristics, such as treating function applications by considering *all* arguments simultaneously (n-ary applications) and elaborate type subsumption procedures relieve the annotation burden but further complicate the specification and the predictability of the system.

Because Boxy Types discover polymorphism locally, programs like `head (cons id ids)` are not typeable in Boxy Types, whereas they are typeable in FPH. On the other hand, because Boxy Types uses a type instance relation that is co-variant in function result types, some programs may be typeable in Boxy Types that are not typeable in FPH. For instance:

```
g :: Int -> forall a. a -> a
f :: (forall a. Int -> a -> a) -> Int

bar = f g
```

Despite these differences, we believe that the simpler specification of FPH (Chapter 5) is a dramatic improvement.

HM^F Leijen's HM^F system [31] is another interesting point in the design space. The HM^F system enjoys a particularly simple inference algorithm (a variant of Algorithm W), and one that is certainly simpler than FPH. In exchange, the typing rules are somewhat unconventional in form, and it is somewhat harder to predict exactly where a type annotation is required and where none is needed.

The key feature of HM^F is a clever application rule, where impredicative instantiations are determined by a local match procedure. In the type system, this approach imposes two "minimality" conditions: First, all types entering the environment are the most general types that can be assigned to programs. For example in the definition `let` $x = \lambda y\,.\,y$ `in`..., x can *only* enter the environment with type $\forall a\,.\,a \to a$ and *not* with type $\forall b\,.\,[b] \to [b]$, which would also be allowed by Damas-Milner. Second, all allowed instantiations are those that "minimize" polymorphism. For example, assume that $(f{:}\forall a\,.\,a \to [a]) \in \Gamma$. Then, while HM^F allows the derivation: $\Gamma \vdash f : a \to [a]$ and $\Gamma \vdash f : \sigma \to [\sigma]$ for any polymorphic σ, it *does not* allow the derivation $\Gamma \vdash f$ `id` $: [\forall a\,.\,a \to a]$ despite the fact that $\Gamma \vdash$ `id` $: \forall a\,.\,a \to a$. The reason is because the type $[\forall a\,.\,a \to a]$ is not the type with the minimum number of quantifiers for f `id`; that type (and the one that Damas-Milner would assign to f `id`) is $[a \to a]$.

The local match procedure means that HM^F makes eager decisions: in general, polymorphic functions get instantiated by default, unless specified otherwise by the programmer. For example, the program `single id` (where `single` has type $\forall a\,.\,a \to [a]$) cannot be typed with type $[\forall a\,.\,a \to a]$. The top-level quantifiers of `id` are instantiated too early, before the local match procedure. Because FPH delays instantiations using constraints, we may type this expression with $[\boxed{\forall a\,.\,a \to a}]$ (but we would still need an annotation to `let`-bind it). In HM^F one may annotate the function `single`, or specify with a *rigid type annotation* that the type of `id` must not be instantiated: `(single (id :: forall a. a -> a))`. Note that HM^F annotations are different than the annotations found, for instance, in Haskell—e.g. `(id :: forall a. a -> a) 42` is rejected.

Leijen observes that local match procedures are, in general, not robust to program transformations. If only a local match were to be used, the application `(cons id) ids` would not be typeable, while `(revcons ids) id` would be (where `revcons` has type $\forall a\,.[a] \to a \to [a]$). Hence, these problems are circumvented in HM^F by using an n-ary application typing rule that uses type information from *all*

arguments in an application. The same idea appears in the local type inference original paper [45].

In general, annotations are needed in HM^F on λ-abstractions with rich types and on arguments that must be kept polymorphic. For example, if $f : \forall a\,.\,a \to \ldots$ and $arg : \forall b\,.\,\tau$, an annotation will be needed, $f\ (arg :: \forall b\,.\,\tau)$, to instantiate a with $\forall b\,.\,\tau$. However in some cases, annotation propagation and n-ary applications may make such annotations redundant.

Because HM^F requires most general types in derivations, there are programs typeable in HM^F but not in FPH. For example, `let g = append ids in ...` requires an annotation in FPH, whereas it seamlessly type checks in HM^F. On the other hand, flexible instantiation allows FPH to type examples such as

```
f :: forall a. [a] -> [a] -> a
g = f (single id) ids
```

where HM^F (even with annotation propagation) fails. Overall, we believe that the placement of required annotations in FPH is easier to describe than in HM^F. But on the other hand, HM^F possesses a significantly simpler implementation and metatheory.

8.3 The global type inference approach

As discussed in Chapter 3, it is hard to reconcile all the possible instantiations and flows of polymorphic information in a specification that uses local type argument synthesis. The Boxy Types system commits to certain choices (e.g. information flows from functions to arguments) but other choices are also perfectly reasonable, and in some cases required. The HM^F system on the other hand commits to its own choices: eagerly instantiate function arguments, but attempt to remedy some of the locality problems by using information from all arguments of an application in a semi-local-semi-global approach. Moreover it is hard to explain to programmers the hard-coded polymorphic information flows and the choices these type systems make. These complications suggest an alternative approach: Forget about polymorphic information flow and switch to global type argument synthesis, an approach originally proposed in the ML^F work [26].

ML^F, Rigid ML^F, and HML We have already outlined the basic characteristics of the ML^F language of Le Botlan and Rémy [26, 25], which partly inspired this work. The biggest difference between this language and other approaches is that it extends System F types with constraints Q to guarantee principal types of the form $\forall(Q).\tau$ for all typeable expressions. Therefore, `let` expansion preserves typeability in ML^F, unlike systems that use only System F types. Because the type language is more expressive, ML^F requires strictly fewer annotations. In ML^F annotations are necessary only when some argument variable is *used* at two or more types—in contrast, in FPH, variables must be annotated when they must be *typed* with rich types. For example, the following program

```
f = \x -> x ids
```

needs no annotation in ML^F because x is only used once. FPH requires an annotation on `x`. Hence FPH is more restrictive.

The specification of ML^F, which allows it to maintain the `let` expansion property, essentially extends Damas-Milner with constraints: constrained types appear in the declarative type system and the type instance relation of ML^F. The typing judgement of ML^F has the form $(Q)\ \Gamma \vdash e : \tau$ and the ML^F instance relation is of the form $(Q)\ \sigma_1 \sqsubseteq \sigma_2$.

Moreover, to achieve the property that only arguments that must be typed at two different types in the bodies of the abstractions must be annotated, all sharing of polymorphism has to happen through the constraint Q, even when there is no impredicative instantiation going on. This is achieved through ML^F *rigid* bounds of the form $(a = \sigma)$. However, constructing typing derivations in this fashion may be more cumbersome than in System F or Damas-Milner. The following is an example typing derivation in ML^F [29]. Assume that $\tau = (\texttt{Int}, \texttt{Bool})$ and $\sigma_{id} = \forall a\,.\,a \to a$. Moreover let $(\texttt{poly}{:}\sigma_{id} \to \tau) \in \Gamma$ and $(\texttt{id}{:}\sigma_{id}) \in \Gamma$. To type $(\emptyset)\ \Gamma \vdash \texttt{poly id} : (\texttt{Int}, \texttt{Bool})$ we may construct the following derivation in ML^F:

$$\dfrac{\dfrac{\dfrac{\dfrac{(a=\sigma_{id})\ \Gamma \vdash \texttt{poly} : \sigma_{id} \to \tau \qquad (a=\sigma_{id})\ \Gamma \vdash \texttt{id} : \sigma_{id}}{\dfrac{(a=\sigma_{id})\ \sigma_{id} \to \tau \sqsubseteq a \to \tau}{(a=\sigma_{id})\ \Gamma \vdash \texttt{poly} : a \to \tau} \qquad \dfrac{(a=\sigma_{id})\ \sigma_{id} \sqsubseteq a}{(a=\sigma_{id})\ \Gamma \vdash \texttt{id} : a}}}{(a=\sigma_{id})\ \Gamma \vdash \texttt{poly id} : \tau}}{(\emptyset)\ \Gamma \vdash \texttt{poly id} : \forall(a=\sigma_{id}).\tau} \qquad (\emptyset)\ \forall(a=\sigma_{id}).\tau \sqsubseteq \tau}{(\emptyset)\ \Gamma \vdash \texttt{poly id} : (\texttt{Int}, \texttt{Bool})}$$

In addition, ML^F types possess certain syntactic artifacts (which go away if type inference is instead presented as manipulating graphic types [47]). For example ML^F assumes the existence of a type $\bot$ such that every other type is an instance of $\bot$ and moreover $\forall a \geq \bot\,.\,a$ is isomorphic to $\bot$.

The specification of FPH does not rely on types with constraints or reasoning with constraints. Moreover, whenever there is no impredicative instantiation, boxes are not present. For example, the derivation for typing `poly id` *is the same as* the System F derivation:

$$\dfrac{\Gamma \vdash \texttt{poly} : \sigma_{id} \to \tau \quad \Gamma \vdash \texttt{id} : \sigma_{id}}{\Gamma \vdash \texttt{poly id} : (\texttt{Int}, \texttt{Bool})}$$

Impredicative instantiation works in ML^F by abstracting in the constraint Q all polymorphic information about the instantiated variable. For example, consider $(\texttt{h}{:}\forall c.[\sigma_{id} \to c] \to c) \in \Gamma$, which in ML^F is written as $(\texttt{h}{:}\forall(b = \sigma_{id}, c \geq \bot).[b \to c] \to c) \in \Gamma$ and $(\texttt{Nil}{:}\forall a\,.\,[a]) \in \Gamma$, which in ML^F is written as $(\texttt{Nil}{:}\forall(a \geq \bot).[a]) \in \Gamma$. The following ML^F derivation shows how we may type the application `h Nil`.

$$\dfrac{\dfrac{\dfrac{\mathcal{D}_1 \ :: \ (b=\sigma_{id}, c=\sigma_{id})\ \Gamma \vdash \texttt{h} : [b \to c] \to c \qquad \mathcal{D}_2 \ :: \ (b=\sigma_{id}, c=\sigma_{id})\ \Gamma \vdash \texttt{Nil} : [b \to c]}{(b=\sigma_{id}, c=\sigma_{id})\ \Gamma \vdash \texttt{h Nil} : c}}{(\emptyset)\ \Gamma \vdash \texttt{h Nil} : \forall(b=\sigma_{id}, c=\sigma_{id}).\,c} \qquad (\emptyset)\ \forall(b=\sigma_{id}, c=\sigma_{id}).\,c \sqsubseteq \forall(c=\sigma_{id}).\,c}{(\emptyset)\ \Gamma \vdash \texttt{h Nil} : \forall(c=\sigma_{id}).\,c}$$

The derivation $\mathcal{D}_1$ may be:

$$\dfrac{\dfrac{(\emptyset)\ \forall(b = \sigma_{id}, c \geq \bot).[b \to c] \to c \sqsubseteq \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c] \to c}{(\emptyset)\ \Gamma \vdash \mathtt{h} : \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c] \to c}}{(b = \sigma_{id}, c = \sigma_{id})\ \Gamma \vdash \mathtt{h} : [b \to c] \to c}$$

On the other hand, derivation $\mathcal{D}_2$ may be the following:

$$\dfrac{\dfrac{(\emptyset)\ \forall(a \geq \bot).[a] \sqsubseteq \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c]}{(\emptyset)\ \Gamma \vdash \mathtt{Nil} : \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c]}}{(b = \sigma_{id}, c = \sigma_{id})\ \Gamma \vdash \mathtt{Nil} : [b \to c]}$$

where $\forall(a \geq \bot).[a] \sqsubseteq \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c]$ follows by:

$$\begin{array}{rl} & \forall(a \geq \bot).[a] \\ \sqsubseteq & \forall(b \geq \bot, c \geq \bot, a \geq \bot).[a] \\ \sqsubseteq & \forall(b \geq \bot, c \geq \bot, a \geq b \to c).[a] \\ \sqsubseteq & \forall(b \geq \bot, c \geq \bot).[b \to c] \\ \sqsubseteq & \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c] \end{array}$$

Some of the complexity of ML^F arises from the flexibility that it offers in constructing many (quite different) derivations for the same judgement. For example, the following derivation is also possible:

$$\begin{array}{rl} & \forall(a \geq \bot).[a] \\ \sqsubseteq & \forall(a \geq (\forall(b \geq \bot, c \geq \bot).b \to c)).[a] \\ \sqsubseteq & \forall(b \geq \bot, c \geq \bot, a \geq b \to c).[a] \\ \sqsubseteq & \forall(b \geq \bot, c \geq \bot).[b \to c] \\ \sqsubseteq & \forall(b = \sigma_{id}, c = \sigma_{id}).[b \to c] \end{array}$$

Finally, here is a derivation of $\Gamma \vdash \texttt{h Nil} : \boxed{\sigma_{id}}$; the FPH type $\boxed{\sigma_{id}}$ corresponds to the ML^F type $\forall(c = \sigma_{id})\,.\,c$ in FPH:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{h} : \forall c.[\sigma_{id} \to c] \to c}{\Gamma \vdash \texttt{h} : [\sigma_{id} \to \boxed{\sigma_{id}}] \to \boxed{\sigma_{id}}} \qquad \dfrac{\Gamma \vdash \texttt{Nil} : \forall a\,.\,[a]}{\Gamma \vdash \texttt{Nil} : [\boxed{\sigma_{id} \to \sigma_{id}}]} \quad \lfloor [\sigma_{id} \to \boxed{\sigma_{id}}] \rfloor = \lfloor [\boxed{\sigma_{id} \to \sigma_{id}}] \rfloor}{\Gamma \vdash \texttt{h Nil} : \boxed{\sigma_{id}}}$$

The derivation is more straightforward to construct. In fact it is enough to take the System F derivation and place boxes around the impredicative instantiations.

The FPH follows the global type inference approach, and hence it actually does use ML^F-style constraints in its implementation. However, because the FPH system does not expose a constraint-based instance relation in the specification, we can formalize its algorithm as directly manipulating sets of System F types. In contrast, ML^F internalizes the subset relation between sets of System F types as the syntactic instance relation $\sqsubseteq$, and formalizes type inference with respect to this syntactic instance relation. Le Botlan and Rémy study the set-based interpretation of (a slight restriction of) ML^F in a recent report [27], which inspired our set-theoretic interpretation of types with constraints.

Despite the simplifications that FPH provides, there are technical parallels between the specifications of FPH and ML^F. One of the key ideas behind ML^F is that all polymorphic instantiations are "hidden" behind constrained type variables. The FPH type system uses anonymous boxes for the same purpose. Furthermore, the *revelation* of implicit polymorphism is achieved in ML^F at type annotation nodes, where explicit type information is present. Similarly, the revelation of polymorphism is achieved in FPH when a boxed type meets an annotation.

Finally, ML^F is a source language and is translated to an explicitly typed intermediate language, such as explicitly typed System F, using coercion terms [32]. Devising a typed intermediate language for ML^F that is suitable for a compiler and does not require term-level coercions is the subject of current research [48]. In contrast, because FPH is based on System F, there is a straightforward elaboration to System F.

A variation of ML^F very similar in expressive power to FPH is Leijen's Rigid ML^F [29]. Rigid ML^F only includes flexible bounds in the environment constraints but only types with inlined rigid bounds

(System F types) can be assigned to variables in the environment. To achieve this, Rigid ML^F resolves constraints by instantiating flexible bounds at `let` nodes. One of the advantages of Rigid ML^F is that it is considerably simpler to translate to System F without the need for term-level coercions. However, Rigid ML^F is specified using the ML^F instance relation. Consequently, despite the fact that types in the environment are System F types, to reason about typeability one must reason using the ML^F machinery. Additionally, the rules of Rigid ML^F require that when instantiating the types of `let`-bound expressions, the type that is used in the typing derivation of the `let`-bound expression is the most general for the expression. Because of this condition, fewer annotations on `let`-bound expressions are needed in Rigid ML^F, compared to FPH. For example, the program from Section 5.1 is typeable without any type annotations in Rigid ML^F:

```
f :: forall a. a -> [a] -> [a]
ids :: [forall a. a -> a]

h1 = f (\x -> x) ids     -- OK in Rigid MLF
```

In contrast, FPH has to be conservative and demand more annotations on `let`-bound expressions because by design it aims to preserve the property that if a variable in the environment (e.g. `f` above) gets a more general type, no program in its scope becomes untypeable.

Finally, a promising ML^F variation is Leijen's HML system [30]. In particular HML retains flexible bounds and hence enjoys principal types as ML^F, but completely inlines rigid bounds. In contrast to ML^F, annotations must be placed on *all* function arguments that are polymorphic (as in FPH), but it requires absolutely no annotations on `let`-bound definitions (contrary to FPH). The HML system still involves reasoning with constraints, but in the absence of rigid bounds there is no need for the introduction of the ML^F abstraction relation—a significant simplification.

	Specification	**Implementation**	**Expressiveness**
HM^F	Simple, "minimality" restrictions	Simple	Annotations may be needed on λ-abstractions with rich types and on arguments that must be kept polymorphic
ML^F	Heavyweight, declarative	Heavyweight	Precise, annotations only required for usage of argument variables at two or more types
Boxy Types	Complex, syntax-directed, dark corners	Simple	No clear guidelines, not clear what fragment of System F is typed without annotations
HML	Constraint-based, declarative	Heavyweight	Precise, annotations on polymorphic function arguments
FPH	Simple, declarative	Heavyweight	Precise, annotations on `let`-bindings and λ-abstractions with rich types, types all applicative System F terms without annotations

Figure 8.1: Comparison of most relevant works

8.4 Summary of closely connected works

It is interesting to compare some of the most recent proposals for annotation-driven type inference for first-class polymorphism (including higher-rank and impredicative types), in terms of specification simplicity, implementation complexity, placement of type annotations, and expressiveness. We present this comparative analysis in Figure 8.1.

The HM^F system has a simple specification but imposes certain minimality conditions in exchange for a simple specification that uses matching. The ML^F approach exposes constraints in exchange for type inference robustness and a very small number of type annotations. The Boxy Types system has a complex specification that is syntax-directed and combines annotation propagation. Annotation propagation can be added independently in the rest of the Systems of Figure 8.1. The HML system is a simplification of ML^F: In exchange for a few more type annotations (all polymorphic arguments must be annotated) it removes the ML^F abstraction relation and reasoning with rigid bounds. Finally the FPH only uses System F types and has a simple and declarative specification. Both ML^F, HML, and FPH type systems internally use types with constraints, and hence their implementation is more involved than the more traditional implementations of HM^F and Boxy Types.

Chapter 9

Summary and future work

This dissertation has addressed the problem of practical type inference for first-class polymorphism. With the work on higher-rank type systems we have been able to study and clarify the connections between various predicative higher-rank type systems. With the work on FPH we were able to lift the predicativity restriction, and presented a simple, expressive, declarative specification of type inference for impredicative polymorphism. The cost is a somewhat complicated algorithm, but we believe that there exist significant programmability advantages.

With the contributions in this dissertation and the various recent ML^F-based proposals, the problem of type inference for first-class universal polymorphism with practical considerations has become well-understood, and the connections between the various proposals have been clarified. There exist however several directions for future work.

Evaluation of FPH The evaluation of FPH requires extending a prototype implementation that we have developed to a full-scale Haskell compiler, such as GHC. The somewhat unintuitive specifications of the current impredicativity implementation in GHC has kept the Haskell community away from using this feature. We believe that with the simplifications that this dissertation offers compared to the existing implementation in GHC, programmers will become more willing to use

impredicativity instead of coding their way around it. Important issues that need be addressed is that of efficiency and error reporting.

Because the algorithmic implementation is quite different from the specification, the problem of producing comprehensive error messages is not trivial. Additionally, because of the lack of principal types, a `let`-bound definition may accidentally be assigned a different type than the type the programmer had in his mind. The fact that the compiler actually implements a choice in a `let`-bound definition (by picking a box-free type for an expression) may cause subtle errors. For example, we may have the following:

```
cons :: forall a. a -> [a] -> [a]
id   :: forall b. b -> b


f = cons id []
...
bar = let g :: ([forall a. a -> a]) ->
          g = ...
      in g f
```

There is certainly a type error in the program above, but where is it? Was the programmer's intention to give `f` the type $[\forall a\,.\,a \to a]$ or not? Probably the programmer was not aware that `f` got type $\forall a\,.[a \to a]$ instead. So, the "real" type error is not in the body of `bar` but rather in the definition of `f`, which nevertheless does type-check.

One idea to eliminate such situations would be to warn the programmer about different alternative types a `let`-bound variable can be assigned. Algorithmically, this is easy to implement: if the algorithm had to instantiate some flexible constraints then this meant that there was indeed a choice between incomparable types.

Interaction with other forms of constraints We would like to study the interaction with qualified types [15, 10, 19, 17] or constraints arising from *type families* [3, 2] or *generalized algebraic*

datatypes (GADTs) [42]. Preliminary work by Leijen and Löh [32] shows how to combine ML^F-style unification with qualified types, and we expect no significant difficulties to arise.

Extensions for existential types Existential types can be useful, particularly when programming with GADTs. However, existential types complicate the story of type inference because computing the join of two existential types along a reasonable instance relation seems to go "the opposite direction" than ordinary unification for universals. To see why, consider types τ_1 and τ_2 below:

$$\begin{aligned} \tau_1 &= \alpha \to \mathtt{Int} \\ \tau_2 &= \exists a\,.\,a \to a \end{aligned}$$

where α is a yet-unconstrained unification variable. Suppose that we want to compute the join of these two types—for example each of the types is the right-hand side of a pattern match construct. One possible solution would be type $\exists ab.a \to b$, if one assumes that we must delay unifying α. On the other hand, one may claim that α *should be* unified to `Int`, to give back the join $\exists a\,.\,a \to a$, which is more informative than $\exists ab\,.\,a \to b$, and hence better. If we are to have implicit existential sealing should we delay unification or not?

A declarative specification does not involve constrained variables, but rather guessed types in their place. Consequently, the implementation has to simultaneously combine ordinary unification and anti-unification for existential variables. Hence, the problem is complicated, and a challenging direction for future work.

Deep instance relations A distinctive difference between work on predicative higher-rank type inference (Chapter 3) and the work in FPH is that the former relies on elaborate type instance relations that treat quantifiers to the right of arrows effectively as if they were top-level and traverse down the function structure in a co/contra-variant way. This is easy to do because of predicative instantiation—in essence all polymorphic structure of a type is known information arising in some user type annotation. On the other hand, FPH essentially builds upon the impredicative System F instance, which is invariant on any type constructors, including $\to$. Though being able to use more elaborate instance relations *in the presence of impredicativity* is desirable, it is to date not clear how

to specify and implement such an extension. In fact such elaborate relations, such as Mitchell's impredicative type containment relation are, in general, undecidable. Lifting these restrictions is certainly another avenue for future work.

Appendix A

FPH: algorithm metatheory

We present here details of the metatheory of the algorithmic implementation. We first need to extend the quantifier rank definition to schemes and types.

