Syntactic Implicit Parameters with Static Overloading

Microsoft Research Technical Report, MSR-TR-2025-56, 2026-01-05 (v3)

DAAN LEIJEN, Microsoft Research, USA
TIM WHITING, Brigham Young University, USA

Implicits provide a powerful mechanism for term-based inference, where "obvious" arguments can be omitted and inferred by the type checker. This can greatly reduce the programmer's burden and improve the clarity of expression. As such, many languages support a form of implicits in practice, such as type classes in Haskell or Lean, or implicits in Scala. Unfortunately, many of these systems have become increasingly complex and often require significant implementation effort.

In this paper we take a fresh look at the design space with an arguably simpler approach based on two orthogonal features: *syntactic implicit parameters* and *static overloading*. Each of these features is limited in scope and has a straightforward implementation. Taken together though, they are surprisingly expressive and we believe they can cover many of the common usage scenarios of implicits in practice.

We formalize our system and provide various examples, and prove our elaboration is coherent. We also give an inference algorithm and show it is sound and complete. Our system is fully implemented in the Koka language, and we describe our experience with these features at scale, and discuss further extensions.

1 INTRODUCTION

Implicits provide a powerful mechanism for term-based inference, where "obvious" terms can be omitted and inferred by the type checker. This can greatly reduce the programmers burden and improve the clarity of expression. For example, without implicits, a programmer may have to write `show_list show_int [1]` to show a list of integers, while a Haskell programmer instead can simply write `show [1]`, where the correct type class dictionary for showing lists of integers is provided implicitly.

As such, many languages support a form of implicits in practice, such as type classes in Haskell or Lean, implicits in Scala, or in systems like Rocq and Agda where the language can provide obvious proof terms [Devriese and Piessens 2011; Odersky et al. 2017; Selsam et al. 2020; Sozeau and Oury 2008; P. Wadler and Blott 1989]. However, the design of many such systems has become increasingly complex and difficult to implement, and it is not always clear how the different approaches in all these systems relate to each other.

In this paper we take a fresh look at the design space with an arguably simpler approach based on two orthogonal features: *syntactic implicit parameters* and *static overloading*. Each of these features is limited in scope and has a straightforward implementation. Taken together though, they are surprisingly expressive and we believe they can cover many of the common usage scenarios of implicits in practice. The first idea is to treat implicit parameters syntactically as a form of dynamic binding:

Syntactic implicit parameters are parameter names that are supplied literally as arguments at the call site.

For example, we can define a function with an implicit parameter `base` as:

```
fun show-int( x : int, ?base : int ) : string
  if x < base
    then show-digit(x)
    else show-int(x / base) ++ show-digit(x % base)
```

This function takes an implicit parameter `base`, denoted by the question mark, which can be used inside the function as any regular parameter (like `x < base`). When calling `show-int` though, we do

not have to provide the `?base` parameter explicitly and can leave it out: at each call site the name `base` is syntactically used as the argument, and resolved *in the scope of the call site*. For example,

```
val base = 10 in show-int(42)
```

elaborates to

```
val base = 10 in show-int(42,base)
```

which evaluates to `"42"`. We see this elaboration as well in the `show-int` function itself where the expression `show-int(x % base)` elaborates to `show-int(x%base,base)` where it passes the implicit parameter `base` again to the recursive call.

This view of implicit parameters as a form of dynamic binding (but with a static declaration!) is quite straightforward and seems almost too simple to be of much use. However, it turns out to be quite expressive in combination with *static overloading*. This brings us to the second idea, where we define static overloading purely as a form of automatic qualification:

Static overloading elaborates plain names to fully qualified names based on the local type context.

The essence of this idea was first described by Leijen and Ye [2025] as an application of type inference under a prefix. First, we allow functions to be declared with qualified names, for example:

```
fun int/show( x : int ) : string
  show-int(x,10)

fun float/show( f : float64 ) : string
  show-float(f)
```

(where `int/` and `float/` can be arbitrary "module" names). Of course, such qualified names already occur naturally as well in most languages when different modules are imported that export the same (unqualified) name. We now allow the programmer to write an unqualified `show` and have it be resolved to either definition based on the local type context. For example `show(1)` is elaborated to the fully qualified `int/show(1)` based on the (static) type of the argument. This is already quite convenient in practice, and is again a simple mechanism that is straightforward to implement – for example the C language implements this form of static overloading for many common math operations. However, static overloading by itself is quite limited as it does not allow for *abstraction*. For example, consider a `show` function for lists:

```
fun list/show( xs : list<a> ) : string
  match xs
    Cons(x,xx) -> show(x) ++ "::" ++ list/show(xx)  // rejected
    Nil        -> "[]"
```

This is rejected since we cannot at this point statically resolve which `show` function is required for the `show(x)` expression (as the type of the list elements is polymorphic). Here is where we can now use our new syntactic implicit parameters to delay resolving which particular `show` to use:

```
fun list/show( xs : list<a>, ?show : a -> string ) : string
  match xs
    Cons(x,xx) -> show(x) ++ "::" ++ list/show(xx)
    Nil        -> "[]"
```

Just like a Haskell programmer, we can now write `show([1])` to show a list of integers.

- Static overloading first elaborates the plain `show` to `list/show([1])`, based on the type `list<int>` of the argument.
- Subsequently, the `?show` implicit parameter is now supplied as `list/show([1],show)`.
- Static overloading kicks in again and further disambiguates this (implicit) `show` to `int/show` (again based on the local type context), which results in the final elaboration as `list/show([1],int/show)`.

As an aside, in the recursive call in the `list/show` function, the implicit parameter is also supplied as `list/show(xx,show)`; this does not need further disambiguation though as it is already bound locally.

Note also that an expression like `show([])` is still rejected as the element type cannot be statically determined (similar to unresolved overloading in Haskell). Here we either need to give an explicit type signature, or provide the implicit parameter ourselves (as `show([],int/show)` for example).

As we could see, the resolving of syntactic overloads and syntactic implicit parameters is applied recursively. For example, we can show lists of lists of integers as `show([[1],[2]])`, which elaborates to `list/show([[1],[2]], fn(xs) list/show(xs,int/show))` (where the implicit `list/show` argument is automatically eta-expanded in order to supply its own implicit argument).

Even though syntactic implicits and static overloading by themselves are straightforward features, this recursive interaction between them makes them surprisingly expressive. We believe that they can handle many situations that are usually addressed using more elaborate language extensions. Also, our new features both are source-to-source elaborations and a programmer can always use unambiguous fully qualified names, or provide implicit parameters explicitly. As a pure elaboration, there are no new semantic features (like an "instance"), or special "implicit scopes" etc. – it is all just names, functions, and parameters. We make the following contributions:

- We formalize syntactic implicit parameters and static overloading precisely (Section 2). The type rules are specified using inference under prefix [Leijen and Ye 2025] to give unambiguous type contexts which are necessary for static overloading. Our rules go beyond the original example of static overloading by Leijen and Ye [2025] in that we study in particular its interaction with syntactic implicit parameters and how we maintain coherence and stability (Section 2.10).
- We show that even though our two ideas look innocent enough, together they make general type checking undecidable where we can encode a Turing machine on the type level which is executed by the type checker (Section 3). We give an improved set of "finite" rules that recover decidablity at the price of giving up completeness. We show though that our new finite rules are still sound with respect to our original system.
- We give an inference algorithm for the finite rules (Section 3.4) and show it is sound and complete. By specifying the algorithm under an effect handler we can modularly optimize the algorithm to not explore unnecessary branches in the search space.
- Our system is fully implemented in the Koka language [Leijen 2019 2021] and we discuss various implementation aspects of using these features at scale (Section 4).
- We also discuss the extension to *phantom implicits* where the compiler can resolve special implicit parameters in other ways than by name. For example `?kk-line:int` may provide the line number at the call site. Taking this further, we show how divergence constraints in the Koka language [Leijen 2014] can be handled by phantom implicits instead (Section 4.3).

Proofs and appendices can be found in the supplementary material.

2 FORMALIZATION

We start with formalizing both syntactic implicits and static overloading within a single calculus.

2.1 Syntax

Figure 1 gives the syntax of our core calculus that has qualified names. A plain name is written as x while a fully qualified name is written as z (or m for modules when needed). We use the *hat* operator to unqualify a name. For example, if a qualified name z has the form $x_1/\ldots/x_n/x$, then $\hat{z} = x$. All bindings can be specified with a qualified name z, and we can write $\lambda foo/x.\ foo/x\ +\ 1$ for example.