Definition A.0.1 (Scheme quantifier rank)**.** If $\varsigma = [D] \Rightarrow \rho$, we let $q(\varsigma) = q(D)$. Overloading the notation we let $q(\sigma) = 0$.

A.1 Unification termination

Definition A.1.1 (Acyclicity)**.** A constraint C is acyclic, written $\texttt{acyclic}(C)$, for the smallest relation that asserts that for all $(\alpha_m \in bnd) \in C$ it is $\alpha \notin \widehat{C}(bnd)$; and moreover for every $(\alpha_\mu \geq [D] \Rightarrow \rho) \in C$ it is $\texttt{acyclic}(D)$.

Definition A.1.2 (Closedness)**.** A constraint C is closed, written $\texttt{closed}(C)$, if $fcv(C) \subseteq dom(C)$.

Lemma A.1.3 (*mkMono* domain monotonicity)**.** *If* $mkMono(C_1, f, \sigma) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.

Proof. Easy induction on the number of recursive calls to *mkMono*. □

Lemma A.1.4 (Domain monotonicity)**.** *The following are true:*

1. *If* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
2. *If* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
3. *If* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
4. *If* $updateRigid(C_1, \alpha, \sigma) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
5. *If* $doUpdateR(C_1, \alpha, \sigma) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
6. *If* $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
7. *If* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then* $dom(C_1) \subseteq dom(C_2)$.
8. *If* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma_r$ *then* $dom(C_1) \subseteq dom(C_2)$.

Proof. The proof is by simultaneous induction, appealing also to Lemma A.1.3. For each case we assume that the property is true for all other cases when they terminate in a smaller number of recursive calls. □

The following lemma asserts that m-flagged variables never have their flags lifted during unification.

Lemma A.1.5. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $mkMono(C_1, f, \sigma) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.

Proof. Easy induction on the number of recursive calls to *mkMono*. □

Lemma A.1.6 (Monomorphic domain monotonicity)**.** *The following are true:*

1. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.
2. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.
3. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $updateRigid(C_1, \alpha, \sigma) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.
4. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $doUpdateR(C_1, \alpha, \sigma) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.
5. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.
6. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.
7. *If* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma_r$ *then* $\Delta_{C_2}(\beta) = \mathsf{m}$.

Proof. Easy simultaneous induction using Lemma A.1.4 and Lemma A.1.5 and observing that we never lift a m flag during unification; instead we merely convert $\star$ flags to m. □

Lemma A.1.7 (*mkMono* preserves closedness)**.** *If* $\texttt{closed}(C_1)$*,* $C_1 \vdash \sigma$*, and* $mkMono(C_1, f, \sigma) = C_2$ *then* $\texttt{closed}(C_2)$*.*

Proof. Induction on the number of recursive calls to *mkMono*. Case M1 follows by induction hypothesis. Case M2 follows by two uses of the inductive hypothesis and Lemma A.1.3. Cases M3 and M4 are trivial. For case M5 we have that $\texttt{closed}(C_1)$ and $fcv([D] \Rightarrow \rho) \subseteq dom(C_1)$; hence $\texttt{closed}(C_1 \bullet D)$ and $fcv(\rho) \subseteq dom(C_1 \bullet D)$. By induction hypothesis $\texttt{closed}(E)$. To finish the case we need to show that $fcv(\rho) \subseteq dom(E \leftarrow (\alpha_{\mathsf{m}} = \rho))$. By Lemma A.1.3 we know that $fcv(\rho) \subseteq dom(E)$ and also that $\alpha \in dom(E)$, hence $fcv(\rho) \subseteq dom(E \leftarrow (\alpha_{\mathsf{m}} = \rho))$. The case of M6 is similar, and M7 is trivial. □

Lemma A.1.8 (Restriction preserves closedness)**.** *If* $\texttt{closed}(C)$ *and* $\overline{\gamma} = \widehat{C}(\overline{\beta})$ *then* $\texttt{closed}(C_{\overline{\gamma}})$*.*

Proof. Assume by contradiction that there exists a $\gamma \in bnd$ and $(\alpha_\mu\ bnd) \in C_{\overline{\gamma}}$ such that $\gamma \notin dom(C_{\overline{\gamma}})$. It follows that $(\alpha_\mu\ bnd) \in C$ and it also follows that $\alpha \in \widehat{C}(\overline{\beta})$. But in this case it must also be that $\gamma \in \widehat{C}(\overline{\beta})$, and hence $\gamma \in dom(C_{\overline{\gamma}})$—a contradiction. □

Theorem A.1.9 (Unification preserves closedness)**.** *The following are true*

1. $\texttt{closed}(C_1) \wedge fcv(\sigma_1, \sigma_2) \subseteq dom(C_1) \wedge eqCheck(C_1, \sigma_1, \sigma_2) = C_2 \Longrightarrow \texttt{closed}(C_2)$
2. $\texttt{closed}(C_1) \wedge fcv(\varsigma, \sigma, \sigma_0) \subseteq dom(C_1) \wedge subsCheck(C_1, \varsigma, \sigma, \sigma_0) = C_2 \Longrightarrow \texttt{closed}(C_2)$
3. $\texttt{closed}(C_1) \wedge fcv(\alpha, \sigma) \subseteq dom(C_1) \wedge updateRigid(C_1, \alpha, \sigma) = C_2 \Longrightarrow \texttt{closed}(C_2)$
4. $\texttt{closed}(C_1) \wedge fcv(\alpha, \sigma) \subseteq dom(C_1) \wedge doUpdateR(C_1, \alpha, \sigma) = C_2 \Longrightarrow \texttt{closed}(C_2)$
5. $\texttt{closed}(C_1) \wedge fcv(\alpha, \sigma) \subseteq dom(C_1) \wedge updateFlexi(C_1, \alpha, \varsigma) = C_2 \Longrightarrow \texttt{closed}(C_2)$
6. $\texttt{closed}(C_1) \wedge fcv(\alpha, \varsigma) \subseteq dom(C_1) \wedge doUpdateF(C_1, \alpha, \varsigma) = C_2 \Longrightarrow \texttt{closed}(C_2)$
7. $\texttt{closed}(C_1) \wedge fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1) \wedge join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma \Longrightarrow \texttt{closed}(C_2) \wedge fcv(\varsigma) \subseteq dom(C_2)$

Proof. We prove the cases simultaneously by induction on the number of recursive calls. For each case we assume that all hold for calls that terminate in a smaller number of recursive calls.

1. Case E1 is trivial. Case E2 follows by induction hypothesis for *updateRigid*, and similarly case E3. For E4 we have that $\texttt{closed}(C_1)$ and $fcv(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \subseteq dom(C_1)$. Moreover

eqCheck$(C_1, \sigma_1 \to \sigma_2, \sigma_3 \to \sigma_4) = C_2$ where we additionally have $E =$ *eqCheck*$(C_1, \sigma_1, \sigma_3)$ and $C_2 =$ *eqCheck*(E, σ_2, σ_4). By induction hypothesis we get that $\texttt{closed}(E)$. By Lemma A.1.4 we get that $\mathit{fcv}(\sigma_2, \sigma_4) \subseteq \mathit{dom}(E)$. Consequently, by induction hypothesis we get that $\texttt{closed}(C_2)$, as required. For case E5 we have that $\texttt{closed}(C_1)$ and $\mathit{fcv}(\forall \overline{a}\,.\,\rho_1, \forall \overline{a}\,.\,\rho_2) \subseteq \mathit{dom}(C_1)$. Consequently also $\mathit{fcv}([\overline{a \mapsto b}]\rho_1, [\overline{a \mapsto b}]\rho_2) \subseteq \mathit{dom}(C_1)$ and by induction hypothesis it is $\texttt{closed}(E)$ with $C_2 = E$. Case E6 cannot happen.

2. Case S1 follows by induction hypothesis for *updateFlexi*. For case S2 we have that $\texttt{closed}(C_1)$ and $\mathit{fcv}([D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2) \subseteq \mathit{dom}(C_1)$. It follows that $\mathit{fcv}(\rho_1, \rho_2) \subseteq \mathit{dom}(C_1 \bullet D)$ and moreover we get that $\texttt{closed}(C_1 \bullet D)$. By induction hypothesis then $\texttt{closed}(E)$. Because $\overline{\gamma} = \widehat{E}(\mathit{dom}(C_1))$, it must also be $\texttt{closed}(E_{\overline{\gamma}})$, by Lemma A.1.8.
3. Case UR1 is trivial. For UR2, we show that $\mathit{fcv}(\alpha, \gamma) \subseteq \mathit{dom}(C_1)$. But we know that $\mathit{fcv}(\alpha, \beta) \subseteq \mathit{dom}(C_1)$, and moreover $\texttt{closed}(C_1)$. Since $(\beta_\mu = \gamma) \in C_1$, it must be that $\gamma \in \mathit{dom}(C_1)$. We can then apply the induction hypothesis to get that $\texttt{closed}(C_2)$. The case for UR3 is similar. Case UR4 follows by induction hypothesis for *doUpdateR*.
4. For case DR1 we know that $\mathit{fcv}(\alpha, \sigma) \in \mathit{dom}(C_1)$ and $\texttt{closed}(C_1)$. It follows using Lemma A.1.7 that $\texttt{closed}(E)$. To finish the case it is enough to show that $\mathit{fcv}(\sigma) \subseteq \mathit{dom}(E - \alpha)$. We know that $\alpha \notin \mathit{fcv}(\sigma)$ and hence it suffices to show that $\mathit{fcv}(\sigma) \subseteq \mathit{dom}(E)$. But that follows by Lemma A.1.4. Case DR2 is similar. For the case of DR3 we know by induction hypothesis that $\texttt{closed}(E)$. Moreover to finish the case we need to show that $\mathit{fcv}(\sigma) \subseteq \mathit{dom}(E \leftarrow (\alpha_m = \sigma))$. But we know that $\mathit{fcv}(\sigma) \subseteq \mathit{dom}(E)$ by Lemma A.1.4 and also $\alpha \in \mathit{dom}(E)$, and we are done. Case DR4. We know that $\texttt{closed}(C_1)$ and $\mathit{fcv}(\alpha, \sigma) \subseteq \mathit{dom}(C_1)$, hence also $\mathit{fcv}(\sigma_\alpha) \subseteq \mathit{dom}(C_1)$. Applying the induction hypothesis for *eqCheck* finishes the case.
5. For case UF1 we know that $\texttt{closed}(C_1)$ and $\mathit{fcv}(\alpha, [D] \Rightarrow \gamma) \subseteq \mathit{dom}(C_1)$—it follows that $\mathit{fcv}(\alpha, \gamma) \subseteq \mathit{dom}(C_1)$ and induction hypothesis for *updateRigid* finishes the case. For case UF2 appealing to induction hypothesis for *doUpdateF* shows the goal.
6. Cases DF1, DF2, DF3 are similar to the corresponding cases of *doUpdateR*. For DF4 we know that $\texttt{closed}(C_1)$ and $\mathit{fcv}(\alpha, \varsigma) \subseteq \mathit{dom}(C_1)$, hence also $\mathit{fcv}(\varsigma_a, \varsigma) \subseteq \mathit{dom}(C_1)$. By induction hypothesis then for *join* we get that $\texttt{closed}(E)$ and $\mathit{fcv}(\varsigma_r) \subseteq \mathit{dom}(E)$, hence also $\mathit{fcv}(\varsigma_r) \subseteq \mathit{dom}(E \leftarrow (\alpha_\mu \geq \varsigma_r))$.

7. Case J1 follows trivially and case J2 is similar. For J3 we have that $\texttt{closed}(C_1)$ and $fcv([D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) \subseteq dom(C_1)$. It follows that $\texttt{closed}(C_1 \bullet D_1 \bullet D_2)$ and it must be the case that $fcv(\rho_1, \rho_2) \subseteq dom(C_1 \bullet D_1 \bullet D_2)$. By induction E is closed, and $E_{\widehat{E}(dom(C_1))}$ also closed (by Lemma A.1.8). To finish the case we need to show that $fcv([E_{\overline{\delta}}] \Rightarrow \rho_1) \subseteq dom(E_{\widehat{E}(dom(C_1))})$. Pick then a variable $\alpha \in fcv([E_{\overline{\delta}}] \Rightarrow \rho_1)$. It must be that $\alpha \notin (\widehat{E}(\rho_1) - \widehat{E}(dom(C_1)))$. Hence we have two cases. If $\alpha \in \widehat{E}(\rho_1)$ then it must be that $\alpha \in \widehat{E}(dom(C_1))$ and $\alpha \in dom(E_{\widehat{E}(dom(C_1))})$ because it must be that $\alpha \in dom(E)$ (by domain monotonicity). On the other hand, if $\alpha \notin \widehat{E}(\rho_1)$ then $\alpha \notin fcv([E_{\overline{\delta}}] \Rightarrow \rho_1)$, a contradiction.

□

Lemma A.1.10 (*mkMono* single-variable separation)**.** *If* $\alpha \notin \widehat{C_1}(\sigma, \sigma_0)$ *and* $\alpha \in dom(C_1)$*, and* $mkMono(C_1, f, \sigma) = C_2$*, then* $mkMono(C_1 - \alpha, f, \sigma) = C_2 - \alpha$ *in the same number of steps, and* $\alpha \notin \widehat{C_2}(\sigma, \sigma_0)$.

Proof. By induction on the number of steps that *mkMono* performs. Cases M1, M3, and M4 are straightforward. For case M2 we have that $\alpha \notin \widehat{C_1}(\sigma_1 \to \sigma_2, \sigma_0)$ and $mkMono(C_1, f, \sigma_1 \to \sigma_2) = C_2$ where $E = mkMono(C_1, f, \sigma_1)$ and $C_2 = mkMono(E, f, \sigma_2)$. By induction hypothesis $E - \alpha = mkMono(C_1 - \alpha, f, \sigma_1)$ and moreover $\alpha \notin \widehat{E}(\sigma_1, \sigma_2, \sigma_0)$. By Lemma A.1.3 we know that $\alpha \in dom(E)$. Hence, by induction hypothesis $C_2 - \alpha = mkMono(E - \alpha, f, \sigma_2)$ and $\alpha \notin \widehat{C_2}(\sigma_2, \sigma_1, \sigma_0)$. That is, $\alpha \notin \widehat{C_2}(\sigma_1 \to \sigma_2, \sigma_0)$. For case M5, we have that $\beta \notin \widehat{C_1}(\alpha, \sigma_0)$ and $mkMono(C_1, f, \alpha) = (E \leftarrow (\alpha_m = \rho))$ whenever $E = mkMono(C_1 \bullet D, \rho)$ and $(\alpha_\mu \geq [D] \Rightarrow \rho) \in C_1$. Then we know that $\beta \notin \widehat{C_1 \bullet D}(\rho) - dom(D)$ but also that $\beta \in dom(C_1)$; hence it is not in the domain of D and consequently, $\beta \notin \widehat{C_1 \bullet D}(\rho)$. By induction hypothesis then $(E - \beta) = mkMono((C_1 - \beta) \bullet D, \rho)$ and $\beta \notin \widehat{E}(\rho, \sigma_0)$. Consequently $mkMono(C_1 - \beta, f, \alpha) = (E - \beta) \leftarrow (\alpha_{\mathsf{m}} = \rho)$ as required. Case M6 is similar to case M5. Case M7 is an easy check. □

Lemma A.1.11. *If* $\texttt{acyclic}(C)$ *and* $\texttt{acyclic}(D)$*, then* $\texttt{acyclic}(C \bullet D)$.

Proof. Easy check. □

Below, we use the symbol $\rightsquigarrow_E$ for the *reachability relation* induced by the bounds of a constraint E. Each edge in this graph corresponds to a set of edges between a bound variable α and the $fcv(bnd)$, in any constraint of the form $(\alpha_\mu\ bnd) \in E$.

Lemma A.1.12 (*mkMono* preserves acyclicity)**.** *If* $\texttt{acyclic}(C_1)$ *and* $mkMono(C_1, f, \sigma) = C_2$ *then* $\texttt{acyclic}(C_2)$.

Proof. By induction on the number of recursive calls to *mkMono*. Case M1 is trivial, and M2 follows by two uses of the induction hypothesis. Cases M3 and M4 are trivial. For M5, we first know that $mkMono(C_1 \bullet D, f, \rho) = E$. The constraint $C_1 \bullet D$ must then be acyclic as well, and by induction $\texttt{acyclic}(E)$. We want to show now that $\texttt{acyclic}(E \leftarrow (\alpha_\mathsf{m} = \rho))$. However we know that $\alpha \notin \widehat{C_1 \bullet D}(\rho)$ since $\alpha \in dom\, C_1$ and $acyclic(C_1)$. In this case, in order for $E \leftarrow (\alpha_\mathsf{m} = \rho)$ to have a cycle it must be that the cycle "closes-off" by some edge induced by $\alpha_\mathsf{m} = \rho$. In other words, there exists a variable $\gamma \rightsquigarrow_{E-\alpha} \alpha_\mathsf{m} = \rho \rightsquigarrow_{E-\alpha} \gamma$. Which means that $\rho \rightsquigarrow_{E-\alpha} \alpha$. But $\alpha \notin \widehat{E}(\rho)$ by Lemma A.1.10, and it cannot be that $\rho \rightsquigarrow_{E-\alpha} \alpha$ either. For M6 the proof is similar. Case M7 does not affect the reachability graph induced by the constraint, and hence the proof follows. □

The following are two "separation" lemmas. Parts of the constraints that are not touched by the algorithm appear intact in the output constraint. We write $C_1\, C_2$ for the disjoint union of two constraints C_1 and C_2 with respect to their domains, without any extra conditions whatsoever (as in $C_1 \bullet C_2$).

Theorem A.1.13 (*mkMono* separation)**.** *If* $\texttt{closed}(C_1)$ *and* $fcv(\sigma) \subseteq dom(C_1)$ *and additionally it is* $mkMono(C_1\, C_r, f, \sigma) = C$ *then* $mkMono(C_1, f, \sigma) = C_2$ *in the same number of recursive calls, and such that* $C = C_2\, C_r$.

Proof. By induction on the number of recursive calls to *mkMono*. Cases M1, M3, and M4 are tirival. For M2 we have that $\texttt{closed}(C_1)$, $fcv(\sigma_1 \to \sigma_2) \subseteq dom(C_1)$. Additionally, $mkMono(C_1\, C_r, f, \sigma_1) = E$ and $C = mkMono(E, f, \sigma_2)$. We have by induction that $mkMono(C_1, f, \sigma_1) = C_2$ such that $E = C_2\, C_r$. Moreover by Theorem A.1.9 we get that $\texttt{closed}(C_2)$ and by Lemma A.1.3 $fcv(\sigma_2) \subseteq dom(C_2)$. It follows by induction hypothesis that $mkMono(C_2, f, \sigma_2) = E_2$ such that $C = E_2\, C_r$, as required. For M5 we have that $\texttt{closed}(C_1)$, $\alpha \in dom(C_1)$, and $mkMono(C_1\, C_r, f, \alpha) = (E \leftarrow (\alpha_m = \rho))$, where

$E = mkMono(True, C_1\,C_r \bullet D, \rho)$ and $(\alpha_\mu \geq [D] \Rightarrow \rho) \in C_1$. We know however that $\texttt{closed}(C_1 \bullet D)$ since $\texttt{closed}(C_1)$ and $\alpha \in dom(C_1)$. It also follows that $fcv(\rho) \subseteq dom(C_1 \bullet D)$. By induction hypothesis then $E_2 = mkMono(True, C_1 \bullet D, \rho)$ and $E = E_2\,C_r$. Since $\alpha \in dom(C_1)$ it follows that $\alpha \in dom(E_2)$ and hence $E \leftarrow (\alpha_m = \rho) = (E_2 \leftarrow (\alpha_m = \rho))C_r$ as required. The case for M6 is similar. The case for M7 is straightforward. □

Theorem A.1.14 (Unification separation)**.** *The following are true:*

1. *If* $\texttt{closed}(C_1)$*,* $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$*, and* $eqCheck(C_1\,C_r, \sigma_1, \sigma_2) = C$ *then there exists a* C_2 *such that* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$*, and moreover* $C = C_2\,C_r$*.*
2. *If* $\texttt{closed}(C_1)$*,* $fcv(\varsigma, \sigma) \subseteq dom(C_1)$*, and* $subsCheck(C_1\,C_r, \varsigma, \sigma) = C$ *then there exists a* C_2 *such that* $subsCheck(C_1, \varsigma, \sigma) = C_2$*, and moreover* $C = C_2\,C_r$*.*
3. *If* $\texttt{closed}(C_1)$*,* $fcv(\alpha, \sigma) \subseteq dom(C_1)$*, and* $updateRigid(C_1\,C_r, \alpha, \sigma) = C$ *then there exists a* C_2 *such that* $updateRigid(C_1, \alpha, \sigma) = C_2$*, and moreover* $C = C_2\,C_r$*.*
4. *If* $\texttt{closed}(C_1)$*,* $fcv(\alpha, \sigma) \subseteq dom(C_1)$*, and* $doUpdateR(C_1\,C_r, \alpha, \sigma) = C$ *then there exists a* C_2 *such that* $doUpdateR(C_1, \alpha, \sigma) = C_2$*, and moreover* $C = C_2\,C_r$*.*
5. *If* $\texttt{closed}(C_1)$*,* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$*, and* $updateFlexi(C_1\,C_r, \alpha, \varsigma) = C$ *then there exists a* C_2 *such that* $updateFlexi(C_1, \alpha, \varsigma) = C_2$*, and moreover* $C = C_2\,C_r$*.*
6. *If* $\texttt{closed}(C_1)$*,* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$*, and* $doUpdateF(C_1\,C_r, \alpha, \varsigma) = C$ *then there exists a* C_2 *such that* $doUpdateF(C_1, \alpha, \varsigma) = C_2$*, and moreover* $C = C_2\,C_r$*.*
7. *If* $\texttt{closed}(C_1)$*,* $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$*, and* $join(C_1\,C_r, \varsigma_1, \varsigma_2) = C, \varsigma_r$ *then there exists a* C_2 *such that* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma_r$*, and moreover* $C = C_2\,C_r$*.*

Moreover, in each case the number of recursive calls is preserved.

Proof. We prove the cases simultaneously by induction on the number of recursive calls. We assume that all cases are true for a smaller number of recursive calls. We have the following:

1. Case E1 follows trivially. Cases E2 and E3 follow by the induction hypothesis for *updateRigid*. Case E4 follows by two applications of the induction hypothesis, the second one enabled by making use of Lemma A.1.4 and Theorem A.1.9. Case E5 follows directly by induction hypothesis, and case E6 cannot happen.

2. Case S1 follows directly by induction hypothesis for *updateFlexi*. We now turn our attention to case S2 (which is actually the only really interesting case along with case J3). In this case we have the following:

$$\texttt{closed}(C_1) \tag{A.1}$$

$$fcv([D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2) \subseteq dom(C_1) \tag{A.2}$$

$$eqCheck(C_1\,C_r, [D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2) = C \tag{A.3}$$

where

$$\rho_3 = [\overline{c \mapsto b}]\rho_2 \tag{A.4}$$

$$E = eqCheck(C_1\,C_r \bullet D, \rho_1, \rho_2) \tag{A.5}$$

$$\overline{\gamma} = \widehat{E}(dom(C_1\,C_r)) \quad \overline{b} \# E_{\overline{\gamma}} \tag{A.6}$$

From (A.1) and (A.2) we confirm that $fcv(\rho_1, \rho_3) \in dom(C_1 \bullet D)$, and additionally it must be that $\texttt{closed}(C_1 \bullet D)$. By induction hypothesis then for (A.5) we get that

$$eqCheck(C_1 \bullet D, \rho_1, \rho_2) = E_2$$

such that $E = E_2\,C_r$. To finish the case we need to show that:

$$(E_2\,C_r)_{\widehat{E_2\,C_r}(dom(C_1\,C_r))} = E_{2\widehat{E_2}(dom(C_1))}\,C_r$$

We consider each direction separately:

- Assume $(\gamma_\mu\ bnd) \in E_{2\widehat{E_2}(dom(C_1))}\,C_r$. If $(\gamma_\mu\ bnd) \in C_r$ then clearly $(\gamma_\mu\ bnd) \in E_2\,C_r$ and $\gamma \in \widehat{E_2\,C_r}(dom(C_1\,C_r))$ as required. If on the other hand $(\gamma_\mu\ bnd) \in E_2$ and $\gamma \in \widehat{E_2}(dom(C_1))$, it also follows that $(\gamma_\mu\ bnd) \in E_2\,C_r$ and $\gamma \in \widehat{E_2\,C_r}(dom(C_1\,C_r))$ as required.

- Assume now that:

$$(\gamma_\mu\ bnd) \in E_2 C_r \tag{A.7}$$

$$\gamma \in \widehat{E_2 C_r}(dom(C_1 C_r)) \tag{A.8}$$

First of all, if $(\gamma_\mu\ bnd) \in C_r$ then directly $(\gamma_\mu\ bnd) \in E_{2\widehat{E_2}(dom(C_1))} C_r$ as required. Assume now that $(\gamma_\mu\ bnd) \in E_2$ and $\gamma \notin dom(C_r)$. Then we may have two cases:

  - Purely (i.e. without involving C_r) it is $\gamma \in \widehat{E_2}(dom(C_1))$, in which case we are done; otherwise
  - we have the following path:

$$\alpha \in dom(C_1) \rightsquigarrow_{E_2} \beta \mapsto_{C_r} \rightsquigarrow_{E_2 Cr} \gamma$$

Let us explain: There is a variable $\alpha \in dom(C_1)$. Now because this variable is not in the domain of C_r it can only be mapped to some other type first by some uses of E_2, hence the $\rightsquigarrow_{E_2}$ notation. But at some point we will reach a variable $\beta \notin dom(E_2)$ that will be mapped via C_r to some other type (hence the notation $\beta \mapsto_{C_r}$). But E_2 is closed, and by Lemma A.1.4 $dom(C_1) \subseteq dom(E_2)$. It follows that this case is impossible, and we are done.

3. Case UR1 is trivial. Case UR2 follows by closedness of C_1, the assumptions, and induction hypothesis. Case UR3 is similar. Case UR4 follows by closedness of C_1, the assumptions, and induction hypothesis.
4. Case DR1 and DR2 are straightforward, appealing to Theorem A.1.13. Case DR3 and case DR4 use the induction hypothesis.
5. Cases UF1,UF2 are similar to UR1 and UR4 respectively.
6. Cases DF1 and DF2 are straightforward, appealing to Theorem A.1.13. Cases DF3 and case DF4 use the induction hypothesis.
7. Cases J1 and J2 are straightforward. The interesting case is J3 which follows in a similar way as the case for S2.

□

We overload below the notation $\texttt{acyclic}(\cdot)$ to schemes. We write $\texttt{acyclic}([D] \Rightarrow \rho)$ to mean $\texttt{acyclic}(D)$.

Theorem A.1.15 (Unification preserves acyclicity)**.** *Assume that in all cases, the constrained variables of the arguments are contained within the domain of C_1. Then, the following are true:*

1. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge \mathit{eqCheck}(C_1, \sigma_1, \sigma_2) = C_2 \Longrightarrow \texttt{acyclic}(C_2)$
2. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1, \varsigma) \wedge \mathit{subsCheck}(C_1, \varsigma, \sigma, \sigma_0) = C_2 \Longrightarrow \texttt{acyclic}(C_2)$
3. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge \mathit{updateRigid}(C_1, \alpha, \sigma) = C_2 \Longrightarrow \texttt{acyclic}(C_2)$
4. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge \alpha \notin \widehat{C_1}(\sigma) \wedge \mathit{doUpdateR}(C_1, \alpha, \sigma) = C_2 \Longrightarrow \texttt{acyclic}(C_2)$
5. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1, \varsigma) \wedge \mathit{updateFlexi}(C_1, \alpha, \varsigma, \sigma_0) = C_2 \Longrightarrow \texttt{acyclic}(C_2)$
6. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1, \varsigma) \wedge \alpha \notin \widehat{C_1}(\varsigma) \wedge \mathit{doUpdateF}(C_1, \alpha, \varsigma) = C_2 \Longrightarrow \texttt{acyclic}(C_2)$
7. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1, \varsigma_1, \varsigma_2) \wedge \mathit{join}(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma \Longrightarrow \texttt{acyclic}(C_2, \varsigma_r)$

Proof. The proof is straightforward induction on the number of recursive calls, and the only interesting cases are those that update the constraint. Let us examine one case, the case for DR3. In this case we know that:

$$\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \tag{A.9}$$

$$\mathit{doUpdateF}(C_1, \alpha, \sigma) = E \leftarrow (\alpha_\mu = \sigma) \tag{A.10}$$

$$(\alpha_\mu \geq \varsigma) \in C_1 \tag{A.11}$$

$$E = \mathit{subsCheck}(C_1, \varsigma_a, \sigma) \tag{A.12}$$

Induction hypothesis here gives us that E is acyclic, hence also $E - \alpha$ is acyclic. Assume that adding $\alpha_\mu = \sigma$ closes-off a cycle. Hence there is a $\gamma \rightsquigarrow_{E-\alpha} \alpha_\mu = \sigma \rightsquigarrow_{E-\alpha} \gamma$, which means that $\alpha \in \widehat{E - \alpha}(\sigma)$. It hence suffices to show that $\alpha \notin \widehat{E}(\sigma)$.

Consider the set of variables $\overline{\gamma} = \widehat{C_1}(\varsigma_a, \sigma)$ and the restriction of C_1 to these variables $C_{1\overline{\gamma}}$; let C_r be such that $C_{1\overline{\gamma}} C_r = C_1$. It must be that $C_{1\overline{\gamma}}$ is acyclic and closed and hence by Theorem A.1.14 $E_1 = \mathit{subsCheck}(C_{1\overline{\gamma}}, \varsigma_a, \varsigma)$ and $E = E_1 C_r$. Then we want to show essentially that $\alpha \notin \widehat{E_1 C_r}(\sigma)$

given that $\alpha \in dom(C_r)$, $\alpha \notin fcv(\sigma)$. But this follows as E_1 is closed and we can only remain within its domain as we move along reachability. □

Lemma A.1.16 (*mkMono* preserves quantifier rank)**.** *If* $mkMono(C_1, f, \sigma) = C_2$ *then* $q(C_2) \leq q(C_1)$.

Proof. Straightforward induction on the number of recursive calls to *mkMono*. □

In what follows, we use Lemma A.1.7, Theorem A.1.9, Lemma A.1.12, and Theorem A.1.15, in order to enable the induction hypotheses. We additionally use the domain monotonicity properties, Lemma A.1.3 and Lemma A.1.4. In order to focus on the interesting parts of the proofs we omit explicitly referring to their uses.

Lemma A.1.17 (Unification preserves quantifier rank)**.** *The following are true, assuming that the argument variables belong each time in the input constraint* C_1.

1. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge eqCheck(C_1, \sigma_1, \sigma_2) = C_2 \Longrightarrow q(C_2) \leq q(C_1)$
2. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge subsCheck(C_1, \varsigma, \sigma, \sigma_0) = C_2 \Longrightarrow q(C_2) \leq q(C_1) + q(\varsigma)$
3. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge updateRigid(C_1, \alpha, \sigma) = C_2 \Longrightarrow q(C_2) \leq q(C_1)$
4. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge \alpha \notin \widehat{C_1}(\sigma) \wedge doUpdateR(C_1, \alpha, \sigma) = C_2 \Longrightarrow q(C_2) \leq q(C_1)$
5. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge updateFlexi(C_1, \alpha, \varsigma, \sigma_0) = C_2 \Longrightarrow q(C_2) \leq q(C_1) + q(\varsigma)$
6. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge \alpha \notin \widehat{C_1}(\varsigma) \wedge doUpdateF(C_1, \alpha, \varsigma) = C_2 \Longrightarrow q(C_2) \leq q(C_1) + q(\varsigma)$
7. $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma \Longrightarrow q(C_2) + q(\varsigma) \leq q(C_1) + q(\varsigma_1) + q(\varsigma_2)$

Proof. We prove each case by induction on the number of recursive calls. We assume all cases for a smaller number of recursive calls.

1. Case E1 is trivial. Cases E2 and E3 follow by induction hypothesis for *updateRigid*. Case E4 follows by two uses of induction hypothesis, the second enabled using Theorem A.1.15 and Theorem A.1.14 (and domain monotonicity). Case E5 follows by one use of the induction hypothesis.

2. Case S1 follows by induction hypothesis for *updateFlexi*. In the case for S2 we have that $q(E_{\overline{\gamma}}) \leq q(E) \leq q(C \bullet D) \leq q(C) + q([D] \Rightarrow \rho)$ as required.
3. Case UR1 is trivial. Case UR2 follows by induction hypothesis for *updateRigid*. Case UR3 follows by induction hypothesis for *updateRigid* and case UR4 follows by induction hypothesis for *doUpdateR*.
4. Case DR1 follows by Lemma A.1.16. Case DR2 is straightforward. The interesting cases are DR3 and DR4. In the DR3 case we have that

$$\begin{aligned} \alpha \notin \widehat{C_1}(\sigma) \\ (\alpha_\mu \geq \varsigma_a) \in C_1 \\ E = \mathit{subsCheck}(C_1, \varsigma_a, \sigma) \\ C_2 = E \leftarrow (\alpha_m = \sigma) \end{aligned}$$

In this case consider the following set $\overline{\gamma} = \widehat{C_1}(\varsigma_a, \sigma)$. It follows that $\texttt{closed}(C_{1\overline{\gamma}})$ and let C_r be such that $C_{1\overline{\gamma}} C_r = C_1$. From this, and Theorem A.1.14 for (A.13) we get that $\mathit{subsCheck}(C_{1\overline{\gamma}}, \varsigma_a, \varsigma) = E_1$ such that $E = E_1 C_r$. But we know that $\alpha \notin \mathit{dom}(E_1)$. Hence $C_2 = E_1 \cup C_r \leftarrow (\alpha_\mu = \sigma)$. We then get that:

$$\begin{aligned} q(C_2) &= q(E_1) + q(C_r \leftarrow (\alpha_\mu = \sigma)) \\ &\leq q(C_{1\overline{\gamma}}) + q(\varsigma_a) + q(C_r - \alpha) + 1 \\ &= q(C_{1\overline{\gamma}} + q(C_r) \\ &= q(C_1) \end{aligned}$$

where in the second line we made use of the inductive hypothesis for $\mathit{subsCheck}(C_{1\overline{\gamma}}, \varsigma_a, \varsigma) = E_1$, as we know that it uses the same number of recursive calls as does the equation (A.13). The DR4 case is similar.
5. The cases for UF1 follows by appealing to the induction hypothesis for *updateRigid*. The case for UF2 follows by appealing to the induction hypothesis for *doUpdateF*.

6. The case for DF1 is straightforward appealing to Lemma A.1.16. Case DF2 is straightforward. Case DF3 is similar to the case for DR3. We consider now the case for DF4. We have in this case that:

$$\begin{aligned} \alpha \notin \widehat{C_1}(\varsigma) \\ (\alpha_\mu \geq \varsigma_a) \in C_1 \\ E, \varsigma_r = join(C_1, \varsigma_a, \varsigma) \\ C_2 = E \leftarrow (\alpha_\mu \geq \varsigma_r) \end{aligned}$$

As in the case of DF3 consider the set $\overline{\gamma} = \widehat{C_1}(\varsigma_a, \varsigma)$. It follows that $\texttt{closed}(C_{1\overline{\gamma}})$ and let C_r be such that $C_{1\overline{\gamma}} C_r = C_1$. From this and Theorem A.1.14 for (A.13) we get that $join(C_{1\overline{\gamma}}, \varsigma_a, \varsigma) = E_1, \varsigma_r$ such that $E = E_1 C_r$. But we know that $\alpha \notin dom(E_1)$. Hence $C_2 = E_1 \cup C_r \leftarrow (\alpha_\mu \geq \varsigma_r)$. We then get that:

$$\begin{aligned} q(C_2) &= q(E_1) + q(C_r \leftarrow (\alpha_\mu \geq \varsigma_r)) \\ &= (q(E_1) + q(\varsigma_r)) + q(C_r - \alpha) + 1 \\ &\leq q(C_{1\overline{\gamma}} + q(\varsigma_a) + q(\varsigma) + q(C_r - \alpha) + 1) \\ &= q(C_{1\overline{\gamma}} + q(C_r) + q(\varsigma) \\ &= q(C_1) + q(\varsigma) \end{aligned}$$

where in the third line we made use of the inductive hypothesis for $join(C_{1\overline{\gamma}}, \varsigma_a, \varsigma) = E_1, \varsigma_r$, as we know that it uses the same number of recursive calls as equation (A.13).

7. Cases J1 and J2 are easy checks. For J3 we have the following: First of all $C_1 \bullet D_1 \bullet D_2$ is acyclic and closed and hence we can apply the induction hypothesis. It follows that

$$q(E_{\widehat{E(dom(C))}}, [E_{\overline{\delta}}] \Rightarrow \rho_1) \leq q(E) \leq q(C \bullet D_1 \bullet D_2) \leq q(C) + q([D_1] \Rightarrow \rho_1) + q([D_2] \Rightarrow \rho_2)$$

as required.

□

Theorem A.1.18 (*mkMono* weakening)**.** *If* $\texttt{closed}(C_1)$ *and* $fcv(\sigma) \subseteq dom(C_1)$ *and additionally it is* $mkMono(C_1, f, \sigma) = C_2$ *then* $mkMono(C_1\,C_r, f, \sigma) = C_2\,C_r$ *in the same number of recursive calls (and the second fails if the first fails).*

Proof. Easy induction, similar to the proof of Theorem A.1.13. □

Theorem A.1.19 (Unification weakening)**.** *The following are true:*

1. *If* $\texttt{closed}(C_1)$ *and* $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$ *and* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then it is the case that* $eqCheck(C_1\,C_r, \sigma_1, \sigma_2) = C_2\,C_r$.
2. *If* $\texttt{closed}(C_1)$ *and* $fcv(\varsigma, \sigma) \subseteq dom(C_1)$ *and* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then it is the case that* $subsCheck(C_1\,C_r, \varsigma, \sigma) = C_2\,C_r$.
3. *If* $\texttt{closed}(C_1)$ *and* $fcv(\alpha, \sigma) \subseteq dom(C_1)$ *and* $updateRigid(C_1, \alpha, \sigma) = C_2$ *then it is the case that* $updateRigid(C_1\,C_r, \alpha, \sigma) = C_2\,C_r$.
4. *If* $\texttt{closed}(C_1)$ *and* $fcv(\alpha, \sigma) \subseteq dom(C_1)$ *and* $doUpdateR(C_1, \alpha, \sigma) = C_2$ *and it is the case that* $doUpdateR(C_1\,C_r, \alpha, \sigma) = C_2\,C_r$.
5. *If* $\texttt{closed}(C_1)$ *and* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$ *and* $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then it is the case that* $updateFlexi(C_1\,C_r, \alpha, \varsigma) = C_2\,C_r$.
6. *If* $\texttt{closed}(C_1)$ *and* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$ *and* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then it is the case that* $doUpdateF(C_1\,C_r, \alpha, \varsigma) = C_2\,C_r$.
7. *If* $\texttt{closed}(C_1)$ *and* $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$ *and* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma_r$ *then it is the case that* $join(C_1\,C_r, \varsigma_1, \varsigma_2) = C_2\,C_r, \varsigma_r$.

Moreover the number of recursive calls is preserved, and if the first call fails instead, then the second fails as well.

Proof. Simultaneous induction on the number of recursive calls, similar to the proof of Theorem A.1.14. □

Theorem A.1.20 (Termination)**.** *The unification algorithm terminates (written with the notation* $\Downarrow$*) when given well-formed inputs. In particular*

1. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1)$ *and* $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$ *then* $eqCheck(C_1, \sigma_1, \sigma_2) \Downarrow$.
2. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1, \varsigma)$ *and* $fcv(\varsigma, \sigma) \subseteq dom(C_1)$ *then* $subsCheck(C_1, \varsigma, \sigma) \Downarrow$.
3. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1)$ *and* $fcv(\alpha, \sigma) \subseteq dom(C_1)$ *then* $updateRigid(C_1, \alpha, \sigma) \Downarrow$.
4. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1)$ *and* $fcv(\alpha, \sigma) \subseteq dom(C_1)$ *and* $\alpha \notin \widehat{C_1}(\sigma)$ *then it is the case that* $doUpdateR(C_1, \alpha, \sigma) \Downarrow$.
5. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1, \varsigma)$ *and* $fcv(\varsigma, \alpha) \subseteq dom(C_1)$ *then* $updateFlexi(C_1, \alpha, \varsigma) \Downarrow$.
6. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1, \varsigma)$ *and* $fcv(\varsigma, \alpha) \subseteq dom(C_1)$ *and* $\alpha \notin \widehat{C_1}(\varsigma)$ *then it is the case that* $doUpdateF(C_1, \alpha, \varsigma) \Downarrow$.
7. *If* $\texttt{closed}(C_1)$ *and* $\texttt{acyclic}(C_1, \varsigma_1, \varsigma_2)$ *and* $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$ *then* $join(C_1, \varsigma_1, \varsigma_2) \Downarrow$.

Proof. Assume that ϖ stands for either types or schemes. We assume that $q(\varpi) = 0$ whenever $\varpi = \sigma$, or $q(\varpi) = q(\varsigma)$ when $\varpi = \varsigma$. For a triple $(C_1, \varpi_1, \varpi_2)$ we associate the following lexicographic triple:

$$\varrho(C_1, \varpi_1, \varpi_2) = \langle q(C_1) + q(\varpi_1) + q(\varpi_2), |\varpi_1| + |\varpi_2|, |\widehat{C_1}(\varpi_1)| + |\widehat{C_1}(\varpi_2)| \rangle$$

where $|[D] \Rightarrow \rho| = 1 + |\rho|$ and $|\sigma|$ is an ordinary size function on types. We proceed to show that eventually in every path of recursive calls, the metric ϱ on the arguments reduces.

1. For *eqCheck*, case E1 is trivial. Case E2 and case E3 follow by inlining the proof for the *updateRigid* function. For E4 we have that $eqCheck(C_1, \sigma_1 \to \sigma_2, \sigma_3 \to \sigma_4)$ relies on a call to $eqCheck(C_1, \sigma_1, \sigma_3)$. First of all we know that $\varrho(C_1, \sigma_1, \sigma_3) < \varrho(C_1, \sigma_1 \to \sigma_2, \sigma_3 \to \sigma_4)$ since the quantifier rank is preserved but the sizes of types become smaller. It follows that either $eqCheck(C_1, \sigma_1, \sigma_3)$ either *fail*s, or returns $E = eqCheck(C_1, \sigma_1, \sigma_3)$. In the first case we are trivially done. In the second case, by Lemma A.1.17 we have that $q(E) \leq q(C_1)$. It follows that $\varrho(E, \sigma_2, \sigma_4) < \varrho(C_1, \sigma_1 \to \sigma_2, \sigma_3 \to \sigma_4)$, and induction hypothesis finishes the case because it must be that $eqCheck(E, \sigma_2, \sigma_4) \Downarrow$. If it *fail*s, the whole case *fail*s, or if it succeeds we return the output of $eqCheck(E, \sigma_2, \sigma_4)$. In all other cases of *eqCheck* we *fail* (and hence terminate) using E6.
2. For *subsCheck*, case S1 follows by inlining the proof of the case of *updateFlexi*. For case S2 we have a call to $subsCheck(C_1, [D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2)$. The first step $\rho_3 = \overline{[b \mapsto c]}\rho_2$ clearly terminates. Then we examine the metric: $\varrho(C_1 \bullet D, \rho_1, \rho_3) < \varrho(C_1, [D] \Rightarrow \rho_1, \rho_3)$ and clearly the constraint

$C_1 \bullet D$ is acyclic and closed with the variables of ρ_1 and ρ_3 in its domain. Induction hypothesis give us then that the recursive call to *eqCheck* always terminates, and so does the original call to *subsCheck*.

3. For *updateRigid*, the case UR1 is trivial. For the case of UR2 we have that

$$updateRigid(C_1, \alpha, \beta) = updateRigid(C_1, \alpha, \gamma)$$

where $(\beta_\mu = \gamma) \in C_1$. Hence we have that

$$\varrho(C_1, \alpha, \gamma) = \langle q(C_1), 2, |\widehat{C_1}(\alpha)| + |\widehat{C_1}(\beta)| \rangle$$

But notice that $\beta \notin \widehat{C_1}(\gamma)$ because the constraint is acyclic, and hence: $|\widehat{C_1}(\gamma)| < |\widehat{C_1}(\beta)|$. It follows that $\varrho(C_1, \alpha, \gamma) < \varrho(C_1, \alpha, \beta)$, and hence the case terminates. The case for UR3 is similar. Finally, the case for UR3 follows by inlining the proof for *doUpdateR*.