For simplicity, our calculus does not make implicit parameters first-class and we can only bind them at let bindings. A let binding binds a *term* t which can start with a sequence of implicit parameter bindings, followed by a regular expression e. For example,

let $mod/plus\ =\ \lambda?mod.\ \lambda x.\ \lambda y.\ (x\ +\ y)\ \%\ mod$
in (let $mod = 8$ in $mod/plus\ 5\ 6$)

z, m	::=	m/x	(qualified name)	τ	::=	α	(type variable)
	|	x	(plain name)		|	$\tau \to \tau$	(function arrow)
					|	$int \mid bool \mid \ldots$	(type constants)
e	::=	z	(variable)				
	|	$e\ e$	(application)	ρ	::=	$?x{:}\tau \to \rho$	(implicit arrow)
	|	$\lambda z.\ e$	(function)		|	τ	(mono type)
	|	let $z = t$ in e	(let binding)	σ	::=	$\forall\alpha.\sigma \mid \rho$	(type scheme))
t	::=	$\lambda?z.\ t$	(implicit param)	Γ	::=	$z_1 : \sigma_1, \ldots, z_n : \sigma_n$	(type env.)
	|	e	(expression)	Q	::=	$\{\alpha_1{=}\tau_1, \ldots, \alpha_n{=}\tau_n\}$	(prefix)
$\hat{m/x}$	=	x	(unqualify)				
$\hat{x}$	=	x					

Fig. 1. Syntax of types and terms.

binds an implicit parameter *mod* (and evaluates to 3).

Expressions have a monomorphic type τ which are either type variables α, or function types $\tau_1 \to \tau_2$. We also use types like *int* or *bool* in examples. Let-bound values are assigned a polymorphic *type scheme* σ. Since let-bound values can have implicit parameters, we extend the standard type schemes with implicit parameter types ρ. An implicit parameter type $?x{:}\tau \to \rho$ denotes a function type that takes an implicit parameter x of type τ. The type of an implicit parameter binds stronger than the function arrow, so $?x{:}\tau \to \tau$ should be read as $(?x{:}\tau) \to \tau$. A type scheme is generally of the form $\forall\overline{\alpha}.\ ?x_1{:}\tau_1 \to \ldots\ ?x_n{:}\tau_n \to \tau$ and at every variable occurrence we fully instantiate the type variables $\overline{\alpha}$ and supply all implicit parameters x_1 to x_n.

2.2 Type Inference under a Prefix

For the purpose of static overloading, we formulate our rules as type rules under a prefix Q [Leijen and Ye 2025]. This allows us to specify deterministic type rules with only principal derivations. For example, consider $\lambda x.\ show\ x$ – we should reject this as it is ambiguous which *show* function to elaborate to. However, when using standard Hindley-Milner rules one is allowed to use more specific types in derivations, and for example assume type *int* or *bool* for the parameter which makes it well-typed. With type inference under a prefix we avoid this issue. Moreover, as shown by Leijen and Ye [2025], specifying the type rules under a prefix also allows us to "read off" the inference algorithm from the rules directly, while still being close to the clarity of standard Hindley-Milner style type rules. We only briefly describe the essential details of prefixes and refer the interested reader to the original work.

A prefix Q is essentially just a set of type variable bounds $\alpha{=}\tau$. In general a prefix is a collection of such bounds and can be inconsistent or have duplicate bindings, like $\{\alpha{=}\beta{\to}int, \alpha{=}int{\to}\gamma\}$ or $\{\alpha{=}bool, \alpha{=}int\}$. We write $\theta \vDash Q$ if a substitution θ is a solution to Q that satisfies all constraints (with $\forall(\alpha{=}\tau) \in Q.\ \theta\alpha = \theta\tau$). If there exists any solution, we call Q consistent and write just $\vDash Q$.

It turns out for any consistent Q there is also a least (or best) solution which we call the prefix solution written as $\langle Q\rangle$. We often write $Q[\tau]$ as a shorthand for applying the prefix solution as $\langle Q\rangle(\tau)$. Finally, two prefixes are equivalent whenever their solution substitutions are equivalent where $Q_1 \equiv Q_2 \iff \langle Q_1\rangle \equiv \langle Q_2\rangle$. For example, we have we have $\{\alpha{=}\beta{\to}int, \alpha{=}\gamma{\to}\gamma\} \equiv \{\gamma{=}int, \beta{=}\gamma, \alpha{=}\gamma{\to}\gamma\} \equiv \{\beta{=}int, \gamma{=}int, \alpha{=}int{\to}int\}$. Similar to α-renaming, we can always substitute equivalent prefixes in type derivations.