4. For *doUpdateR*, the cases DR1 and DR2 are trivial by an easy termination lemma on acyclic constraints for *mkMono*. For case DR3 we have that $(\alpha_\mu \geq \varsigma_a) \in C_1$ and we have a call to $doUpdateF(C_1, \alpha, \sigma)$. We recursively call $subsCheck(C_1, \varsigma, \sigma)$. Now, it is not obvious that this terminates. However let us consider $\overline{\gamma} = \widehat{C_1}(\varsigma_a, \sigma)$ and $C_{1\overline{\gamma}}$. It must be that $C_{1\overline{\gamma}}$ is closed and acyclic and a subset of C_1, and moreover $(\alpha_\mu \geq \varsigma_a) \notin C_{1\overline{\gamma}}$ since α is not reachable from ς_a nor σ through C_1. It follows that $subsCheck(C_{1\overline{\gamma}}, \varsigma, \sigma)$ does terminate by induction hypothesis because: $\varrho(C_{1\overline{\gamma}}, \varsigma_a, \sigma) < \varrho(C_1, \alpha, \sigma)$ for the reason that $(\alpha_\mu \geq \varsigma_a) \notin C_{1\overline{\gamma}}$. By Theorem A.1.19 that the call to $subsCheck(C_1, \varsigma, \sigma)$ must also terminate.
5. For *updateFlexi* the cases are similar to *updateRigid*.
6. For *doUpdateF* the cases are similar to *doUpdateR*, appealing to Theorem A.1.19.
7. For *join* the cases J1 and J2 are obvious. For J3 observe that $q(C_1) + q([D_1] \Rightarrow \rho_1) + q([D_2] \Rightarrow \rho_2) = q(C_1 \bullet D_1 \bullet D_2) = q(C_1 \bullet D_1 \bullet D_2) + q(\rho_1) + q(\rho_2)$. however $|[D_1] \Rightarrow \rho_1| + |[D_2] \Rightarrow \rho_2| = 2 + |\rho_1| + |\rho_2|$ and hence, by induction hypothesis the recursive call to *eqCheck* terminates, and hence the whole case J3 terminates.

□

A.2 Unification soundness

We start by giving a series of lemmas that ensure that unification produces well-formed outputs, when given well-formed inputs.

Lemma A.2.1. *Assume that* $\vdash C$ *and* $\overline{\alpha}$ *is some set of variables. If* $\overline{\gamma} = \widehat{C}(\overline{\alpha})$ *then also* $\vdash C_{\overline{\gamma}}$.

Proof. Easy unfolding of the definitions. □

Definition A.2.2 (Mono-flags not in scheme bounds)**.** We write $\texttt{mweak}(C)$ for the smallest relation that asserts that for all $(\alpha_\mu \geq [D] \Rightarrow \rho) \in C$, it is the case that $\mu = \star$ and $\texttt{mweak}(D)$.

Definition A.2.3 (Well-flagged constraints)**.** Consider the relations $\Delta \vdash^\mu C$ and $\Delta \vdash^\mu \varsigma$ defined by:

$$\frac{(a)\ dom(C) \# dom(\Delta) \quad (b)\ \text{for all } (\alpha_{\mathsf{m}}\ bnd) \in C, \text{for all } \beta \in fcv(bnd), \Delta\Delta_c(\beta) = \mathsf{m} \text{ and } (bnd = \bot \text{ or } bnd = \tau) \quad (c)\ \text{for all } (\alpha_\mu \geq \varsigma) \in C, \Delta\Delta_C \vdash^\mu \varsigma \wedge \mu = \star}{\Delta \vdash^\mu C}\ \text{SMWF}$$

$$\frac{\Delta \vdash^\mu D}{\Delta \vdash^\mu [D] \Rightarrow \rho}\ \text{SMSCH}$$

We write $\vdash^\mu C$ whenever $\emptyset \vdash^\mu C$.

Lemma A.2.4. *If* $\vdash^\mu C_1$, $mkMono(C_1, f, \sigma) = C_2$, $\texttt{closed}(C_1)$, *and* $fcv(\sigma) \subseteq dom(C_1)$ *then it is the case that* $\vdash^\mu C_2$, *and moreover* $\Delta_{C_2}(fcv(\sigma)) = \mathsf{m}$. *Additionally, if* $f = True$ *then* $\sigma = \tau$.

Proof. Induction on the number of recursive calls. Case M1 is trivial. For the case of M2 we have that $mkMono(C_1, f, \sigma_1 \to \sigma_2) = C_2$ where $E = mkMono(C_1, f, \sigma_1)$ and $C_2 = mkMono(E, f, \sigma_2)$. By induction hypothesis we have that $\vdash^\mu E$ and $\Delta_E(fcv(\sigma_1)) = \mathsf{m}$. By Lemma A.1.7, and Lemma A.1.3 we have that $\texttt{closed}(E)$ and $fcv(\sigma_2) \subseteq dom(E)$. Hence, by induction $\vdash^\mu C_2$ and $\Delta_{C_2}(fcv(\sigma_2)) = \mathsf{m}$. But, by Lemma A.1.5 we also have that $\Delta_{C_2}(fcv(\sigma_1)) = \mathsf{m}$. Consequently, $\Delta_{C_2}(fcv(\sigma_1 \to \sigma_2)) = \mathsf{m}$, as required. Cases M3 and M4 are trivial. For case M5 we have that $mkMono(C_1, f, \alpha) = E \leftarrow (\alpha_{\mathsf{m}} = \rho)$ where $E = mkMono(C_1 \bullet D, True, \rho)$ and $(\alpha_\mu \geq [D] \Rightarrow \rho) \in C_1$. First of all, since $\vdash^\mu C_1$,

$\mu = \star$. Moreover it must be that $\vdash^\mu C_1 \bullet D$ and $\texttt{closed}(C_1 \bullet D)$. By induction then $\vdash^\mu E$ and $\Delta_E(fcv(\rho)) = \mathsf{m}$ and $\rho = \tau$. It follows from these last three conditions that $\vdash^\mu E \leftarrow (\alpha_{\mathsf{m}} = \rho)$. The case for M6 is similar. Case M7 is an easy check. □

Lemma A.2.5. *The following are true:*

1. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1)$, $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$, *and* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $\vdash^\mu C_2$.
2. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1, \varsigma)$, $\Delta_{C_1} \vdash^\mu \varsigma$, *and* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then* $\vdash^\mu C_2$.
3. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1)$, $fcv(\alpha, \sigma) \subseteq dom(C_1)$, *and* $updateRigid(C_1, \alpha, \sigma) = C_2$ *then* $\vdash^\mu C_2$.
4. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1)$, $fcv(\alpha, \sigma) \subseteq dom(C_1)$, $\alpha \notin \widehat{C_1}(\sigma)$, *and it is the case that* $doUpdateR(C_1, \alpha, \sigma) = C_2$ *then* $\vdash^\mu C_2$.
5. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1, \varsigma)$, $fcv(\varsigma, \alpha) \subseteq dom(C_1)$, $\Delta_{C_1} \vdash^\mu \varsigma$, *and it is the case that* $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then* $\vdash^\mu C_2$.
6. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1, \varsigma)$, $fcv(\varsigma, \alpha) \subseteq dom(C_1)$, $\alpha \notin \widehat{C_1}(\varsigma)$, $\Delta_{C_1} \vdash^\mu \varsigma$, *and it is the case that* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then* $\vdash^\mu C_2$.
7. *If* $\vdash^\mu C_1$, $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1, \varsigma_1, \varsigma_2)$, $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$, $\Delta_{C_1} \vdash^\mu \varsigma_1, \varsigma_2$, *and it is the case that* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$ *then* $\vdash^\mu C_2$ *and* $\Delta_{C_2} \vdash^\mu \varsigma$.

Proof. We show the cases simultaneously by induction on the number of recursive calls, assuming that all cases are true for calls with smaller number of recursive sub-calls.

1. Case E1 is trivial, cases E2 and E3 follow by induction hypothesis for *updateRigid*. The case of E4 follows by induction hypothesis, Theorem A.1.9, Lemma A.1.4, and a second use of induction hypothesis. Case E5 follows directly by induction hypothesis.
2. Case S1 follows by induction hypothesis for *updateFlexi*. For the case of S2, since $\texttt{closed}(C_1)$, $\texttt{acyclic}(C_1)$, and $fcv(\varsigma) \in dom(C_1)$, we have that $\vdash^\mu C_1 \bullet D$, and moreover $\texttt{closed}(C_1 \bullet D)$ and $\texttt{acyclic}(C_1 \bullet D)$. It follows by induction that $\vdash^\mu E$ and consequently it is straightforward to verify that $\vdash^\mu E_{\overline{\gamma}}$ where $\overline{\gamma} = \widehat{E}(dom(C_1))$.

3. Case UR1 is trivial. Cases UR2 and UR3 follow because the constraint is closed, and appealing to the induction hypothesis for *updateRigid*. Case UR4 follows by induction hypothesis for *doUpdateR*.
4. The case for DR1 follows from Lemma A.2.4. The case for DR2 follows by the assumptions directly. For case DR3 we have that $\mathit{doUpdateR}(C_1, \alpha, \sigma) = E \leftarrow (\alpha_\mu = \sigma)$ where $(\alpha_\mu \geq \varsigma_a) \in C_1$ and $E = \mathit{subsCheck}(C_1, \varsigma_a, \sigma)$. Because $\texttt{closed}(C_1)$ it must be that $\mathit{fcv}(\varsigma_a, \varsigma) \subseteq \mathit{dom}(C_1)$. Moreover, it must also be that $\mu = \star$ because $\vdash^\mu C_1$. Hence, by induction hypothesis $\vdash^\mu E$. By the separation theorem (taking as the set of separating variables the reachable variables through C_1 of ς_a, σ) it must be that $(\alpha_\mu \geq \varsigma_a) \in E$. and hence $\vdash^\mu E \leftarrow (\alpha_\mu = \sigma)$ (i.e. it is not the case that $\Delta_E(\alpha) = \mathsf{m}$, in which case the final update would potentially break the $\vdash^\mu$ relation). Case DR4 follows by induction hypothesis.
5. Case UF1 follows by induction hypothesis for *updateRigid*. Case UF2 follows by induction hypothesis for *doUpdateF*.
6. Case DF1 is similar to the case for DR1. Case DF2 is trivial Case DF3 follows by induction hypothesis for *subsCheck*. Finally let us examine case DF4. We have that $\mathit{doUpdateF}(C_1, \alpha, \varsigma) = E \leftarrow (\alpha_\mu \geq \varsigma)$, where $E, \varsigma_r = \mathit{join}(C_1, \varsigma, \varsigma_r)$ and $(\alpha_\mu \geq \varsigma_r) \in C_1$. It must be that $\mu = \star$, and by induction $\vdash^\mu E$. To finish the case we need to show that $\vdash^\mu E \leftarrow (\alpha_\star \geq \varsigma_r)$. By separation it must be that $(\alpha_\mu \geq \varsigma) \in E$, so we did not un-lift some monomorphic flag from E. Hence we must show that $\Delta_{E \leftarrow (\alpha_\star \geq \varsigma_r)} \vdash^\mu \varsigma_r$. But we know that $\Delta_E \vdash^\mu \varsigma_r$ by induction, and by separation $\alpha \notin \mathit{fcv}(\varsigma_r)$. It follows that $\Delta_{E \leftarrow (\alpha_\star \geq \varsigma_r)} \vdash^\mu \varsigma_r$.
7. Cases J1 and J2 are trivial. For J3 it is the case that

$$\mathit{join}(C_1, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) = E_{\widehat{E}(\mathit{dom}(C_1))}, [E_{\overline{\delta}}] \Rightarrow \rho_1$$

where $E = \mathit{eqCheck}(C_1 \bullet D_1 \bullet D_2, \rho_1, \rho_2)$ and $\overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(\mathit{dom}(C_1))$. Clearly the constraint $C_1 \bullet D_1 \bullet D_2$ satisfies $\vdash^\mu$, and is closed and acyclic. Consequently by induction hypothesis $\vdash^\mu E$, and hence $\vdash^\mu E_{\widehat{E}(\mathit{dom}(C_1))}$. We need to additionally show that $\Delta_{E_{\widehat{E}(\mathit{dom}(C_1))}} \vdash^\mu [E_{\overline{\delta}}] \Rightarrow \rho_1$, which follows because it is easy to see that $\Delta_{E_{\widehat{E}(\mathit{dom}(C_1))}} \vdash^\mu E_{\overline{\delta}}$ (since $\vdash^\mu E$).

□

At this point we have shown all the structural properties of the constraints being preserved by unification. But we have not talked yet about (i) the fact that schemes are not allowed to bind monomorphic variables (the intuition being that these always originate in some environment variable, and (ii) the fact that if the body of a scheme is a single variable, it is either monomorphic, or it is quantified with $\bot$ in the current scheme.

Definition A.2.6 (Weak well-formedness)**.** A constraint C_1 is weakly well-formed, written with the notation $\texttt{wfweak}(C_1)$, iff $\texttt{closed}(C_1) \wedge \texttt{acyclic}(C_1) \wedge \vdash^{\mu} C_1$.

Lemma A.2.7. *The following are true:*

1. *If* $\texttt{wfweak}(C_1)$*,* $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$*,* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$*, and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
2. *If* $\texttt{wfweak}(C_1)$*,* $\texttt{acyclic}(\varsigma)$*,* $\Delta_{C_1} \vdash^{\mu} \varsigma$*,* $fcv(\varsigma, \sigma) \subseteq dom(C_1)$*,* $subsCheck(C_1, \varsigma, \sigma) = C_2$*, and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
3. *If* $\texttt{wfweak}(C_1)$*,* $fcv(\alpha, \sigma) \subseteq dom(C_1)$*,* $updateRigid(C_1, \alpha, \sigma) = C_2$*, and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
4. *If* $\texttt{wfweak}(C_1)$*,* $fcv(\alpha, \sigma) \subseteq dom(C_1)$*,* $\alpha \notin \widehat{C_1}(\sigma)$*,* $doUpdateR(C_1, \alpha, \sigma) = C_2$*, and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
5. *If* $\texttt{wfweak}(C_1)$*,* $\texttt{acyclic}(\varsigma)$*,* $\Delta_{C_1} \vdash^{\mu} \varsigma$*,* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$*,* $updateFlexi(C_1, \alpha, \varsigma) = C_2$*, and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
6. *If* $\texttt{wfweak}(C_1)$*,* $\texttt{acyclic}(\varsigma)$*,* $\Delta_{C_1} \vdash^{\mu} \varsigma$*,* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$*,* $\alpha \notin \widehat{C_1}(\varsigma)$*, and additionally it is* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
7. *If* $\texttt{wfweak}(C_1)$*,* $\texttt{acyclic}(\varsigma_1, \varsigma_2)$*,* $\Delta_{C_1} \vdash^{\mu} \varsigma_1, \varsigma_2$*,* $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$*,* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$*, and* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.

Proof. Simultaneous induction on the number of recursive calls; assuming that all cases are true for smaller number of recursive calls. We do not show explicitly the $\vdash^{\mu}$, $\texttt{closed}(\cdot)$, and $\texttt{acyclic}(\cdot)$ properties are preserved, nor that the variables of the arguments are preserved in the domains of the outputs, as these follows whenever they are needed by Lemma A.2.5, Theorem A.1.9, Theorem A.1.15, and Lemma A.1.4.

1. Case E1 is trivial. Cases E2 and E3 follows directly from induction hypothesis for *updateRigid*.

For E4 we have that $eqCheck(C_1, \sigma_1 \to \sigma_2, \sigma_3 \to \sigma_4) = C_2$ where $eqCheck(C_1, \sigma_1, \sigma_3) = E$ and $eqCheck(E, \sigma_2, \sigma_4) = C_2$. Assume that $\Delta_{C_1}(\gamma) = \mathsf{m}$. By induction $\widehat{C_1}(\gamma) \subseteq \widehat{E}(\gamma)$. Moreover from Lemma A.1.6 we get that $\Delta_E(\gamma) = \mathsf{m}$. It follows by induction that $\widehat{E}(\gamma) \subseteq \widehat{C_2}(\gamma)$. Consequently $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$, as required. Case E5 follows by induction hypothesis, and case E6 cannot happen.

2. Case S1 follows by induction hypothesis for *updateFlexi*. For S2 we have that $\texttt{wfweak}(C_1)$, $\texttt{acyclic}([D] \Rightarrow \rho_1)$, $\Delta_{C_1} \vdash^{\mu} [D] \Rightarrow \rho_1$, $subsCheck(C_1, [D] \Rightarrow \rho_1, \forall \overline{c}.\rho_2) = E_{\overline{\gamma}}$ where $\rho_3 = [\overline{c \mapsto b}]\rho_2$, $E = eqCheck(C_1 \bullet D, \rho_1, \rho_3)$ and $\gamma = \widehat{E}(dom(C_1))$. It follows by induction hypothesis, if $\Delta_{C_1 \bullet D}(\gamma) = \mathsf{m}$ then $\widehat{C_1 \bullet D}(\gamma) \subseteq \widehat{E}(\gamma)$. Assume now that $\Delta_{C_1}(\gamma) = \mathsf{m}$. It follows that $\gamma \in dom(C_1)$. But also $\Delta_{C_1 \bullet D}(\gamma) = \mathsf{m}$. Hence $\widehat{C_1 \bullet D}(\gamma) = \widehat{C_1}(\gamma)$ since $dom(D) \# fcv(C_1)$. Then $\widehat{C_1}(\gamma) \subseteq \widehat{E}(\gamma)$. But $\gamma \in dom(C_1)$ and consequently $\widehat{E}(\gamma) = \widehat{E_{\overline{\gamma}}}(\gamma)$. It follows that $\widehat{C_1}(\gamma) \subseteq \widehat{E_{\overline{\gamma}}}(\gamma)$ as required.
3. Case UR1 is trivial. For case UR2 we have that $\texttt{wfweak}(C_1)$, and $updateRigid(C_1, \alpha, \beta) = C_2$ where $C_2 = updateRigid(C_1, \alpha, \gamma)$ and $(\beta_\mu = \gamma) \in C_1$. By induction hypothesis for *updateRigid* we are done. The case of UR3 is similar, and the case of UR4 follows by induction hypothesis for *doUpdateR*.
4. For case DR1 we have that $doUpdateR(C_1, \alpha, \sigma) = E \leftarrow (\alpha_{\mathsf{m}} = \sigma)$ where $(\alpha_{\mathsf{m}}\ \bot) \in C_1$ and $E = mkMono(True, C_1, \sigma)$. Assume that $\Delta_{C_1}(\gamma) = \mathsf{m}$. By an easy similar property for *mkMono* it follows that $\widehat{C_1}(\gamma) \subseteq \widehat{E}(\gamma)$. But by the separation theorem Theorem A.1.13 (by using as the separating set of variables the set $\widehat{C_1}(\sigma)$) we know that $(\alpha_{\mathsf{m}}\ \bot) \in dom(E)$. Consequently $\widehat{E}(\gamma) \subseteq (E \leftarrow \widehat{(\alpha_{\mathsf{m}} = \sigma)})(\gamma)$ as required. Case DR2 is straightforward. For case DR3 we have that $doUpdateR(C_1, \alpha, \sigma) = C_2$ where $(\alpha_\star \geq \varsigma_a) \in C_1$ (since $\vdash^{\mu} C_1$)it must be that $\mu = \star$), $E = subsCheck(C_1, \varsigma_a, \sigma)$, and $C_2 = E \leftarrow (\alpha_\star = \sigma)$. By the separation theorem, Theorem A.1.14, it must be that $(\alpha_\star \geq \varsigma_a) \in E$, but we also have that $\Delta_E(\gamma) = \mathsf{m}$ and hence $\alpha \notin \widehat{E}(\gamma)$. By induction hypothesis we get that if $\Delta_{C_1}(\gamma) = \mathsf{m}$, $\widehat{C_1}(\gamma) \subseteq \widehat{E}(\gamma)$. But because $\vdash^{\mu} E$ by Lemma A.2.5 and since $\Delta_E(\gamma) = \mathsf{m}$, it must be that $\widehat{E}(\gamma) = E \leftarrow \widehat{(\alpha_\star = \sigma)}(\gamma)$. and we are done. Case DR4 follows by induction hypothesis.
5. The cases for *updateFlexi* are similar to the cases for *updateRigid*.
6. The cases for *doUpateF* are similar to the cases for *doUpdateR*.

7. Cases J1 and J2 follow trivially. For case J3 we have that

$$join(C_1, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) = (E_{\widehat{E}(dom(C_1))}, [E_\delta] \Rightarrow \rho_1)$$

given that $E = eqCheck(C_1 \bullet D_1 \bullet D_2, \rho_1, \rho_2)$, and $\overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(dom(C_1))$. Assume that $\Delta_{C_1}(\gamma) = \mathsf{m}$. Then also $\Delta_{C_1 \bullet D_1 \bullet D_2}(\gamma) = \mathsf{m}$. By induction hypothesis we then get that $\widehat{C_1}(\gamma) = \widehat{C_1 \bullet D_1 \bullet D_2}(\gamma) \subseteq \widehat{E}(\gamma) = \widehat{E_{\widehat{E}(dom(C_1))}}(\gamma)$, as required.

□

Definition A.2.8 (Quantified variable condition)**.** We define the relation $\texttt{wfqvar}(C)$ and similarly $\texttt{wfqvar}(\varsigma)$ as:

$$\frac{\text{for all } (\alpha_\mu \geq \varsigma) \in C, \texttt{wfqvar}(\varsigma)}{\texttt{wfqvar}(C)}\ \text{CWFQ} \qquad \frac{\texttt{wfqvar}(D) \quad \text{for all } \alpha \in dom(D), \Delta_D(\alpha) = \star}{\texttt{wfqvar}([D] \Rightarrow \rho)}\ \text{SWFQ}$$

Lemma A.2.9. *The following are true:*

1. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1)$ *and* $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$ *and* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $\texttt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.
2. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1, \varsigma)$ *and* $\texttt{acyclic}(\varsigma)$ *and* $\Delta_{C_1} \vdash^\mu \varsigma$ *and* $fcv(\varsigma, \sigma) \subseteq dom(C_1)$ *and* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then* $\texttt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.
3. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1)$ *and* $fcv(\alpha, \sigma) \subseteq dom(C_1)$ *and* $updateRigid(C_1, \alpha, \sigma) = C_2$ *then* $\texttt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.
4. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1)$ *and* $fcv(\alpha, \sigma) \subseteq dom(C_1)$ *and* $\alpha \notin \widehat{C_1}(\sigma)$ *and additionally it is the case that* $doUpdateR(C_1, \alpha, \sigma) = C_2$ *then* $\texttt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.
5. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1, \varsigma)$ *and* $\texttt{acyclic}(\varsigma)$ *and* $\Delta_{C_1} \vdash^\mu \varsigma$ *and* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$ *and* $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then it is the case that* $\texttt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.

6. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1, \varsigma)$ *and* $\texttt{acyclic}(\varsigma)$ *and* $\Delta_{C_1} \vdash^{\mu} \varsigma$ *and* $fcv(\alpha, \varsigma) \subseteq dom(C_1)$ *and* $\alpha \notin \widehat{C_1}(\varsigma)$ *and moreover* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then* $\texttt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.
7. *If* $\texttt{wfweak}(C_1)$ *and* $\texttt{wfqvar}(C_1, \varsigma_1, \varsigma_2)$ *and* $\texttt{acyclic}(\varsigma_1, \varsigma_2)$ *and* $\Delta_{C_1} \vdash^{\mu} \varsigma_1, \varsigma_2$ *and* $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$ *and* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$ *then* $\texttt{wfqvar}(C_2, \varsigma)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \widehat{C_2}(\beta)$.

Proof. We prove all the cases simultaneously by induction on the number of recursive calls. For each case we assume that they all hold for sub-calls. In the recursive cases we make use of Lemma A.2.5, Theorem A.1.9, Theorem A.1.15 to show the necessary well-formedness conditions. In the case of E4 we need to appeal to Lemma A.2.7. All the rest cases are straightforward except for the cases for S2 and J3. We focus on these particular cases:

- Case S2. In this case we have that $\texttt{wfweak}(C_1)$, $\texttt{wfqvar}(C_1, \varsigma)$, $\texttt{acyclic}(\varsigma)$, and $fcv(\varsigma, \sigma) \subseteq dom(C_1)$, where $\varsigma = [D] \Rightarrow \rho_1$ and $\sigma = \forall \overline{c}\,.\,\rho_2$. Moreover $subsCheck(C_1, [D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2) = E_{\overline{\gamma}}$ where $\rho_3 = [\overline{c \mapsto b}]\rho_2$, $E = eqCheck(C_1 \bullet D, \rho_1, \rho_2)$ and $\overline{\gamma} = \widehat{E}(dom(C_1))$. We have that $\texttt{wfweak}(C_1 \bullet D)$, $\texttt{wfqvar}(C_1 \bullet D)$, $fcv(\rho_1, \rho_2) \subseteq dom(C_1 \bullet D)$. Hence by induction hypothesis we have that $\texttt{wfqvar}(E)$ and consequently $\texttt{wfqvar}(E_{\overline{\gamma}})$ as well. Now, pick a variable γ such that $\Delta_{E_{\overline{\gamma}}}(\gamma) = \mathsf{m}$. By induction hypothesis there exists a $\beta \in dom(C_1 \bullet D)$ such that $\Delta_{C_1 \bullet D}(\beta) = \mathsf{m}$ and $\gamma \in \widehat{E_{\overline{\gamma}}}(\beta)$. But $\texttt{wfqvar}([D] \Rightarrow \rho_1)$ and hence $\Delta_{C_1}(\beta) = \mathsf{m}$, and the case is finished.
- Case J3. In this case we have that $\texttt{wfweak}(C_1)$ and $\texttt{acyclic}([D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2)$ and $\texttt{wfqvar}([D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2)$ and $\texttt{wfqvar}(C_1)$. Moreover $join(C_1, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) = E_{\widehat{E}(dom(C_1))}, [E_{\overline{\delta}}] \Rightarrow \rho_1$ where $E = eqCheck(C_1 \bullet D_1 \bullet D_2, \rho_1, \rho_2)$, $\overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(dom(C_1))$. First, we can confirm that $\texttt{wfweak}(C_1 \bullet D)$ and $\texttt{wfqvar}(C_1 \bullet D_1 \bullet D_2)$, and additionally it is the case that $fcv(\rho_1, \rho_2) \subseteq dom(C_1 \bullet D_1 \bullet D_2)$. Hence, by induction hypothesis $\texttt{wfqvar}(E)$ and consequently also $\texttt{wfqvar}(E_{\widehat{E}(dom(C_1))})$. Next, we need to show that $\texttt{wfqvar}([E_{\overline{\delta}}] \Rightarrow \rho_1)$. Hence, we have to show two things:
  - For all $\gamma \in dom(E_{\overline{\delta}})$, $\Delta_{E_{\overline{\delta}}} = \star$. Assume by contradiction that there exists a γ such that $\Delta_{E_{\overline{\delta}}}(\gamma) = \mathsf{m}$. It follows that $\Delta_E(\gamma) = \mathsf{m}$. By induction then there exists a β such that

$\Delta_{C_1 \bullet D_1 \bullet D_2}(\beta) = \mathsf{m}$ such that $\gamma \in \widehat{E}(\beta)$. But it must be the case that $\Delta_{C_1}(\beta) = \mathsf{m}$ because $\mathtt{wfqvar}([D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2)$, and hence $\gamma \notin \overline{\delta}$, a contradiction!

– That $E_{\overline{\delta}}$ itself satisfies $\mathtt{wfqvar}(E_{\overline{\delta}})$. But we know that $\mathtt{wfqvar}(E)$ and $E_{\overline{\delta}}$ is only a restriction of E.

To finish the case assume a variable ζ such that $\Delta_{E_{\widehat{E(dom(C_1))}}}(\zeta) = \mathsf{m}$. It follows that $\Delta_E(\zeta) = \mathsf{m}$ and by induction there exists a variable δ such that $\Delta_{C_1 \bullet D_1 \bullet D_2}(\delta) = \mathsf{m}$ such that $\zeta \in \widehat{E}(\delta)$. But, since $\mathtt{wfqvar}([D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2)$ it must be the case that $\Delta_{C_1}(\delta) = \mathsf{m}$ and $\zeta \in \widehat{E_{\widehat{E(dom(C_1))}}}(\delta)$ as required.

□

Definition A.2.10 (Well-formed scheme bodies)**.** Consider the relations $\Delta \vdash^b C$ and $\Delta \vdash^b \varsigma$ whenever:

$$\frac{(a)\ dom(C) \# dom(\Delta) \qquad (b)\ \text{for all } (\alpha_\mu \geq \varsigma) \in C, \Delta\Delta_C \vdash^b \varsigma}{\Delta \vdash^b C}\ \text{CBWF}$$

$$\frac{\Delta \vdash^b D \qquad (\rho = \gamma) \Longrightarrow ((\gamma_\star\ \bot) \in D) \vee (\Delta\Delta_D(\gamma) = \mathsf{m})}{\Delta \vdash^b [D] \Rightarrow \rho}\ \text{SBWF}$$

We write $\vdash^b C$ whenever $\emptyset \vdash^b C$.

Definition A.2.11 (Well-formedness)**.** We write $\mathtt{wf}(C_1)$ iff $\mathtt{wfweak}(C_1) \wedge \mathtt{wfqvar}(C_1) \wedge \vdash^b C_1$.

Lemma A.2.12. *If* $\mathtt{wf}(C_1)$*,* $fcv(\sigma_1) \subseteq dom(C_1)$*,* $mkMono(C_1, f, \sigma) = C_2$ *then* $\vdash^b C_2$*.*

Proof. Easy induction appealing to previous lemmas for *mkMono*: A.1.3, A.2.4, A.1.7, A.1.12. □

Lemma A.2.13 (Well-formed scheme bodies preservation)**.** *The following are true:*

1. *If* $\mathtt{wf}(C_1)$*,* $fcv(\sigma_1, \sigma_2) \subseteq dom(C_1)$*,* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $\vdash^b C_2$*.*
2. *If* $\mathtt{wf}(C_1)$*,* $\mathtt{acyclic}(\varsigma)$*,* $\mathtt{wfqvar}(\varsigma)$*,* $\Delta_{C_1} \vdash^{b/\mu} \varsigma$*,* $fcv(\varsigma, \sigma) \subseteq dom(C_1)$*,* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then* $\vdash^b C_2$*.*
3. *If* $\mathtt{wf}(C_1)$*,* $fcv(\alpha, \sigma) \subseteq dom(C_1)$*,* $updateRigid(C_1, \alpha, \sigma) = C_2$ *then* $\vdash^b C_2$*.*

4. *If* $\texttt{wf}(C_1)$, $fcv(\alpha, \sigma) \subseteq dom(C_1)$, $\alpha \notin \widehat{C_1}(\sigma)$, $doUpdateR(C_1, \alpha, \sigma) = C_2$ *then* $\vdash^b C_2$.
5. *If* $\texttt{wf}(C_1)$, $\texttt{wfqvar}(\varsigma)$, $\texttt{acyclic}(\varsigma)$, $\Delta_{C_1} \vdash^{b/\mu} \varsigma$, $fcv(\alpha, \varsigma) \subseteq dom(C_1)$, $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then* $\vdash^b C_2$.
6. *If* $\texttt{wf}(C_1)$, $\texttt{wfqvar}(\varsigma)$, $\texttt{acyclic}(\varsigma)$, $\Delta_{C_1} \vdash^{b/\mu} \varsigma$, $fcv(\alpha, \varsigma) \subseteq dom(C_1)$, $\alpha \notin \widehat{C_1}(\varsigma)$, *and additionally it is the case that* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then* $\vdash^b C_2$.
7. *If* $\texttt{wf}(C_1)$, $\texttt{acyclic}(\varsigma_1, \varsigma_2)$, $\texttt{wfqvar}(\varsigma_1, \varsigma_2)$, $\Delta_{C_1} \vdash^{b/\mu} \varsigma_1, \varsigma_2$, $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$, *and additionally it is the case that* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$ *then* $\vdash^b C_2$ *and* $\Delta_{C_2} \vdash^b \varsigma$.

Proof. The proof is by simultaneous induction on the number of recursive calls. For each case we assume that the corresponding case is true for the recursive sub-calls. Most cases are straightforward, except for the cases of *join*. The cases for J1 and J2 are straightforward. For J3 we have that $\texttt{wf}(C_1)$, $\texttt{acyclic}(\varsigma_1, \varsigma_2)$, $\texttt{wfqvar}(\varsigma_1, \varsigma_2)$, $\Delta_{C_1} \vdash^{b/\mu} \varsigma_1, \varsigma_2$, $fcv(\varsigma_1, \varsigma_2) \subseteq dom(C_1)$ and $\varsigma_1 = [D_1] \Rightarrow \rho_1$, $\varsigma_2 = [D_2] \Rightarrow \rho_2$. We also have that $join(C_1, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) = E_{\widehat{E}(dom(C_1))}, [E_{\overline{\delta}}] \Rightarrow \rho_1$ where $E = eqCheck(C_1 \bullet D_1 \bullet D_2, \rho_1, \rho_2)$, $\overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(dom(C_1))$. By induction it is easy to show that $\vdash^b E$, and consequently $\vdash^b E_{\widehat{E}(dom(D_1))}$ (since it is a closed constraint). To finish the case we need to show that $\Delta_{E_{\widehat{E}(dom(D_1))}} \vdash^b [E_{\overline{\delta}}] \Rightarrow \rho_1$. First of all clearly $\Delta_{E_{\widehat{E}(dom(C_1))}} \vdash^b E_{\overline{\delta}}$. Assume now that $\rho_1 = \gamma$. Then, it must have been that either $(\gamma_\star \perp) \in D_1$, in which case we are done (because this branch of the algorithm cannot have been taken). Or it must be that $\Delta_{C_1}(\gamma) = \mathsf{m}$. In which case it must be by preservation of the monomorphic domain (Lemma A.1.6) that also $\Delta_{E_{\widehat{E}(dom(C_1))}}(\gamma) = \mathsf{m}$, as is required to finish the case. □

Corollary A.2.14. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \sigma$, *and* $mkMono(C_1, f, \sigma) = C_2$ *then* $\vdash C_2$.

Proof. The proof is by putting together the following Lemmas or Theorems: A.1.3, A.1.7, A.1.12, A.2.4, and A.2.12. □

Corollary A.2.15 (Well-formedness preservation)**.** *The following are true:*

1. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \sigma_1, \sigma_2$, *and* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *then* $\vdash C_2$.
2. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma, \sigma$, $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then* $\vdash C_2$.
3. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \sigma$, $updateRigid(C_1, \alpha, \sigma) = C_2$ *then* $\vdash C_2$.

4. *If* $\vdash C_1$, $\alpha \notin \widehat{C_1}(\sigma)$, $\Delta_{C_1} \vdash \alpha, \sigma$, *and* $doUpdateR(C_1, \alpha, \sigma) = C_2$ *then* $\vdash C_2$.
5. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \varsigma$, $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then* $\vdash C_2$.
6. *If* $\vdash C_1$, $\alpha \notin \widehat{C_1}(\varsigma)$, $\Delta_{C_1} \vdash \alpha, \varsigma$, *and* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then* $\vdash C_2$.
7. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma_1, \varsigma_2$, $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma_r$ *then* $\vdash C_2$ *and* $\Delta_{C_2} \vdash \varsigma_r$.

Proof. The proof is by putting together the following Lemmas or Theorems: A.1.9, A.1.15, A.2.5, A.2.9, and A.2.13.

□

Up to this point we have shown that unification preserves our (rather complex) notion of well-formed constraints. For the rest of this section, we turn to the actual soundness of unification proofs, which relies—for the inductive cases to go through—on Corollary A.2.14 and Corollary A.2.15. We additionally make two extra assumptions:

1. In every constraint of the form $[D] \Rightarrow \rho$ encountered in the algorithm, all quantified variables of D ($dom(D)$) appear in the reachable variables $\widehat{D}(\rho)$ (i.e. there exist no useless quantified variables in schemes). Rule J3 which is the only rule that creates quantification clearly preserves this invariant.
2. An additional requirement is that no System F types with unused quantified variables (such as $\forall ab.\gamma$ or $\forall ab.a \to a$) are encountered during type inference, and that our substitutions never substitute for types with unused quantified variables.

Definition A.2.16 (Logical constraint entailment)**.** We write $C_1 \vdash C_2$ iff for every $\theta \models C_1$ it is $\theta \models C_2$.

Lemma A.2.17. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \sigma$ *and* $mkMono(C_1, f, \sigma) = C_2$ *then* $C_2 \vdash C_1$. *Moreover* $\Delta_{C_2}(fcv(\sigma)) = \mathsf{m}$ *and if* $f = True$ *then* $\sigma = \tau$.

Proof. The second part follows from Lemma A.2.4. We show the first goal by induction on the number of recursive calls to *mkMono*. Case M1 is straightforward. Case M2 follows by the induction hypothesis, Corollary A.2.14 and Lemma A.1.3. Case M3 and case M4 are straightforward. For M5 we have that $mkMono(C_1, f, \alpha) = E \leftarrow (\alpha_{\mathsf{m}} = \rho)$ given that $(\alpha_\mu \geq [D] \Rightarrow \rho) \in C_1$ and

$E = mkMono(True, C_1 \bullet D, \rho)$. Assume that $\theta \models E \leftarrow (\alpha_{\mathsf{m}} = \rho)$. By separation consider the constraint $C_0 = (C_1 \bullet D)_{\widehat{C_1 \bullet D}(\rho)}$ and C_r the rest such that $C_0 C_r = C_1 \bullet D$ and then $E = E_0 C_r$. However it must be that $\alpha \in dom(C_r)$. Then $\theta \models E_0$ and by induction (the call with C_0 as the starting constraint takes the same number of recursive calls and clearly $\vdash C_0$) $\theta \models C_0$. Additionally $\theta \models C_r - \alpha$. Consequently $\theta \models C_0(C_r - \alpha)$, that is $\theta \models C_1 \bullet D - \alpha$. It follows (since $\alpha \notin dom(D)$) that $\theta \models D$. Then $\theta\alpha = \theta\rho \in [\![\theta([D^\circ] \Rightarrow \rho^\circ)]\!]$, where the $\circ$ symbol denotes a freshening of the domain of D and ρ to fresh constrained variables. That is because $\theta\alpha$ must be monomorphic, and hence it suffices to find a substitution $\psi \models \theta(D^\circ)$ and such that $\theta\rho = \psi\theta(\rho^\circ)$. But simply take $\psi = \theta^\circ_{dom(D)}$ (i.e. the restriction of θ to the $dom(D)$ variables, where we rename its domain as necessary). Hence, $\theta \models C_1 \bullet D$ and $\theta \models C_1$ as required. Case M6 is similar, and case M7 is straightforward. □

Lemma A.2.18 (Satisfiability of well-formed constraints)**.** *If $\vdash D$ then there always exists a θ such that $\theta \models D$.*

Proof. Simply instantiate all flexible bounds to convert them to equalities. The result is an acyclic equality constraint which is always solvable by ordinary first-order unification (by first creating a unifier for the m-flagged variables). □

We prove the rest of the lemmas in this section simultaneously by induction on the number of the steps that the algorithm performs. For each lemma we assume that all other lemmas are true for any sub-calls of the algorithm.

Lemma A.2.19. *If $\vdash C_1$, $\Delta_{C_1} \vdash \sigma_1, \sigma_2$, $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$, for all θ such that $\theta \models C_2$ it is the case that $\theta \models C_1$ and $\theta\sigma_1 = \theta\sigma_2$.*

Proof. The case for E1 is straightforward. Case E2 and E3 follow from induction hypothesis for Lemma A.2.20. For case E4 the result follows from induction hypothesis, Corollary A.2.15, Lemma A.1.4, and a second use of induction hypothesis. Case E5 gives us that

$$eqCheck(C_1, \forall \overline{a} \,.\, \rho_1, \forall \overline{a} \,.\, \rho_2) = C_2$$

where we have that $C_2 = eqCheck(C_1, \overline{[a \mapsto b]}\rho_1, \overline{[a \mapsto b]}\rho_2)$ for some fresh $\overline{b}$ and additionally $\overline{b} \# C_2$. Assume that $\theta \models C_2$. Induction hypothesis gives $\theta \models C_1$ and $\theta\overline{[a \mapsto b]}\rho_1 = \theta\overline{[a \mapsto b]}\rho_2$. Let us assume without loss of generality that $\overline{a} \# ftv(\theta)$. Now we know that $\overline{b} \# C_2$ and $\theta \models C_2$. Let $\theta_0 = \overline{[b \mapsto d]}\theta$, where $\overline{d}$ completely fresh. By an easy renaming property we see that $\theta_0 \models C_2$. Then by induction hypothesis also $\theta_0\overline{[a \mapsto b]}\rho_1 = \theta_0\overline{[a \mapsto b]}\rho_2$, therefore $\forall\overline{b}.\theta_0\overline{[a \mapsto b]}\rho_1 = \forall\overline{b}.\theta_0\overline{[a \mapsto b]}\rho_2$. Now $\overline{b} \# ftv(\theta_0)$ and hence we have that $\theta_0(\forall\overline{b}.\overline{[a \mapsto b]}\rho_1) = \theta_0(\forall\overline{b}.\overline{[a \mapsto b]}\rho_2)$. Then, $\theta_0(\forall\overline{a}.\rho_1) = \theta_0(\forall\overline{a}.\rho_2)$. Finally since neither $\forall\overline{a}.\rho_1$ or $\forall\overline{a}.\rho_2$ contain $\overline{b}$ or $\overline{d}$, by another renaming we get: $\theta(\forall\overline{a}.\rho_1) = \theta(\forall\overline{a}.\rho_2)$, as required to finish the case. □

Lemma A.2.20. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \sigma$, $updateRigid(C_1, \alpha, \sigma) = C_2$, *for all* θ *such that* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $\theta\alpha = \theta\sigma$.

Proof. Case UR1 is trivial. For UR2 we have that $updateRigid(C_1, \alpha, \beta) = C_2$ where $(\beta_\mu = \gamma) \in C_1$ and $C_2 = updateRigid(C_1, \alpha, \gamma)$. Assume $\theta \models C_2$. By induction $\theta\alpha = \theta\gamma$ and $\theta \models C_1$. Since $(\beta_\mu = \gamma) \in C_1$ it must be that $\theta\beta = \theta\gamma$. It follows as well that $\theta\alpha = \theta\beta$. For case UR3 we have that $updateRigid(C_1, \alpha, \beta) = C_2$ where $(\beta_\mu \geq [D] \Rightarrow \gamma) \in C_1$, $\gamma \notin dom(D)$, and $C_2 = updateRigid(C_1, \alpha, \gamma)$. Assume that $\theta \models C_1$ and $\theta\alpha = \theta\gamma$. Moreover, since $\gamma \notin dom(D)$ we know that $\Delta_{C_1}(\gamma) = \mathsf{m}$. It follows that $\theta\gamma$ is monomorphic. Hence we have that $\theta\beta \in [\![[\theta D] \Rightarrow \theta\gamma]\!]$ (because θD must be empty). It follows that (assuming that our substitutions add no useless quantifiers) that $\theta\beta = \theta\gamma$. Consequently $\theta\alpha = \theta\beta$ as required. Case UR4 follows from the inductive hypothesis for Lemma A.2.21. □

Lemma A.2.21. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \sigma$, $\alpha \notin \widehat{C_1}(\sigma)$, $doUpdateR(C_1, \alpha, \sigma) = C_2$, *for all* θ *such that* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $\theta\alpha = \theta\sigma$.

Proof. We have the following cases to consider:

- Case DR1. In this case we have $doUpdateR(C_1, \alpha, \sigma) = C_2$ where $(\alpha_\mathsf{m}\ \bot) \in C_1$. Moreover $E = mkMono(True, C_1, \sigma)$ and $C_2 = E \leftarrow (\alpha_\mathsf{m} = \sigma)$. Assume that $\theta \models E \leftarrow (\alpha_\mathsf{m} = \sigma)$. It must be by the separation theorem however that $(\alpha_\mathsf{m}\ \bot) \in E$. Consequently, $\theta\alpha = \theta\sigma$ and $\theta \models E$. By Lemma A.2.17, $\theta \models C_1$ and we are done.

- Case DR2. We have that $doUpdateR(C_1, \alpha, \sigma) = C_1 \leftarrow (\alpha_\star = \sigma)$ where $(\alpha_\star \bot) \in C_1$. Assume that $\theta \models C_1 \leftarrow (\alpha_\star = \sigma)$. It follows that $\theta \models C_1$ since $\theta\alpha$ can be any type in order to satisfy the $(\alpha_\star \bot)$ bound in C_1. Moreover $\theta\alpha = \theta\sigma$ as required.
- Case DR3. We have $doUpdateR(C_1, \alpha, \sigma)$ where $(\alpha_\mu \geq \varsigma_a) \in C_1$, $E = subsCheck(C_1, \varsigma_a, \sigma)$, and $C_2 = E \leftarrow (\alpha_\mu = \sigma)$. By well-formedness of the constraint C_1 we know that $\mu = \star$. Assume that $\theta \models E \leftarrow (\alpha_\mu = \sigma)$. consider the restriction of C_1 to the reachable variables of ς_a, σ through C_1, call it C_0. The $C_1 = C_0\, C_r$ and $E = E_0\, C_r$ such that $E_0 = subsCheck(C_0, \varsigma_a, \varsigma)$ in the same number of steps (by Theorem A.1.14). Since $\alpha \notin dom(E_0)$ it follows that $\theta \models E_0$ and by induction hypothesis $\theta \models C_0$ and $\theta\sigma \in [\![\theta\varsigma]\!]$. But also $\theta \models E - \alpha$, that is $\theta \models E_0(C_r - \alpha)$ and hence $\theta \models C_0(C_r - \alpha) = C_1 - \alpha$. But since $\theta\sigma \in [\![\theta\varsigma]\!]$ it follows that $\theta \models C_1$. Finally $\theta\alpha = \theta\sigma$, because $\theta \models E \leftarrow (\alpha_\mu = \sigma)$.
- Case DR4. We have that $doUpdateR(C_1, \alpha, \sigma) = C_2$ where $(\alpha_\mu = \sigma_a) \in C_1$ and $C_2 = eqCheck(C_1, \sigma_a, \sigma)$. Assume $\theta \models C_2$. By induction hypothesis for Lemma A.2.19 it follows that $\theta \models C_1$ and $\theta\sigma_a = \theta\sigma$. Since $(\alpha_\mu = \sigma_a) \in C_1$ it must be that $\theta\alpha = \theta\sigma_a$. It follows that $\theta\alpha = \theta\sigma$ as required.