$$\boxed{\begin{array}{c} Q \vdash \tau \approx \tau \\ \downarrow \quad \uparrow \quad \uparrow \\ \text{out} \quad \text{in} \quad \text{in} \end{array}}$$

$$\frac{\alpha \notin \text{ftv}(\tau)}{\{\alpha{=}\tau\} \vdash \alpha \approx \tau}\text{EQ-VAR} \qquad \frac{}{\varnothing \vdash \tau \approx \tau}\text{EQ-ID}$$

$$\frac{Q_1 \vdash \tau_1 \approx \tau_3 \quad Q_2 \vdash \tau_2 \approx \tau_4}{Q_1, Q_2 \vdash \tau_1 \to \tau_2 \approx \tau_3 \to \tau_4}\text{EQ-FUN} \qquad \frac{Q \vdash \tau_2 \approx \tau_1}{Q \vdash \tau_1 \approx \tau_2}\text{EQ-REFL}$$

Fig. 2. Type equivalence.

2.3 Type Equivalence under a Prefix

A *consistent union* is written as Q_1, Q_2 and denotes the union $Q_1 \cup Q_2$ where $Q_1 \cup Q_2$ is solvable. We use this in the conclusion of derivation rules to concisely denote that we can only derive consistent prefixes. This helps us write nice declarative type rules while avoiding having to thread a substitution linearly through each sub-derivation. We see this when defining type equivalence as shown in Figure 2.

A rule $Q \vdash \tau_1 \approx \tau_2$ states that a type τ_1 is equal to a type τ_2 under a (result) prefix Q. In particular, in the rule [EQ-FUN] we can easily compose the prefixes Q_1 and Q_2 from each sub derivation. These definitions are sound and complete [Leijen and Ye 2024]:

Theorem 2.1. (*Type equivalence under a prefix is sound*)
If $Q \vdash \tau_1 \approx \tau_2$ then $Q[\tau_1] = Q[\tau_2]$.

Theorem 2.2. (*Type equivalence under prefix is complete*)
If $\theta\tau_1 = \theta\tau_2$, then there exists a Q such that $Q \vdash \tau_1 \approx \tau_2$ and $Q \sqsubseteq \theta$.

Soundness states that if we can derive that τ_1 and τ_2 are equivalent under a prefix Q, then the types are syntactically equal under the prefix solution: $Q[\tau_1] = Q[\tau_2]$. Completeness shows that if there exists any substitution θ that makes two types equal, then we can also derive that these types are equivalent under a prefix Q, and that this prefix is also the "best" (most-general) solution: $\langle Q \rangle \sqsubseteq \theta$.

The prefix Q gives us an elegant way to declaratively specify type rules where we are assured that we can still implement this efficiently in a compiler using a linear substitution. In the case of the type equivalence rules, this corresponds directly to the usual unification algorithm [Pierce [2002],§22.4.5].

2.4 Instantiation

For most of the type checking rules, we are using essentially the bidirectional inference rules under a prefix as given by Leijen and Ye [2025]. For now, we look in particular at the checking rules for resolving variables:

$$\frac{z : \sigma \in \Gamma \qquad Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \;\rightsquigarrow \overline{e}}{Q \mid \Gamma \Vdash z \overleftarrow{:} \tau \;\rightsquigarrow z\,\overline{e}}\text{VAR-DIRECT}$$

Here the typing judgement $Q \mid \Gamma \Vdash z \overleftarrow{:} \tau \;\rightsquigarrow z\,\overline{e}$ states that a variable z can be *checked* ($\overleftarrow{:}$) to have type τ in a type environment Γ under the (result) prefix Q, and returning an elaborated expression $z\,\overline{e}$ (where $\overline{e}$ is a list of potentially instantiated implicit parameters). In particular, we must have $z : \sigma \in \Gamma$, and we must be able to *instantiate* the type scheme σ to τ, written as $Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \;\rightsquigarrow \overline{e}$.