□

Lemma A.2.22. *If* $\vdash C_1$ *and* $\Delta_{C_1} \vdash \varsigma, \sigma$*, and* $subsCheck(C_1, \varsigma, \sigma) = C_2$ *then for all* θ *such that* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $\theta\sigma \in [\![\theta\varsigma]\!]$*.*

Proof. The case for S1 follows by induction hypothesis for Lemma A.2.23. The case for S2 is the interesting one. Assume that $subsCheck(C_1, [D] \Rightarrow \rho_1, \forall \overline{c}\,.\,\rho_2) = C_2$ where $\rho_3 = [\overline{c \mapsto b}]\rho_2$, $E = eqCheck(C_1 \bullet D, \rho_1, \rho_3)$, $\overline{\gamma} = \widehat{E}(dom(C_1))$, $\overline{b} \# E_{\overline{\gamma}}$, and $C_2 = E_{\overline{\gamma}}$. By the well-formedness assumptions it must be that $\vdash C_1 \bullet D$ and moreover $C_1 \bullet D \vdash \rho_1, \rho_3$, assuming that $\overline{b}$ are fresh enough.

Assume now that $\theta \models E_{\overline{\gamma}}$. Consider the restriction of θ to $\overline{\gamma}$, call it $\theta_{\overline{\gamma}}$. Consider the permutation of $\theta_{\overline{\gamma}}$ such that maps $\overline{b}$ to completely fresh variables $\overline{d}$, call it $\theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}}$. It follows since $\overline{b} \# E_{\overline{\gamma}}$ that $\theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}} \models E_{\overline{\gamma}}$. Since every well-formed constraint is satisfiable (Lemma A.2.18), there is an extension θ_1, such that $\theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}} \theta_1 \models E$. Then by induction hypothesis we get that $\theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}} \theta_1(\rho_1) = \theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}} \theta_1(\rho_3)$. Since $fcv(\rho_3) \in \overline{\gamma}$ this means that $\theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}} \theta_1(\rho_1) = \theta_{\overline{\gamma}}^{\overline{b} \mapsto \overline{d}}(\rho_3)$. However induction also gives that

$\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\theta_1 \models C_1 \bullet D$. Let $\theta_{\overline{\delta}} = \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\theta_1 \mid_{dom(D)}$. Then $\theta_{\overline{\delta}} \models \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\theta_1(D)$. To finish the case we need to show that $\theta(\forall\overline{c}\,.\,\rho_2) \in [\![[\theta D] \Rightarrow \theta(\rho_1)]\!]$. We know that:

$$\theta_{\overline{\delta}}\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_1 = \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_3 \tag{A.13}$$

$$\overline{b}\#([\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}D^\star] \Rightarrow \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_1^\star) \tag{A.14}$$

where $D^\star$ and $\rho_1^\star$ have their $\overline{\delta} = dom(D)$ renamed to fresh $\overline{\delta}^\star$. The second holds because of the choice of choosing fresh $\overline{b}$. Moreover, by simply renaming the $\overline{\delta}$ to $\overline{\delta}^\star$ in the first equation we get

$$\theta_{\overline{\delta}^\star}\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_1^\star = \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_3$$

Hence, it follows that

$$\forall\overline{b}\,.\,\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}[\overline{b \mapsto c}]\rho_2 \in [\![[\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}D^\star] \Rightarrow \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_1^\star]\!]$$

But $\forall\overline{b}\,.\,\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}[\overline{b \mapsto c}]\rho_2 = \theta^{\overline{b}\mapsto\overline{d}}(\forall\overline{c}\,.\,\rho_2)$ and $[\![[\theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}D^\star] \Rightarrow \theta^{\overline{b}\mapsto\overline{d}}_{\overline{\gamma}}\rho_1^\star]\!] = [\![\theta^{\overline{b}\mapsto\overline{d}}([D] \Rightarrow \rho_1)]\!]$ and consequently: $\theta^{\overline{b}\mapsto\overline{d}}(\forall\overline{c}\,.\,\rho_2) \in [\![\theta^{\overline{b}\mapsto\overline{d}}([D] \Rightarrow \rho_1)]\!]$. By a renaming, because of the fresh choice of $\overline{b}$ of the algorithm, we get $\theta(\forall\overline{c}\,.\,\rho_2) \in [\![\theta([D] \Rightarrow \rho_1)]\!]$, as required. □

Lemma A.2.23. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma, \alpha$, *and* $updateFlexi(C_1, \alpha, \varsigma) = C_2$ *then for all* θ *such that* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $\theta\alpha \in [\![\theta\varsigma]\!]$.

Proof. Similar to the proof of Lemma A.2.20, appealing to inductive hypotheses for Lemma A.2.20 and Lemma A.2.24. □

Lemma A.2.24. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma, \alpha$, $\alpha \notin \widehat{C_1}(\varsigma)$, *and* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *then for all* θ *such that* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $\theta\alpha \in [\![\theta\varsigma]\!]$.

Proof. Similar to the proof of Lemma A.2.21. The only interesting case is the call to the *join* function, which we describe next. We have that $doUpdateF(C_1, \alpha, \varsigma) = C_2$ where $(\alpha_\mu \geq \varsigma_a) \in C_1$, $E, \varsigma_r = join(C_1, \varsigma_a, \varsigma)$, and $C_2 = E \leftarrow (\alpha_\mu \geq \varsigma_r)$. Consider the split of C_1 to $C_0\,C_r$ such that C_0 contains all the reachable variables through C_1 of ς_a and ς. By separation it must be that $join(C_0, \varsigma_a, \varsigma) = E_0, \varsigma_r$, where $E = E_0\,C_r$. It must be that $\alpha \in dom(C_r)$, and also $\theta \models E_0$, and by

induction $\theta \models C_0$. Consequently $\theta \models C_0 \bullet C_r - \alpha$ and hence $\theta \models C_1 - \alpha$. However by induction for Lemma A.2.25 also $[\![\theta\varsigma_r]\!] \subseteq [\![\theta\varsigma]\!] \cap [\![\theta\varsigma_a]\!]$. Because $\theta\alpha \in [\![\theta\varsigma_r]\!] \subseteq [\![\theta\varsigma_a]\!]$ it must be that $\theta \models C_1$. Additionally $\theta\alpha \in [\![\theta\varsigma_r]\!] \subseteq [\![\theta\varsigma]\!]$, or $\theta\alpha \in [\![\theta\varsigma]\!]$ as required to finish the case. □

Lemma A.2.25. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma_1, \varsigma_2$, *and* $join(C_1, \varsigma_1, \varsigma_2) = C_2, \varsigma$ *then for all* θ *such that* $\theta \models C_2$ *it is the case that* $\theta \models C_1$ *and* $[\![\theta\varsigma]\!] \subseteq [\![\theta\varsigma_1]\!] \cap [\![\theta\varsigma_2]\!]$.

Proof. We have the following cases to consider:

- $join(C_1, [D] \Rightarrow \alpha, \varsigma_2) = (C_1, \varsigma_2)$ whenever $(\alpha_\mu \perp) \in D$. We know moreover by well-formedness of $[D] \Rightarrow \alpha$ that $\mu = \star$, and $D = \emptyset$ (no useless quantified variables). Hence it is adequate to finish the case to show that $[\![\theta\varsigma]\!] \subseteq [\![[\theta D] \Rightarrow \alpha]\!]$. This will be the case if $[\![[\theta D] \Rightarrow \alpha]\!]$ is the set of all types, which is the case, since the θD can always be satisfied by any substitution, and in fact one that can assign any type σ to α (because $(\alpha_\star \perp) \in \theta D$).
- $join(C_1, \varsigma_1, [D] \Rightarrow \alpha) = (C_1, \varsigma_1)$ whenever $(\alpha_\mu \perp) \in D$. The case is similar to the previous one.
- $join(C_1, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2) = C_2, \varsigma$, where $E = eqCheck(C_1 \bullet D_1 \bullet D_2, \rho_1, \rho_2)$, and moreover $\overline{\gamma} = \widehat{E}(dom(C_1))$, $\overline{\delta} = \widehat{E}(\rho_1) - \overline{\gamma}$, and $C_2 = E_{\overline{\gamma}}$, $\varsigma = [E_{\overline{\delta}}] \Rightarrow \rho_1$. First of all $C_1 \bullet D_1 \bullet D_2$ is well formed. Assume $\theta \models E_{\overline{\gamma}}$. Then there is an extension θ_1 such that $\theta\theta_1 \models E$. By induction $\theta\theta_1 \models C_1 \bullet D_1 \bullet D_2$ but since $dom(\theta) \supseteq dom(E_{\overline{\gamma}}) \supseteq dom(C_1)$ and C_1 is closed, it must be that $\theta \models C_1$ as required. Now consider θ_{D_1} and θ_{D_2} to be the restrictions of $\theta\theta_1$ to $dom(D_1)$ and $dom(D_2)$ respectively. Assume $\forall \overline{c} \,.\, \rho \in [\![\theta([E_{\overline{\delta}}] \Rightarrow \rho_1)]\!]$ hence $\forall \overline{c} \,.\, \rho \in [\![[\theta E^\star_{\overline{\delta}}] \Rightarrow \theta\rho^\star_1]\!]$ where $\overline{\delta}^\star$ and $\rho^\star_1$ are a suitable renaming of $\overline{\delta}$ to some fresh variables. Then it must be that:

$$\overline{c} \# [\theta E^\star_{\overline{\delta}}] \Rightarrow \theta\rho^\star_1 \tag{A.15}$$

$$\rho = \theta^\star_{\overline{\delta}} \theta\rho^\star_1 \tag{A.16}$$

$$\theta^\star_{\overline{\delta}} \models \theta E^\star_{\overline{\delta}} \tag{A.17}$$

  for some suitable $\theta^\star_{\overline{\delta}}$. We need to show that: $\forall \overline{c} \,.\, \rho \in [\![[\theta D^\circ_1] \Rightarrow \theta\rho^\circ_1]\!]$ where D°_1 and ρ°_1 are completely fresh renamings of the domain of D_1. We may pick from the beginning $\overline{c}$ to be fresh from $[\![[\theta D^\circ_1] \Rightarrow \theta\rho^\circ_1]\!]$. Consider now the substitution $\theta^\circ_{D_1}$. It must be that $\theta^\circ_{D_1} \models \theta D^\circ_1$. Moreover $\theta^\circ_{D_1}\theta(\rho^\circ_1) = \theta^\star_{\overline{\delta}}\theta\rho^\star_1 = \rho$. Hence $[\![\theta\varsigma]\!] \subseteq [\![\theta([D_1] \Rightarrow \rho_1)]\!]$. For showing $[\![\theta\varsigma]\!] \subseteq [\![\theta([D_2] \Rightarrow \rho_2)]\!]$, it

suffices to show that $\forall \overline{c}\,.\,\rho \in [\![[\theta D_2^\circ] \Rightarrow \theta\rho_2^\circ]\!]$ where D_2° and ρ_2° are completely fresh renamings of the domain of D_2. We may pick from the beginning $\overline{c}$ to be fresh from $[\![[\theta D_2^\circ] \Rightarrow \theta\rho_2^\circ]\!]$. Consider now the substitution $\theta_{D_2}^\circ$. It must be that $\theta_{D_2}^\circ \models \theta D_2^\circ$. To conclude we must show that $\theta_{D_2}^\circ\theta(\rho_2^\circ) = \rho$. It suffices to show that $\theta_{D_2}^\circ\theta(\rho_2^\circ) = \theta\overset{\star}{\underset{\delta}{}}\theta\rho_1^\star$ which holds since $\theta\theta_1\rho_1 = \theta\theta_1\rho_2$ by induction.

□

A.3 Unification completeness

We proceed with showing the completeness of unification with respect to the set semantics that we have defined earlier.

Lemma A.3.1. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \sigma_1, \sigma_2$, $\theta \models C_1$, *with* $dom(\theta) = dom(C_1)$, $\theta\sigma_1 = \theta\sigma_2$, *and eqCheck terminates, then* $eqCheck(C_1, \sigma_1, \sigma_2) = C_2$ *such that there exists a* θ_r *with* $\theta\theta_r \models C_2$ *and* $dom(\theta\theta_r) = dom(C_2)$.

Proof. The proof is by induction on termination relation for *eqCheck*. Let us consider cases on σ_1 and σ_2. If both types are (rigid) type variables, then it must be that $\sigma_1 = a$ and $\sigma_2 = a$, and consequently case E1 is applicable. If σ_1 or σ_2 is α then, case E2 or case E3 is applicable , and the result follows by Lemma A.3.3. If none of σ_1 or σ_2 are variables, assume that they are function types: $\sigma_1 = \sigma_{11} \to \sigma_{21}$ and $\sigma_2 = \sigma_{12} \to \sigma_{22}$. Then we have that $\theta\sigma_{11} = \theta\sigma_{12}$ and $\theta\sigma_{21} = \theta\sigma_{22}$. Hence by induction hypothesis we have that $eqCheck(C_1, \sigma_{11}, \sigma_{21}) = E$ such that there exists a θ_r^1 with $\theta\theta_r^1 \models E$. Moreover by the well-formedness lemmas we know that E is well formed, and moreover $\Delta_E \vdash \sigma_{21}, \sigma_{22}$. By induction hypothesis again $eqCheck(E, \sigma_{21}, \sigma_{22}) = C_2$, and there exists a θ_r^2 such that $\theta\theta_r^1\theta_r^2 \models C_2$ as required. The final case we have to consider is when $\sigma_1 = \forall\overline{a}\,.\,\rho_1$ and $\sigma_2 = \forall\overline{a}\,.\,\rho_2$, and without loss of generality assume that $\rho_1 \neq \gamma_1$ and $\sigma_2 \neq \gamma_2$ because we assume that there exist no useless quantified variables in types. Also without loss of generality assume that $\overline{a}\overline{b}\#ftv(\theta)$ (i.e. we picked the fresh variables, fresh from θ as well). Then $\theta\sigma_1 = \forall\overline{a}\,.\theta\rho_1$ and $\theta\sigma_2 = \forall\overline{a}\,.\theta\rho_2$. It follows that $\theta\rho_1 = \theta\rho_2$. Consequently by induction $eqCheck(C_1, [\overline{a \mapsto b}]\rho_1, [\overline{a \mapsto b}]\rho_2) = C_2$ such that there exists a θ_r so that $\theta\theta_r \models C_2$. To finish the case we need to show that $\overline{b}\#E$. But observe

that it suffices to show that $\overline{b}\#E_{\widehat{E}(dom(C_1))}$ (by the separation lemma, the rest of the constraint E is also present in C_1 and not touched, and hence cannot have contained $\overline{b}$). If there were some $b \in \overline{b}$ in $E_{\widehat{E}(dom(C_1))}$ it must have been that $b \in range(\theta)$, a contradiction. Case E6 is not applicable, because the types cannot be equal. □

Lemma A.3.2. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma, \sigma$, *and* $\theta \models C_1$ *with* $dom(\theta) = dom(C_1)$, $\theta\sigma \in [\![\theta\varsigma]\!]$ *and subsCheck terminates, then subsCheck*$(C_1, \varsigma, \sigma) = C_2$ *then there exists a* θ_r *such that* $\theta\theta_r \models C_2$ *and* $dom(\theta\theta_r) = dom(C_2)$.

Proof. Let us do a case analysis on σ.

- $\sigma = \beta$. In this case the result follows by Lemma A.3.5.
- $\sigma \neq \beta$. In this case $\sigma = \forall\overline{c}\,.\,\rho_2$ where we assume that ρ_2 is not a variable. Since *subsCheck* terminates, then it must also be that *eqCheck*$(C_1 \bullet D, \rho_1, \rho_3)$ terminates where $\rho_3 = [\overline{c \mapsto b}]\rho_2$. Since $\theta(\forall\overline{c}\,.\,\rho_2) = [\![[\theta D] \Rightarrow \theta\rho]\!]$. We assume without loss of generality that $\overline{b}\#ftv(\theta)$ (i.e. we picked the variables $\overline{b}$ fresh from θ as well). Then there must be a substitution θ_D such that $\theta_D \models \theta D$ such that $\theta_D\theta\rho = \theta\rho_2$ (because $\theta\rho_2$ must be a ρ-type when ρ_2 is not a variable), and moreover $\overline{c}\#ftv([\theta D] \Rightarrow \theta\rho)$. It follows that $\theta\theta_D \models C_1 \bullet D$. By Lemma A.3.1, since *eqCheck*$(C_1 \bullet D, \rho_1, \rho_3)$ terminates, there exists a θ_r such that $\theta\theta_D\theta_r \models E$. Finally consider the restriction of $\theta_D\theta_r$, call it $\theta_r^\star$, such that $\theta\theta_r^\star \models E_{\overline{\gamma}}$. To finish the case we need to show that $\overline{b}\#E_{\overline{\gamma}}$. This follows, because if $\overline{b} \in E_{\overline{\gamma}}$ it must be that $\overline{b} \in range(\theta)$, which is impossible.

□

Lemma A.3.3. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \sigma$, $\theta \models C_1$, *with* $dom(\theta) = dom(C_1)$, $\theta\alpha = \theta\sigma$, *and updateRigid terminates, then updateRigid*$(C_1, \alpha, \sigma) = C_2$ *such that there exists a* θ_r *such that* $\theta\theta_r \models C_2$ *and additionally it is the case that* $dom(\theta\theta_r) = dom(C_2)$.

Proof. We consider cases on σ. If $\sigma = \beta$ we have the following cases:

- We have $\beta = \alpha$. The case finishes by taking θ_r to be the identity substitution.

- We have $\beta \neq \alpha$ and $(\beta_\mu = \gamma) \in C$. It follows that case UR2 is reachable. However since $\theta\alpha = \theta\beta$ and $\theta \models C$, it must be that $\theta\alpha = \theta\gamma$. Induction hypothesis finishes the case.
- We have $\beta \neq \alpha$ and $(\beta_\mu \geq [D] \Rightarrow \rho_1)$. Moreover we know that $\theta\alpha = \theta\beta$ and $\theta\beta \in [\![[\theta D] \Rightarrow \theta\rho_1]\!]$. We have the following cases for ρ_1. If $\rho_1 = \gamma$ then one case is if $(\gamma_\star \perp) \in D$. In that case, case UR4 is reachable and the result follows by Lemma A.3.4. The other case is that $\Delta_{C_1}(\gamma) = \mathsf{m}$. In this case, case UR3 is reachable (and also $D = \emptyset$). Then, since $\theta\beta \in [\![[\emptyset] \Rightarrow \theta\gamma]\!]$ and there are no useless quantifiers, we have that $\theta\beta = \theta\gamma$ (and is monomorphic). But then it must be that *updateRigid*(C_1, α, γ) terminates, and hence by induction hypothesis we are done. Now, let us consider the cases where $\rho_1 \neq \gamma$. It must be in that case that $\alpha \notin \widehat{C_1}(\beta)$ otherwise it can't happen that $\theta\alpha = \theta\beta \in [\![[\theta D] \Rightarrow \theta\rho_1]\!]$. Hence, case UR4 is reachable and we get the result by Lemma A.3.4.

Now, assume that $\sigma \neq \beta$. It must be the case that $\alpha \notin \widehat{C_1}(\sigma)$, because otherwise it cannot be that $\theta\alpha = \theta\sigma$. Then the case UR4 is reachable and Lemma A.3.4 shows the goal. □

Lemma A.3.4. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \sigma$, $\theta \models C_1$, *with* $dom(\theta) = dom(C_1)$, $\alpha \notin \widehat{C_1}(\sigma)$, $\theta\alpha = \theta\sigma$, *and doUpdateR terminates, then doUpdateR*$(C_1, \alpha, \sigma) = C_2$ *such that there exists a* θ_r *with* $\theta\theta_r \models C_2$ *and* $dom(\theta\theta_r) = dom(C_2)$.

Proof. The proof is by case analysis on the bound of α in C_1. If $(\alpha_m \perp) \in C_1$ and $\theta\alpha = \theta\sigma$, and moreover $\theta\alpha$ is monomorphic. By completeness for *mkMono*, we have that there exists a θ_r^1 such that $\theta\theta_r^1 \models E$. It follows that $\theta\theta_r^1 \models E \leftarrow (\alpha_\mathsf{m} = \sigma)$. If on the other hand $(\alpha_\star \perp) \in C_1$ we have as before that $\theta\alpha = \theta\sigma$ and $\theta \models C_1$, so it must also be that $\theta \models C_1 \leftarrow (\alpha_\star = \sigma)$. Now, consider the case where $(\alpha_\star \geq \varsigma_a) \in C_1$. We know that $\theta\alpha = \theta\sigma$ and moreover $\theta\alpha \in [\![\theta\varsigma_a]\!]$. It follows that case DR3 is reachable and by Lemma A.3.2 we have that *subsCheck*$(C_1, \varsigma_a, \sigma) = E$ and there exists a θ_r such that $\theta\theta_r \models E$. Since $\theta\alpha = \theta\sigma$ this implies also that $\theta\theta_r \models E \leftarrow (\alpha_\star = \sigma)$. Finally assume that $(\alpha_\mu = \sigma_a) \in C_1$. Moreover we have that $\theta\alpha = \theta\sigma$ and since $\theta \models C_1$ we have $\theta\alpha = \theta\sigma_a$. Then case DR4 is reachable and then by Lemma A.3.1 the case is finished. □

Lemma A.3.5. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \varsigma$, $\theta \models C_1$, *with* $dom(\theta) = dom(C_1)$, $\theta\alpha \in [\![\theta\varsigma]\!]$, *and updateFlexi terminates, then updateFlexi*$(C_1, \alpha, \varsigma) = C_2$ *such that there exists a* θ_r *such that* $\theta\theta_r \models C_2$ *and*

additionally it is the case that $dom(\theta\theta_r) = dom(C_2)$.

Proof. Similar to the proof of Lemma A.3.3, appealing to Lemma A.3.3, and Lemma A.3.6. (Observe that if $\varsigma = [D] \Rightarrow \gamma$ then by well-formedness γ is either a monomorphic variable, or bound to $\perp$ in D, hence the call to *updateRigid* in UF1). □

Lemma A.3.6. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \alpha, \varsigma$, $\theta \models C_1$, $\alpha \notin \widehat{C_1}(\alpha)$, *with* $dom(\theta) = dom(C_1)$, $\theta\alpha \in [\![\theta\varsigma]\!]$, *and doUpdateF terminates, then* $doUpdateF(C_1, \alpha, \varsigma) = C_2$ *such that there exists a* θ_r *with* $\theta\theta_r \models C_2$ *and* $dom(\theta\theta_r) = dom(C_2)$.

Proof. Similar to the proof of Lemma A.3.4, by analysis to the bound of α in C_1, and appealing to Lemma A.3.2. The interesting case is when $(\alpha_\star \geq \varsigma_a) \in C_1$. In this case we know that $\theta\alpha \in [\![\theta\varsigma]\!]$ and $\theta\alpha \in [\![\theta\varsigma_a]\!]$. Then we are in case DF4 and it follows by Lemma A.3.7 that $join(C_1, \varsigma, \varsigma_a) = E$, and there exists a θ_r such that $\theta\theta_r \models E$. From the same lemma it is the case that $\theta\alpha \in [\![\theta\theta_r\varsigma_r]\!]$. It follows that $\theta\theta_r \models E \leftarrow (\alpha_\star \geq \varsigma_r)$. □

Lemma A.3.7. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \varsigma_1, \varsigma_2$, $\theta \models C_1$ *with* $dom(\theta) = dom(C_1)$, *and* $join(C_1, \varsigma_1, \varsigma_2)$ *terminates and* $\sigma \in [\![\theta\varsigma_1]\!] \cap [\![\theta\varsigma_2]\!]$, *then* $join(C_1, \varsigma_1, \varsigma_2) = E, \varsigma$ *such that there exists a* θ_r *with* $\theta\theta_r \models E$ *and* $dom(\theta\theta_r) = \theta(E)$, *and* $\sigma \in [\![\theta\theta_r\varsigma]\!]$.

Proof. Let us consider cases on ς_1 and ς_2. If one of them is $[\alpha_\star \perp] \Rightarrow \alpha$ then either the case of J1 or J2 is reachable and, since the set $[\alpha_\star \perp] \Rightarrow \alpha$ is the set of all types we are done. Now, let us assume that this is not the case. The reachable clause is then J3. In this case we have that $\theta \models C_1$ and the call is made to $join(C_1, [D_1] \Rightarrow \rho_1, [D_2] \Rightarrow \rho_2)$. We have that $\sigma \in [\![[\theta D_1] \Rightarrow \theta\rho_1]\!]$ and $\sigma \in [\![\theta D_2] \Rightarrow \theta\rho_2]\!]$. Assume that $\sigma = \forall\overline{c}.\rho$. It must be hence that $\overline{c}\#ftv([\theta D_1] \Rightarrow \theta\rho_1, [\theta D_2] \Rightarrow \theta\rho_2)$ and there exist θ_{D_1} and θ_{D_2} such that $\theta_{D_1} \models \theta D_1$ and $\theta_{D_2} \models \theta D_2$. Without loss of generality assume that D_1 and D_2 have fresh domains, and it also must be that $\rho = \theta_{D_1}\theta\rho_1 = \theta_{D_2}\theta\rho_2$. Hence it must be that $\theta\theta_{D_1}\theta_{D_2} \models C_1 \bullet D_1 \bullet D_2$, and

$$\theta\theta_{D_1}\theta_{D_2}\rho_1 = \theta\theta_{D_1}\theta_{D_2}\rho_2 = \rho \tag{A.18}$$

Hence, by Lemma A.3.1 $eqCheck(C_1 \bullet D_1 \bullet D_2, \rho_1, \rho_2) = E$ such that there exists a θ_r for which it is the case that $\theta\theta_{D_1}\theta_{D_2}\theta_r \models E$. Consider the restriction of $\theta_{D_1}\theta_{D_2}\theta_r$ to $E_{\widehat{E}(dom(C_1))}$ to be $\theta_r^\star$ and then $\theta\theta_r^\star \models E_{\widehat{E}(dom(C_1))}$. If $\overline{\delta} = \widehat{E}(\rho_1) - \widehat{E}(dom(C_1))$ then it is also the case that $\theta\theta_{D_1}\theta_{D_2}\theta_r \models E_{\overline{\delta}}$ and equation (A.18) finishes the case. □

A.4 Main algorithm soundness

We start by showing some well-formedness preservation lemmas about the algorithm of Figure 6.3.

Lemma A.4.1 (Instantiation well-formedness)**.** *The following properties are true:*

1. *If* $inst(C_1, \sigma) = C_2, \rho$ *then* $dom(C_1) \subseteq dom(C_2)$ *and if* $\Delta_{C_1}(\gamma) = \mathsf{m}$ *then* $\Delta_{C_2}(\gamma) = \mathsf{m}$.
2. *If* $\mathtt{wfweak}(C_1)$ *and* $inst(C_1, \sigma) = C_2, \rho$ *and* $\Delta_C(\gamma) = \mathsf{m}$ *then* $\widehat{C_1}(\gamma) \subseteq \widehat{C_2}(\gamma)$.
3. *If* $\mathtt{wfweak}(C_1)$, $\mathtt{wfqvar}(C_1)$, $fcv(\sigma) \subseteq dom(C_1)$, $inst(C_1, \sigma) = C_2, \rho$ *then* $\mathtt{wfqvar}(C_2)$ *and if* $\Delta_{C_2}(\gamma) = \mathsf{m}$ *then there is a* $\beta \in dom(C_1)$ *such that* $\Delta_{C_1}(\beta) = \mathsf{m}$ *and* $\gamma \in \Delta_{C_2}(\beta)$.

Proof. Easy check. □

Lemma A.4.2 (Instantiation well-formedness)**.** *If* $\vdash C_1$ *and* $\Delta_{C_1} \vdash \sigma$ *and* $inst(C_1, \sigma) = C_2, \rho$ *then* $\vdash C_2$ *and* $C_2 \vdash \rho$.

Proof. Easy check. □

Lemma A.4.3 (Instantiation soundness)**.** *If* $\vdash C_1$ *and* $\Delta_{C_1} \vdash \sigma$, *and* $inst(C_1, \sigma) = C_2, \rho$, *then for all* $\theta \models C_2$, $\theta \models C_1$ *and* $\vdash^{\mathsf{inst}} \theta[\sigma] \leq \theta[\rho]$.

Proof. It must be that $\sigma = \forall \overline{a} \,.\, \rho_1$ in which case $C_2 = C_1 \bullet (\overline{\alpha}_\star \, \bot)$ and $\rho = [\overline{a \mapsto \alpha}]\rho_1$. Assume that $\theta \models C_2$. It follows that $\theta \models C_1$. Moreover let θ_o, $\theta_{\overline{\alpha}}$ be the split of θ such that $dom(\theta_{\overline{\alpha}}) = \overline{\alpha}$. Then $\theta[\sigma] = \forall \overline{a} \,.\, \theta_0[\rho_1]$ where without loss of generality we assume that $\overline{a} \# ftv(\theta)$. But then

$$\vdash^{\mathsf{inst}} \forall \overline{a} \,.\, \theta_0[\rho_1] \leq [\overline{a \mapsto \theta_{\overline{\alpha}}[\alpha]}]\theta_0[\rho_1] = \theta[([\overline{a \mapsto \alpha}]\rho_1)] = \theta\rho$$

as required. □

The following important lemma ensures that our normalization procedure produces well-formed schemes.

Lemma A.4.4 (Normalization well-formedness)**.** *Assume that:*

1. $\Delta \vdash C$,
2. $\Delta\Delta_C \vdash \sigma$,
3. *for all* $\alpha \in dom(C)$, $\Delta_C(\alpha) = \star$, *and there exists a* $\beta \in fcv(\sigma)$ *such that* $\alpha \in \widehat{C}(\sigma)$, *and*
4. $normalize(C, \sigma) = [E] \Rightarrow \rho$

Then, $\Delta \vdash [E] \Rightarrow \rho$.

Proof. We prove the lemma by induction on the number of recursive calls. We have the following cases to consider:

- Case N1. We have that $normalize(C, \alpha) = [C \bullet D - \alpha] \Rightarrow \rho$. Hence it must also be that $\Delta \vdash C \bullet D$, and because no variable from the domain of C points to α, and C and D are acyclic it follows that $\Delta \vdash C \bullet D - \alpha$. The interesting part is to verify two conditions, (i) the domain of $E = C \bullet D - \alpha$ does not contain any m-bound variables, and (ii) if $\rho = \gamma$ it is either bound to $\bot$ in E or monomorphic in the environment Δ. For (i), observe first that, because $\Delta \vdash [D] \Rightarrow \rho$, D does not contain any m-bound variables in its domain, and so does C by assumptions. It follows that $C \bullet D$ does not contain any m-bound variables, and so does $C \bullet D - \alpha$. For (ii) let us examine the case where $\rho = \gamma$, If $\gamma \in dom(D)$, since $\Delta \vdash [D] \Rightarrow \rho$ it must be that $(\gamma_\star \bot) \in dom(D)$ which implies that $(\gamma_\star \bot) \in dom(C \bullet D)$ and clearly $\gamma \neq \alpha$. On the other hand if $\Delta(\gamma) = \mathsf{m}$, $\gamma \notin dom(C)$ and $\gamma \notin dom(D)$ and hence $\gamma \notin dom(C \bullet D)$ and hence the case is done.
- Case N2 is similar to N1 appealing additionally to the induction hypothesis.
- Case N3. We have in this case that $normalize(C, \forall \overline{a} \,.\, \rho) = normalize(C \bullet (\overline{\alpha}_\star \bot), [\overline{a \mapsto \alpha}]\rho)$. We can assume without loss of generality that $\overline{\alpha}$ were picked fresh from Δ as well, and consequently $\Delta \vdash C \bullet (\overline{\alpha}_\star \bot)$. Moreover by assumptions $C \bullet (\overline{\alpha}_\star \bot)$ does not contain any m-bound variables. by induction then the case is finished.

- Case N4. In this case we have that either ρ is not a variable in which case the well-formedness follows easily. If on the other hand $\rho = \gamma$, since we know that, since we are in this case it must be that
  - either $(\gamma_\mu \perp) \in C$, and since C does not contain any m-bound variables, it is $(\gamma_\star \perp) \in C$,
  - or $\gamma \in \mathit{fcv}(\sigma)$ and $\gamma \notin \mathit{dom}(C)$. But in this case $\Delta(\gamma) = \mathsf{m}$ by assumptions.

  It follows in every case that $\Delta \vdash [E] \Rightarrow \rho$, as required.

□

Lemma A.4.5 (Normalization soundness)**.** *Assume that:*

1. $\vdash C$,
2. $\Delta_C \vdash C_1$,
3. $\Delta\Delta_{C_1} \vdash \sigma$,
4. *for all* $\alpha \in \mathit{dom}(C_1)$, $\Delta_{C_1} = \star$, *and there exists a* $\beta \in \mathit{fcv}(\sigma)$ *such that* $\alpha \in \widehat{C_1}(\sigma)$, *and*
5. $\mathit{normalize}(C_1, \sigma) = [E] \Rightarrow \rho$

Then, for all $\theta \models C \bullet E$ *there exists a* θ_r *such that* $\theta\theta_r \models C \bullet C_1$ *and* $\vdash^{\mathsf{F}} \theta\theta_r\sigma \leq \theta\rho$.

Proof. We show the case by induction on the number of recursive calls of *normalize*. In case N1 we have that $\mathit{normalize}(C_1, \alpha) = [C_1 \bullet D - \alpha] \Rightarrow \rho$ when $(\alpha_\star \geq [D] \Rightarrow \rho) \in C_1$. Assume that $\theta \models C_1 \bullet D - \alpha$. Then it must be that $\theta_D \models \theta^{-D} D$, ($\theta^{-D}$ being the restriction of θ to its domain that is disjoint from D), it suffices then to choose $\theta_r(\alpha) = \theta\rho$ and we are done. In case N2 we have that $\mathit{normalize}(C_1, \alpha) = \mathit{normalize}(C_1 - \alpha, \sigma)$ whenever $(\alpha_\star = \sigma) \in C_1$. Assume that $\mathit{normalize}(C_1 - \alpha, \sigma) = [E] \Rightarrow \rho$ and let $\theta \models C \bullet E$, then there exists a θ^1_r such that $\theta\theta^1_r \models C \bullet (C_1 - \alpha)$. Let $\theta^2_r(\alpha) = \theta\theta^1_r\sigma$. It follows that $\theta\theta^1_r\theta^2_r \models C \bullet C_1$ as required. Moreover we know that $\vdash^{\mathsf{F}} \theta\theta^1_r\sigma \leq \theta\rho$, which implies $\vdash^{\mathsf{F}} \theta\theta^1_r\theta^2_r(\alpha) \leq \theta\rho$. In case N3 we have that $\mathit{normalize}(C_1, \forall\overline{a}\,.\,\rho) = \mathit{normalize}(C_1 \bullet (\overline{\alpha}_\star \perp), [\overline{a \mapsto \alpha}]\rho) = [E] \Rightarrow \rho$. Assume that $\theta \models C \bullet E$, by induction there exists a θ_r such that $\theta\theta_r \models C \bullet C_1 \bullet (\overline{\alpha}_\star \perp)$, and $\vdash^{\mathsf{F}} \theta\theta_r[\overline{a \mapsto \alpha}]\rho \leq \theta\rho$. It follows that $\vdash^{\mathsf{F}} \theta\theta_r(\forall\overline{a}\,.\,\rho) \leq \theta\rho$. Case N4 is straightforward. □

Lemma A.4.6 (Generalization well-formedness)**.** *If* $\vdash C$, $C \vdash \Gamma, \rho$, *and* $generalize(C, \Gamma, \rho) = [E_g] \Rightarrow \rho_g, E$ *then* $\vdash E$, $E \vdash \Gamma$, *and* $E \vdash [E_g] \Rightarrow \rho_g$.

Proof. Unfolding of the definitions, appealing to Lemma A.4.4 □

Lemma A.4.7 (Generalization soundness)**.** *Assume that* $\vdash C$ *and* $C \vdash \Gamma$ *and* $C \vdash \rho$ *and moreover it is the case that* $generalize(C, \Gamma, \rho) = [E_g] \Rightarrow \rho_g, E$ *and* $\theta \models E$ *then, for all* θ_g *such that* $\theta\theta_g \models E \bullet E_g$ *there exists a* θ_r *such that* $\theta\theta_g\theta_r \models C$ *and* $\vdash^{\mathsf{F}} (\theta\theta_g\theta_r\rho) \leq \theta\theta_g\rho_g$.

Proof. Let $\overline{\beta} = \widehat{C}(\rho) - \widehat{C}(\Gamma)$. Then $[E_g] \Rightarrow \rho_g = normalize(C_{\overline{\beta}}, \rho)$, and $E = C - C_{\overline{\beta}}$. Assume that $\theta \models C - C_{\overline{\beta}}$, and assume that θ_g is such that $\theta\theta_g \models (C - C_{\overline{\beta}}) \bullet E_g$. By Lemma A.4.5 there exists a θ_r such that $\theta\theta_g\theta_r \models (C - C_{\overline{\beta}}) \bullet C_{\overline{\beta}}$, that is, $\theta\theta_g\theta_r \models C$. Moreover it is the case that $\vdash^{\mathsf{F}} \theta\theta_g\theta_r\rho \leq \theta\theta_g\rho_g$ as required. □

Given the fact that unification, normalization, and generalization preserve well-formedness of constraints and preserve the sets of monomorphic variables, it is an easy check that functions *instFun*, *instMono*, and *infer* have the same properties. In the rest of this section we assume domain monotonicity, preservation of well-formedness and origination of all monomorphic variables in some monomorphic variable appearing in the environment. Establishing this property is a tedious but straightforward task.

Below we give soundness of the functions *instFun*, *instMono*, and *infer*.

Lemma A.4.8. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \rho$, $C_1 \vdash \Gamma$, *and* $instFun(C_1, \Gamma, \rho) = C_2, \sigma_1 \to \sigma_2$ *then for every* $\theta \models C_2$, *there exists a* θ_r *such that* $\theta\theta_r \models C_1$ *and* $\theta\theta_r[\sigma] \preceq \sqsubseteq^{\to} \theta[\sigma_1] \to \theta[\sigma_2]$.

Proof. Let us consider two cases on σ. If $\rho = \sigma_1 \to \sigma_2$ we are trivially done. If on the other hand $\rho = \alpha$ we have that $instFun(C_1, \alpha) = (E_2, \beta \to \gamma)$ where $E, [E_g] \Rightarrow \rho_g = generalize(C_1, \Gamma, \alpha)$, and $E_2 = eqCheck(E \bullet E_g \bullet (\beta_\star\ \bot) \bullet (\gamma_\star\ \bot), \rho_g, \beta \to \gamma)$. Assume that $\theta \models E_2$. By Lemma A.2.19 we have $\theta \models E \bullet E_g \bullet (\beta_\star\ \bot) \bullet (\gamma_\star\ \bot)$. By Lemma A.4.7 we have that there exists a θ_r such that $\vdash^{\mathsf{F}} \theta\theta_r(\alpha) \leq \theta\rho_g$, and $\theta\theta_r \models C_1$ (the use of θ_g in the theorem is inlined inside θ), and $\theta(\beta) \to \theta(\gamma) = \theta\rho_g$. It follows that $\boxed{\theta\theta_r(\alpha)} \preceq \sqsubseteq^{\to} \theta[\beta] \to \theta[\gamma]$ (since $\theta\rho_g$ must be a ρ-type). □

Lemma A.4.9. *If* $\vdash C_1$, $\Delta_{C_1} \vdash \rho_1$, $C_1 \vdash \Gamma$, *and* $instMono(C_1, \Gamma, \rho_1) = C_2, \rho$ *then for every* $\theta \models C_2$, *it is the case that* $\theta \models C_1$ *and* $\theta[\rho_1] \preceq\sqsubseteq \theta\rho$

Proof. Similar to the proof of Lemma A.4.8 appealing to soundness for *mkMono*. □

Lemma A.4.10 (Soundness)**.** *If* $\vdash C_1$, $C_1 \vdash \Gamma$, *and* $infer(C_1, \Gamma, e) = C_2, \rho$ *then for all* $\theta \models C_2$, *it is the case that* $\theta \models C_1$ *and* $\theta\Gamma \vdash e : \theta[\rho]$.

Proof. We sketch the proof of soundness by induction on the number of steps of the algorithm. We omit showing the well-formedness conditions in the inductive cases. We consider the following cases.

- $infer(C_1, \Gamma, x) = C_2$ where $(x{:}\sigma) \in \Gamma$ and $inst(C_1, \sigma) = C_2$. In this case we may directly appeal to Lemma A.4.3.
- $infer(C_1, \Gamma, e_1\ e_2) = C_2, \rho_2$ where

$$(E_1, \rho_1) = infer(C_1, \Gamma, e_1) \tag{A.19}$$

$$(E_2, \sigma_1 \to \sigma_2) = instFun(E_1, \Gamma, \rho_1) \tag{A.20}$$

$$(E_3, \rho_3) = infer(E_2, \Gamma, e_2) \tag{A.21}$$

$$E_4, [E_g] \Rightarrow \rho_g = generalize(\Gamma, E_3, \rho_3) \tag{A.22}$$

$$E_5 = subsCheck(E_4, [E_g] \Rightarrow \rho_g, \sigma_1) \tag{A.23}$$

$$C_2, \rho_2 = inst(E_5, \sigma_2) \tag{A.24}$$

$$\tag{A.25}$$

  Assume $\theta \models C_2$, it follows that $\vdash^{\mathsf{inst}} \theta[\sigma_2] \leq \theta[\rho_2]$ by Lemma A.4.3. Moreover $\theta \models E_5$. By Lemma A.2.22 we get that $\theta\sigma_1 \in [\![\theta([E_g] \Rightarrow \rho_g)]\!]$ and $\theta \models E_4$. It follows that there exists a θ_g such that $\theta\theta_g \models E_4E_g$ and let $\theta\sigma_1 = \forall\overline{a}.\rho_1$. Assume without loss of generality that $\overline{a}\#ftv(\theta\Gamma)$. Moreover it must be that $\overline{a}\#ftv([\theta E_g] \Rightarrow \theta\rho_g)$ and $\rho_1 = \theta_g\theta\rho_g$ ($\overline{a}$ may appear in the range of θ_g). By Lemma A.4.7 it must then be that there exists a θ_r such that $\vdash^{\mathsf{F}} \theta\theta_g\theta_r(\rho_3) \leq \theta_g\theta\rho_g = \rho_1$. But $\overline{a}\#\theta\Gamma$ and hence it must be that $\overline{\theta\Gamma}(\theta\theta_g\theta_r(\rho_3) \leq \forall\overline{a}.\rho_1$. Moreover $\theta\theta_g\theta_r \models E_3$ and hence by induction hypothesis $\theta\Gamma \vdash e_2 : \theta\theta_g\theta_r[\rho_3]$, and $\theta\theta_g\theta_r \models E_2$.

Observe that the residual $\theta_g\theta_r$ was only referring to fresh variables, and hence at this point we know that $\theta \models E_2$. By Lemma A.4.8 we get that there exists a θ_r such that $\theta\theta_r \models E_1$ and $\theta\theta_r[\rho_1] \preceq\sqsubseteq^{\rightarrow} \theta[\sigma_1] \rightarrow \theta[\sigma_2]$. Moreover, by induction we get that $\theta\theta_r \models C_1$ and because θ_r is referring to fresh variables, $\theta \models C_1$. By induction we get $\theta\Gamma \vdash e_1 : \theta\theta_r[\rho_1]$. Applying rule SDAPP is enough to finish the case.