Figure 3 shows the rules for type scheme instantiation. The rule [INST-MONO] says that two mono types need to be equivalent $Q \vdash \tau_1 \approx \tau_2$, i.e. they need to be unifiable under Q. The rule [INST-QUANTIFY] is also standard and instantiates a quantifier using a fresh type variable α.

$$\underset{\substack{\downarrow\\ \text{out}}}{Q} \mid \Gamma \vdash \underset{\substack{\uparrow\\ \text{in}}}{\sigma} \sqsubseteq \underset{\substack{\uparrow\\ \text{in}}}{\tau} \rightsquigarrow \underset{\substack{\downarrow\\ \text{out}}}{\overline{e}}$$

$$\frac{Q_1 \mid \Gamma \vdash \rho \sqsubseteq \tau_2 \rightsquigarrow \overline{e} \quad Q_2 \mid \Gamma \Vdash x \mathbin{\overset{\leftarrow}{:}} Q_1[\tau_1] \rightsquigarrow e}{Q_1, Q_2 \mid \Gamma \vdash ?x{:}\tau_1 \to \rho \sqsubseteq \tau_2 \rightsquigarrow e\ \overline{e}}\ \textsc{inst-implicit}$$

$$\frac{Q \vdash \tau_1 \approx \tau_2}{Q \mid \Gamma \vdash \tau_1 \sqsubseteq \tau_2 \rightsquigarrow \cdot}\ \textsc{inst-mono} \qquad \frac{Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e} \quad \text{fresh } \alpha}{Q \mid \Gamma \vdash \forall \alpha.\sigma \sqsubseteq \tau \rightsquigarrow \overline{e}}\ \textsc{inst-quantify}$$

Fig. 3. Type scheme instantiation.

$$\underset{\substack{\downarrow\\ \text{out}}}{Q} \mid \Gamma \Vdash \underset{\substack{\uparrow\\ \text{in}}}{z} \mathbin{\overset{\leftarrow}{:}} \underset{\substack{\uparrow\\ \text{in}}}{\tau} \rightsquigarrow \underset{\substack{\downarrow\\ \text{out}}}{e}$$

$$\frac{z{:}\sigma \in \Gamma \quad Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}}{Q \mid \Gamma \Vdash z \mathbin{\overset{\leftarrow}{:}} \tau \rightsquigarrow z\ \overline{e}}\ \textsc{var-direct}$$

$$\frac{z \notin \mathrm{dom}(\Gamma) \quad m/z{:}\sigma \in \Gamma \quad Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}}{Q \mid \Gamma \Vdash z \mathbin{\overset{\leftarrow}{:}} \tau \rightsquigarrow m/z\ \overline{e}}\ \textsc{var-qualify}$$

Fig. 4. Variable rules.

2.5 Syntactic Implicit Parameters

For standard Hindley-Milner type schemes the previous rules suffice, but in our case we added one more rule: the [INST-IMPLICIT] rule instantiates an implicit parameter type $?x{:}\tau_1 \to \rho$. In this case we can use the type rule for variables (recursively) to resolve the literal name x as:

$$Q_2 \mid \Gamma \Vdash x \mathbin{\overset{\leftarrow}{:}} Q_1[\tau_1] \rightsquigarrow e$$

This rule concisely captures the essence of syntactic implicit parameters: we resolve the name of an implicit parameter $?x$ (recursively) in the local scope Γ.

Note that we first instantiate $Q_1 \mid \Gamma \vdash \rho \sqsubseteq \tau_2$ which returns the prefix Q_1, and then propagate this information to check x under $Q_1[\tau_1]$. This is important as the regular parameters often give rise to further type constraints that can help to resolve the implicit parameters unambiguously. We will show some examples after discussing static overloading.

Furthermore, each resolved implicit parameter is elaborated as e and added to the list of implicit parameters that need to be applied (as $e\ \overline{e}$). If we look at the example at the start of this section, we can derive for the *mod/plus* expression:

$$\frac{\dfrac{\dfrac{}{\varnothing \vdash \tau \approx \tau}}{\varnothing \mid \Gamma \vdash \tau \sqsubseteq \tau \rightsquigarrow \cdot} \qquad \dfrac{mod : int \in \Gamma \qquad \dfrac{\dfrac{}{\varnothing \vdash int \approx int}}{\varnothing \mid \Gamma \vdash int \sqsubseteq int \rightsquigarrow \cdot}}{\varnothing \mid \Gamma \Vdash mod \mathbin{\overset{\leftarrow}{:}} \varnothing[int] \rightsquigarrow mod}}{mod/plus : ?mod{:}int \to \tau \in \Gamma \qquad \varnothing \mid \Gamma \vdash ?mod{:}int \to \tau \sqsubseteq \tau \rightsquigarrow mod}}{\varnothing \mid \Gamma \Vdash mod/plus \mathbin{\overset{\leftarrow}{:}} \tau \rightsquigarrow mod/plus\ mod}$$

where we use $\tau = int \to int \to int$ and $\Gamma = \{mod/plus : ?mod{:}int \to \tau,\ mod : int\}$. As we can see, the variable *mod/plus* is now elaborated to *mod/plus mod* where the implicit parameter is passed to the *mod/plus* function.