- $\mathit{infer}(C_1, \Gamma, (e :: \sigma)) = C_2, \rho_2$. This case involves the same reasoning about generalization as the application case and we omit it.
- $\mathit{infer}(C, \Gamma, \texttt{let } x = u \texttt{ in } e) = C_2, \rho$ where

$$E_1, \rho_1 = \mathit{infer}(C, \Gamma, u) \tag{A.26}$$

$$E_2, \rho_2 = \mathit{instMono}(E_1, \rho_1) \tag{A.27}$$

$$\rho_3 = \mathit{zonkType}(E_2, \rho_2) \tag{A.28}$$

$$\overline{\alpha} = \widehat{E_2}(\rho_3) - \widehat{E_2}(\Gamma) \quad \overline{\beta} = \overline{\alpha} \cap \mathit{fcv}(\rho_3) \quad \overline{a} \text{ fresh} \tag{A.29}$$

$$C_2, \rho = \mathit{infer}(E_2 - E_{2\overline{\alpha}}, \Gamma, (x{:}\forall \overline{a}\,.\,[\overline{\beta \mapsto a}]\rho_3), e) \tag{A.30}$$

Assume first of all that $\theta \models C_2$, and without loss of generality assume that $\overline{\alpha} \# \mathit{dom}(\theta)$ (otherwise we can work with a restriction of θ). Then, by induction hypothesis for (A.30) we get that $\theta \models E_2 - E_{2\overline{\alpha}}$. But consider the variable $\overline{\alpha}$. $E_{2\overline{\alpha}}$ contains only monomorphic equality bounds, and we can consider the most general unifier of $E_{2\overline{\alpha}}$, call it $E^{\dagger}_{2\overline{\alpha}}$. With a substitution of $\overline{\beta}$ to $\overline{a}$, we know that $[\overline{\beta \mapsto a}]E^{\dagger}_{2\overline{\alpha}}$, call it $\theta_{\overline{\alpha}}$, satisfies $E_{2\overline{\alpha}}$. That is, $\theta_{\overline{\alpha}} \models E_{2\overline{\alpha}}$. Then $\theta\theta_{\overline{\alpha}} \models E_2$ and by Lemma A.4.9 there exists a θ_r such that $\theta\theta_{\overline{\alpha}}\theta_r[\rho_1] \preceq\sqsubseteq \theta\theta_{\overline{\alpha}}\rho_2$. and $\theta\theta_{\overline{\alpha}}\theta_r \models E_1$. Hence, by induction $\theta\Gamma \vdash u : \theta\theta_{\overline{\alpha}}\theta_r[\rho_1]$ and $\theta\theta_{\overline{\alpha}}\theta_r \models C_1$. But observe that the $\overline{\alpha}$ and the domain of θ_r were freshly created and hence $\theta \models C_1$. Hence induction hypothesis will finish the case if we can show that $\overline{a} = \mathit{ftv}([\overline{\beta \mapsto a}]\theta\theta_{\overline{\alpha}}\rho_2) - \mathit{ftv}(\theta\Gamma)$:

  - $\overline{a} \subseteq \mathit{ftv}([\overline{\beta \mapsto a}]\theta\theta_{\overline{\alpha}}\rho_2) - \mathit{ftv}(\theta\Gamma)$. This direction is trivial, for fresh enough $\overline{a}$.
  - $\mathit{ftv}([\overline{\beta \mapsto a}]\theta\theta_{\overline{\alpha}}\rho_2) - \mathit{ftv}(\theta\Gamma) \subseteq \overline{a}$. Assume an $a \in \mathit{ftv}([\overline{\beta \mapsto a}]\theta\theta_{\overline{\alpha}}\rho_2) - \mathit{ftv}(\theta\Gamma)$. The hard case is when either $a \in \rho_2$ or $\exists \gamma \in \mathit{fcv}(\rho_2)$ such that $a \in \theta\theta_{\overline{\alpha}}\gamma$. We will show that this case cannot happen. In the first case it must be that a belongs in $\theta\Gamma$. In the second it must be the case that $\gamma \notin \overline{\beta}$, and hence $\gamma \notin \overline{\alpha}$ or $\gamma \notin \mathit{fcv}(\rho_2)$. If $\gamma \notin \overline{\alpha}$ it means that it

must be in $\widehat{E_2}(\Gamma)$ and hence $a \in \theta\Gamma$, a contradiction. The latter case, $\gamma \notin fcv(\rho_2)$ cannot happen as $\gamma \in fcv(\rho_2)$.

Putting the above together, induction hypothesis for (A.30) and application of SDLET finishes the case.

- $infer(C_1, \Gamma, \lambda x\,.\,e) = C_2, \rho$. The case is similar to the case for generalizing `let`-bindings.

□

A.5 Main algorithm completeness

Lemma A.5.1 (Scheme substitutivity)**.** *Assume that* $\Delta \vdash [D] \Rightarrow \rho$ *and* $\overline{a}\#ftv([D] \Rightarrow \rho)$ *and* Δ *contains only* $\star$*-flagged variables. If* $\theta \models D$ *then* $[\overline{a \mapsto \sigma}]\theta \models D$ *(where* $\overline{\sigma}$ *are constrained-variable-free).*

Proof. By induction on the metric of the constraint D. First of all, if $(\alpha_\star = \sigma_1) \in D$ we know that equality is preserved by substitutions, hence $\theta\alpha = \theta\sigma_1$ implies $[\overline{a \mapsto \sigma}]\theta\alpha = [\overline{a \mapsto \sigma}]\theta\sigma_1$, and moreover if $(\alpha_\star \geq \bot)$ then the result follows easily. $(\alpha_\star \geq [E] \Rightarrow \rho)$ we have that $\theta\alpha \in [\![[\theta E] \Rightarrow \theta\rho]\!]$ where without loss of generality we assume that $dom(E)\#dom(\theta)$. Assume that $\theta\alpha = \forall\overline{b}\,.\,\rho_a$. Then $\overline{b}\#ftv([\theta E] \Rightarrow \theta\rho)$, and without loss of generality assume also that $\overline{b}\#\overline{a}, ftv(\overline{\sigma})$. Then, there exists also a substitution θ_E such that $\theta_E \models \theta E$ and $\rho_a = \theta_E\theta\rho$. It follows that $\theta_E\theta \models E$ and consequently by induction $[\overline{a \mapsto \sigma}] \cdot (\theta_E\theta) \models E$ and $[\overline{a \mapsto \sigma}]\rho_a = [\overline{a \mapsto \sigma}](\theta_E\theta\rho)$. This implies that $([\overline{a \mapsto \sigma}] \cdot \theta_E)([\overline{a \mapsto \sigma}]\theta) \models E$, and moreover $([\overline{a \mapsto \sigma}] \cdot \theta_E)([\overline{a \mapsto \sigma}]\theta)(\rho) = [\overline{a \mapsto \sigma}]\rho_a$. Because $\overline{b}\#\overline{a}, ftv(\overline{\sigma})$ we have that $[\overline{a \mapsto \sigma}]\theta\alpha \in [\![[[\overline{a \mapsto \sigma}]\theta E] \Rightarrow [\overline{a \mapsto \sigma}]\theta\rho]\!]$, as required. □

Lemma A.5.2 (Scheme F-instance transitivity)**.** *If* $\vdash \varsigma$*,* $\sigma_1 \in [\![\varsigma]\!]$*, and* $\vdash^{\mathsf{F}} \sigma_1 \leq \sigma_2$ *then* $\sigma_2 \in [\![\varsigma]\!]$*.*

Proof. Let $\varsigma = [D] \Rightarrow \rho$. Since $\vdash \varsigma$ it must be that, if $\rho = \gamma$, then $\varsigma = [(\gamma_\star \bot)] \Rightarrow \gamma$. Additionally we know that for all $\gamma \in dom(D)$, $\Delta_D(\gamma) = \star$. Let $\sigma_1 = \forall\overline{a}\,.\,\rho_1$ and $\sigma_2 = \forall\overline{b}\,.\,[\overline{a \mapsto \sigma}]\rho_1$ such that $\overline{b}\#ftv(\forall\overline{a}\,.\,\rho_1)$. Then we know that there exists a θ_D such that $\theta_D \models D$ and $\rho_1 = \theta_D\rho$. Moreover $\overline{a}\#ftv([D] \Rightarrow \rho)$. We want to show that $\forall\overline{b}\,.\,[\overline{a \mapsto \sigma}]\rho_1 \in [\![[D] \Rightarrow \rho]\!]$. First of all, assume that ρ_1 is

not a single quantified variable a because in this case it must be that $\varsigma = [(\gamma_\star \perp)] \Rightarrow \gamma$, and then trivially also $\sigma_2 \in [\![\varsigma]\!]$. So, assume that all quantified variables of σ_2 are the $\overline{b}$. We have to show several things:

- $\overline{b} \# ftv([D] \Rightarrow \rho)$. Assume on the contrary that there exists a $b \in ftv([D] \Rightarrow \rho)$. It follows that $b \in \theta_D \rho = \rho_1$. But since $\overline{a} \# ftv([D] \Rightarrow \rho)$ it follows that $b \in ftv(\forall \overline{a} \,.\, \rho_1)$, a contradiction.
- We want to find a substitution θ°_D such that $[\overline{a \mapsto \sigma}]\rho_1 = \theta^\circ_D \rho$. We know that $\rho_1 = \theta_D \rho$, so consider as a candidate θ°_D the substitution $[\overline{a \mapsto \sigma}]\theta_D$ (where the notation means that we substitute the $\overline{a}$ for $\overline{\sigma}$ in the range of θ_D. We need to show that since $\theta_D \models D$ then $[\overline{a \mapsto \sigma}]\theta_D \models D$ as well. But this follows by Lemma A.5.1.

□

Lemma A.5.3. *If* $(\forall \overline{b} \,.\, \rho) \in [\![\varsigma]\!]$ *and* $\overline{a} \# ftv(\varsigma), \overline{b}$ *then* $\forall \overline{ab} \,.\, \rho) \in [\![\varsigma]\!]$.

Proof. Straightforward unfolding of the definitions. □

Lemma A.5.4 (Normalization completeness)**.** *Assume that* $C \vdash E$, $\Delta_C \Delta_E \vdash \rho$, $\theta \models C \bullet E$, *the domain of* E *is* $\star$*-bound variables only, constraint* E *and type* ρ *contain no rigid type variables, and* $normalize(E, \rho) = [E_g] \Rightarrow \rho_g$. *Then* $\theta\rho \in [\![[E_g] \Rightarrow \rho_g]\!]$.

Proof. By induction on the number of recursive calls to $normalize(E, \rho)$. The only interesting case is the case for N1:

- We have that $normalize(E, \alpha) = [E \bullet D - \alpha] \Rightarrow \rho$, when $(\alpha_\star \geq [D] \Rightarrow \rho) \in E$. since $\theta \models C \bullet E$ it is the case that $\theta\alpha \in [\![[\theta D] \Rightarrow \theta\rho]\!]$ (where we assume that the domain of D is disjoint from the domain of θ). We need to show that $\theta\alpha \in [\![\theta[E^\circ \bullet (D - \alpha)^\circ] \Rightarrow \theta\rho^\circ]\!]$ where $\circ$ is a renaming of the domain of E and D such that they are disjoint from the domain of θ. Let $\theta\alpha = \forall \overline{a} \,.\, \rho_a$ such that $\overline{a} \# ftv([\theta D] \Rightarrow \theta\rho)$. then there exists a θ_D such that $\theta_D \theta\rho = \rho_a$ and $\theta_D \models \theta D$. However, it must also be that $\theta^\circ_D \theta^\circ \models \theta[E^\circ \bullet (D - \alpha)^\circ]$ and then $\theta_D \theta^\circ \theta\rho^\circ = \theta_D \theta\rho = \rho_a$. To finish the case we need to show that $\overline{a} \# ftv(\theta[E^\circ \bullet (D - \alpha)^\circ] \Rightarrow \theta\rho^\circ)$. But assume on the contrary that there exists such an a. But since $\overline{a} \in ftv(\rho_a)$ and they cannot be in $\theta\rho$ it must be that

$\overline{a} \in range(\theta_D)$. If there exists such an a it must be that $a \in range(\theta_{ftv(\rho)})$ which is impossible because $\overline{a} \# ftv([\theta D] \Rightarrow \theta\rho)$.

□

Lemma A.5.5 (Generalization completeness)**.** *Assume that* $\vdash C_1$*,* $C_1 \vdash \Gamma, \rho$*, for all* $\theta \models C_1$ *and* $E, \varsigma = generalize(C_1, \Gamma, \rho)$*,* C_1 *and* ρ *do not contain any free rigid variables, and* $\overline{a} = ftv(\theta\rho) - ftv(\theta\Gamma)$*. It follows that* $(\forall \overline{a} \,.\, \theta\rho) \in [\![\theta\varsigma]\!]$ *and* $\theta \models E$*.*

Proof. We have that $\overline{\beta} = \widehat{C_1}(\rho) - \widehat{C_1}(\Gamma)$, $normalize(C_{\overline{\beta}}, \rho) = [E_g] \Rightarrow \rho_g$ and $E = C - C_{\overline{\beta}}$, $\varsigma = [E_g] \Rightarrow \rho_g$. We know that $\theta \models C_1$ and assume a renaming of the variables $\overline{\beta}$ to $\overline{\beta}^\circ$. Then we will first show that $\overline{a} \# ftv([\theta E_g^\circ] \Rightarrow \theta\rho_g^\circ)$. Assume, by contradiction, that there exists an $a \in \overline{a}$ such that $a \in ftv([\theta E_g^\circ] \Rightarrow \theta\rho_g^\circ)$. It follows that there exists a $\beta \in \widehat{C_1}(\Gamma)$ such that $\theta(\beta) = a$. But then it must be that $a \notin \overline{a}$. Or, it must be that $a \in ftv([E_g] \Rightarrow \rho_g)$. But this cannot happen as E_g and ρ_g can't contain any rigid variables. By Lemma A.5.3, it is hence enough to show that $\theta\rho \in [\![\theta\varsigma]\!]$. But this follows from Lemma A.5.4. □

Lemma A.5.6 (Inst-fun completeness)**.** *If* $\vdash C_1$ *and* $C_1 \vdash \Gamma$ *and* $\theta_1 \models C_1$*,* $dom(\theta_1) = dom(C_1)$*, and* $\theta_1[\rho] \preceq\sqsubseteq^{\rightarrow} \sigma_1' \rightarrow \sigma_2'$ *then* $instFun(C_1, \Gamma, \rho) = C_2, \sigma_1 \rightarrow \sigma_2$ *such that there exists a* θ_2 *with* $\theta_1\theta_2 \models C_2$ *and* $\sigma_1' = \theta_1\theta_2[\sigma_1]$*,* $\sigma_2' = \theta_1\theta_2[\sigma_2]$*.*

Proof. Unfolding definitions and appealing to completeness of *eqCheck* and $normalize(\cdot)$. □

We use below the fact that whenever we infer a type, that type contains no rigid type variable, but only unification ones. In essence, rigid type variables can only start appearing by open type annotations, (and the proofs need be slightly modified), but for simplicity we only treat closed type annotations.

Lemma A.5.7 (Main completeness)**.** *If* $\vdash C_1$ *and* $C_1 \vdash \Gamma$ *and* $\theta_1 \models C_1$ *and* $\theta_1\Gamma \vdash e : \rho'$ *then it is the case that* $infer(C_1, \Gamma, e) = C_2, \rho$*, and there exists a* θ_2 *such that* $\theta_1\theta_2 \models C_2$*, and* $\theta[\rho] \preceq\sqsubseteq \rho'$*.*

Proof. By induction on the size of term e, and considering cases on the derivation $\theta_1\Gamma \vdash e : \rho'$. The cases for `let` nodes rely on Lemma 5.3.5 and Corollary 6.4.3, and is similar to the ordinary case

for higher-rank types (since the types are box-free), and the substitutions monomorphic. The case for abstraction relies on Lemma 5.3.5 and is similar. The cases for annotations and applications are more interesting as they use the power of constrained types. We show below the case for applications, and the case for annotations is quite similar.

- Case SDAPP. In this case we have that $\vdash C_1$, $C_1 \vdash \Gamma$, $\theta \models C_1$ and $\theta_1\Gamma \vdash e_1\ e_2 : \rho_2'$ given that

$$\theta_1\Gamma \vdash^{\mathsf{sd}} e_1 : \rho' \tag{A.31}$$

$$\rho'(\preceq_{\sqsubseteq}^{\rightarrow})\sigma_1' \rightarrow \sigma_2' \tag{A.32}$$

$$\Gamma \vdash^{\mathsf{sd}} e_2 : \rho_3' \quad \overline{a} = \mathit{ftv}(\rho_3') - \mathit{ftv}(\Gamma) \tag{A.33}$$

$$\vdash^{\mathsf{F}} \lfloor \forall \overline{a} \,.\, \rho_3' \rfloor \leq \lfloor \sigma_1' \rfloor \quad \vdash^{\mathsf{inst}} \sigma_2' \leq \rho_2' \tag{A.34}$$

  By induction hypothesis for (A.31) we get that $\mathit{infer}(C_1, \Gamma, e_1) = E_1, \rho_1$ such that there exists a θ_2 with $\theta_1\theta_2 \models C_2$ and $\theta_1\theta_2[\rho_1] \preceq_{\sqsubseteq} \rho'$. By Lemma A.5.6 and transitivity of $\preceq_{\sqsubseteq}$ we get that $\mathit{instFun}(E_1, \rho_1) = E_2, \sigma_1 \rightarrow \sigma_2$ such that there exists a θ_3 with $\theta_1\theta_2\theta_3 \models E_2$ and $\theta_1\theta_2\theta_3[\sigma_1] = \sigma_1'$ and $\theta_1\theta_2\theta_3[\sigma_2] = \sigma_2'$. Moreover, we know at this point that $\theta_1\theta_2\theta_3\Gamma \vdash e_2 : \rho_3'$ (since all the variables of Γ are already in the domain of θ_1). It follows by induction hypothesis that $\mathit{infer}(E_2, \Gamma, e_2) = E_3, \rho_3$ such that there exists a θ_4 with $\theta_1\theta_2\theta_3\theta_4 \models E_3$ and $\theta_1\theta_2\theta_3\theta_4[\rho_3] \preceq_{\sqsubseteq} \rho_3'$. Next, we have that if $\overline{a} = \mathit{ftv}(\theta_1\theta_2\theta_3\theta_4\rho_3) - \mathit{ftv}(\theta_1\Gamma)$ it is the case that $\vdash^{\mathsf{F}} \forall \overline{a} \,.\, \theta_1\theta_2\theta_3\theta_4\rho_3 \leq \theta_1\theta_2\theta_3\sigma_1 = \theta_1\theta_2\theta_3\theta_4\sigma_1$. By Lemma A.5.5 we may call $\mathit{generalize}(\Gamma, E_3, \rho_3)$ to get back E_4, ς_3 (*generalize* never fails). By Lemma A.5.5 we get that $\forall \overline{a} \,.\, \theta_1\theta_2\theta_3\theta_4\rho_3 \in [\![\theta_1\theta_2\theta_3\theta_4\varsigma_3]\!]$, and $\theta_1\theta_2\theta_3\theta_4 \models E_4$, and by Lemma A.5.2 it must be that $\theta_1\theta_2\theta_3\theta_4\sigma_1 \in [\![\theta_1\theta_2\theta_3\theta_4\varsigma_3]\!]$. By Lemma A.3.2 it must be the case that $E_5 = \mathit{subsCheck}(E_4, \varsigma_3, \sigma_1)$ such that there exists a θ_5 with $\theta_1\theta_2\theta_3\theta_4\theta_5 \models E_5$. Finally, the instantiation step concludes the case by applying the algorithm clause IAPP.

□

Bibliography

[1] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*, pages 379–405, 1991.

[2] Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallin, Estonia, 2005.

[3] Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM Symposium on Principles of Programming Languages (POPL'05)*. ACM Press, 2005.

[4] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–12, New York, 1982. ACM Press.

[6] Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Journal of Information and Computation*, 155:134–169, 1999.

[7] Jean-Yves Girard. *Interprétation foctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII, 1972.

[8] Jean-Yves Girard. The system F of variable types: fifteen years later. In Gérard Huet, editor, *Logical Foundations of Functional Programming*. Addison-Wesley, 1990.

[9] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[10] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In Don Sannella, editor, *European Symposium on Programming (ESOP'94)*, pages 241–256. Springer Verlag LNCS 788, April 1994.

[11] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, Pittsburgh, PA, June 1997. (Also published as Fox Memorandum CMU–CS–FOX–97–01.).

[12] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. (146):29–60, 1969.

[13] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[14] Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1:22–58, 1975.

[15] Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, November 1994.

[16] Mark P. Jones. First-class polymorphism with type inference. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 1997.

[17] Mark P. Jones. Type classes with functional dependencies. In *European Symposium on Programming (ESOP'00)*, number 1782 in Lecture Notes in Computer Science, Berlin, Germany, March 2000. Springer Verlag.

[18] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[19] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[20] Simon Peyton Jones and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[21] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476, New York, NY, USA, 1990. ACM.

[22] Assaf J. Kfoury and Joe B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda calculus. In *ACM Symposium on Lisp and Functional Programming*, pages 196–207. ACM, Orlando, Florida, June 1994.

[23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, January 2003. ACM Press.

[24] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *ACM Conference on Programming Languages Design and Implementation, Orlando (PLDI'94)*, pages 24–35. ACM Press, June 1994.

[25] Didier Le Botlan. *MLF : Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, École Polytechnique, May 2004.

[26] Didier Le Botlan and Didier Rémy. MLF: raising ML to the power of System F. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 27–38, Uppsala, Sweden, September 2003. ACM.

[27] Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, June 2007.

[28] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[29] Daan Leijen. A type directed translation of MLF to System-F. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, Freiburg, Germany, 2007. ACM.

[30] Daan Leijen. Flexible types: robust type inference for first-class polymorphism. Technical Report MSR-TR-2008-55, Microsoft Research, March 2008.

[31] Daan Leijen. HMF: simple type inference for first-class polymorphism. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*. ACM, 2008.

[32] Daan Leijen and Andres Löh. Qualified types for MLF. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 144–155. ACM Press, 2005.

[33] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml system, documentation and user's guide*. INRIA, 1998. Available at `http://pauillac.inria.fr/ocaml/htmlman/`.

[34] Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. A logic of subtyping (extended abstract). In *LICS95*, pages 292–299, 1995.

[35] Robin Milner. A theory of type polymorphism in programming. *JCSS*, 13(3), December 1978.

[36] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[37] John C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, 1988.

[38] Greg Morrisett. *Compiling with types*. Ph.D. thesis, Carnegie Mellon University, December 1995.

[39] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 54–67. ACM, St Petersburg Beach, Florida, January 1996.

[40] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, London, January 2001. ACM.

[41] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.

[42] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, Portland, Oregon, 2006. ACM Press.

[43] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM.

[44] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[45] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 252–265, San Diego, January 1998. ACM.

[46] Didier Rémy. Simple, partial type inference for System F, based on type containment. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 130–143, Tallinn, Estonia, September 2005. ACM.

[47] Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In *TLDI'07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 27–38, Nice, France, January 2007. ACM Press.

[48] Didier Rémy and Boris Yakobowski. A Church-Style Intermediate Language for MLF. July 2008.

[49] Didier Rémy and Boris Yakobowski. Graphic type constraints and efficient type inference: from ML to MLF. Submitted, April 2008.

[50] J. Alan Robinson. Computational logic: The unification computation. In *Machine Intelligence 6*, pages 63–72. Edinburgh University Press, 1971.

[51] Ken Shan. Sexy types in action. *SIGPLAN Notices*, 39(5):15–22, May 2004.

[52] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[53] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*, pages 181–192. ACM, Philadelphia, May 1996.

[54] Jerzy Tiuryn. A sequent calculus for subtyping polymorphic types. *Inf. Comput.*, 164(2):345–369, 2001.

[55] Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second order types is undecidable. In *Proc IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 74–85, 1996.

[56] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Practical type inference for arbitrary-rank types, Technical Appendix. Technical Report MS-CIS-05-14, University of Pennsylvania, July 2005.

[57] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity, Technical Appendix. Technical Report MS-CIS-05-23, University of Pennsylvania, April 2006.

[58] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, Portland, Oregon, 2006. ACM Press.

[59] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH:first-class polymorphism for haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*. ACM, 2008.

[60] Joe B. Wells. Typability is undecidable for F+eta. Technical Report 96-022, Computer Science Department, Boston University, March 1996.

[61] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98:111–156, 1999.

[62] Hermann Weyl. *Das Kontinuum, trans. 1987 The Continuum : A Critical Examination of the Foundation of Analysis*. 1918. ISBN: 0-486-67982-9.

[63] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235. ACM Press, 2003.