2.6 Static Overloading as Qualification

For our notion of static overloading we require another variable rule. In particular, when we have a variable occurrence z that is not occurring exactly in the type environment Γ, we can attempt to find a qualified name m/z that matches the required type. Figure 4 shows the new variable rule

[VAR-QUALIFY] (together with the earlier [VAR-DIRECT] rule):

$$\frac{z \notin \mathrm{dom}(\Gamma) \quad m/z : \sigma \in \Gamma \quad Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}}{Q \mid \Gamma \Vdash z \overset{\leftarrow}{:} \tau \rightsquigarrow m/z\ \overline{e}}\ \text{VAR-QUALIFY}$$

The first pre-condition states that z cannot occur directly in Γ, such that applying [VAR-DIRECT] or [VAR-QUALIFY] is deterministic. We can satisfy the other conditions if there exists an m with $m/z : \sigma$ in Γ such that we can instantiate σ to the required type τ, giving an elaborated expression $m/z\ \overline{e}$. Note that we keep the [INST-IMPLICIT] rule (in Figure 3) unchanged such that any implicit parameters can now also be resolved to a qualified name with the [VAR-QUALIFY] rule.

However, the rule as stated introduces *incoherence*: there might be multiple m with m/z in Γ that can be instantiated to the required type, each elaborating to a different expression. Which one to choose? In our case we eliminate incoherence by construction. The main type inference rule for variables requires a unique elaboration:

$$\frac{\exists!e.\quad Q \mid \Gamma \Vdash z \overset{\leftarrow}{:} \tau \rightsquigarrow e}{Q \mid \Gamma \vdash z \overset{\leftarrow}{:} \tau}\ \text{VAR}$$

where we use the notation $\exists!e$ for "there exists a unique e".

These rules concisely capture the essence of static overloading: we can elaborate a name z to a fully qualified name m/z as long as the solution is unique for a given type context τ.

Although each of these rules is quite simple, the recursive nature of the rules [VAR-QUALIFY] and [INST-IMPLICIT] make it quite expressive. Note in particular that the elaboration from z to m/z may include further implicit parameters $\overline{e}$ that were resolved recursively. Consider for example the following type environment Γ:

$$\mathit{int/show} : \mathit{int} \to \mathit{string},\ \mathit{list/show} : \forall\alpha.\ ?\mathit{show}{:}(\alpha \to \mathit{string}) \to \mathit{list}\ \alpha \to \mathit{string}$$

When we type check the expression $\mathit{show}\ [1,2]$ we can proceed as:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\alpha{=}\mathit{int}\} \vdash \mathit{list}\ \alpha{\to}\mathit{string} \approx \mathit{list}\ \mathit{int}{\to}\mathit{string} \qquad \dfrac{\mathit{int/show} \in \Gamma \quad \varnothing \mid \Gamma \vdash \mathit{int}{\to}\mathit{string} \sqsubseteq \mathit{int}{\to}\mathit{string} \rightsquigarrow \cdot}{\varnothing \mid \Gamma \Vdash \mathit{show} \overset{\leftarrow}{:} \{\alpha{=}\mathit{int}\}[\alpha{\to}\mathit{string}] \rightsquigarrow \mathit{int/show}}}{\{\alpha{=}\mathit{int}\} \mid \Gamma \vdash \mathit{list}\ \alpha{\to}\mathit{string} \sqsubseteq \mathit{list}\ \mathit{int}{\to}\mathit{string} \rightsquigarrow \cdot}}{\{\alpha{=}\mathit{int}\} \mid \Gamma \vdash ?\mathit{show}{:}(\alpha{\to}\mathit{string}) \to \mathit{list}\ \alpha \to \mathit{string} \sqsubseteq \mathit{list}\ \mathit{int} \to \mathit{string} \rightsquigarrow \mathit{int/show}}}{\begin{array}{l}\mathit{list/show} : ?\mathit{show}{:}(\alpha{\to}\mathit{string}) \to \mathit{list}\ \alpha \to \mathit{string} \in \Gamma \\ \{\alpha{=}\mathit{int}\} \mid \Gamma \vdash \forall\alpha.\ ?\mathit{show}{:}(\alpha{\to}\mathit{string}) \to \mathit{list}\ \alpha \to \mathit{string} \sqsubseteq \mathit{list}\ \mathit{int} \to \mathit{string} \rightsquigarrow \mathit{int/show}\end{array}}}{\exists!\ \{\alpha{=}\mathit{int}\} \mid \Gamma \Vdash \mathit{show} \overset{\leftarrow}{:} \mathit{list}\ \mathit{int} \to \mathit{string} \rightsquigarrow \mathit{list/show}\ \mathit{int/show}}\ \text{VAR}}{\{\alpha{=}\mathit{int}\} \mid \Gamma \vdash \mathit{show} \overset{\leftarrow}{:} \mathit{list}\ \mathit{int} \to \mathit{string}}$$

where the elaborated *list/show* is passed its implicit parameter as *int/show* (which is itself elaborated from ?*show*). Here we also see the importance of applying Q_1 (as $Q_1[\tau_1]$) in the [INST-IMPLICIT] rule: in the above derivation this propagates the element type α as $\{\alpha{=}\mathit{int}\}[\alpha{\to}\mathit{string}] = \mathit{int}{\to}\mathit{string}$ to resolve the implicit ?*show* parameter recursively in the [VAR-QUALIFY] rule – where it can now be uniquely elaborated to *int/show*. As another example, the expression *show* [[1], [2]] from the introduction is recursively elaborated to *list/show* (*list/show int/show*) [[1], [2]]. As an aside, we do not need eta-expansion here since in our calculus all implicit parameters come first and thus we can use partial applications instead.

2.7 Implicit Parameter Binding

Before we give the full type inference rules, we first consider the inference ($\overset{\leftarrow}{:}$) rule for implicit bindings:

$$\frac{Q \mid \Gamma, z : \alpha \vdash t \mathrel{\vec{:}} \rho \quad \text{fresh } \alpha}{Q \mid \Gamma \vdash \lambda ?z.t \mathrel{\vec{:}} ?\hat{z} : \alpha \to \rho}\ \textsc{implicit}$$

Just like a regular lambda binding, the name z is added to the environment Γ and is in scope when checking the body t. Following the rules for type inference under a prefix, we assign an abstract type α to z which can be bound in the result prefix Q (if so required).

Even though we can bind an implicit parameter with a qualified name z, we still always use a unqualified plain name $\hat{z}$ in the type: at each call site we always look for the unqualified name. This is important when using multiple implicit parameters of the same name. Consider for example a *show* function for tuples:

let $\mathit{tuple/show} = \lambda ?\mathit{fst/show}.\, \lambda ?\mathit{snd/show}.\, \lambda x.\ \ \mathit{fst/show}\ (\mathit{fst}\ x) +\!\!+ "," +\!\!+ \mathit{snd/show}\ (\mathit{snd}\ x)$
in $\mathit{show}\ (1, \mathit{True})$

where $\mathit{tuple/show}$ has type $\forall \alpha\beta.\ ?\mathit{show}{:}(\alpha \to \mathit{string}) \to ?\mathit{show}{:}(\beta \to \mathit{string}) \to (\alpha, \beta) \to \mathit{string}$. Here we use qualified implicit parameters to distinguish showing each component of the tuple in the body, while at a call site we still resolve each one individually as a plain *show*.

The *show* function for tuples nicely illustrates how static overloading and implicit parameters can really help in practice: consider an expression like $\mathit{show}\ [([1], \mathit{True})]$ which is automatically elaborated to $\mathit{list/show}\ (\mathit{tuple/show}\ (\mathit{list/show}\ \mathit{int/show})\ \mathit{bool/show})\ [([1], \mathit{True})]$ – significantly reducing the programmer's burden.

2.8 Full Type Rules

Figure 5 gives the full type rules under a prefix. These are based mostly on the rules given by Leijen and Ye [2025] and we refer to that work for in-depth discussion on the design. For our purposes, the variable and instantiation rules are new, and we added the [IMPLICIT] rule. For clarity, the rules are stated without rewriting to an elaborated expression, but sometimes we make this explicit. In particular, the full [VAR] rule would be:

$$\frac{\exists! e.\ \ Q \mid \Gamma \Vdash z \mathrel{\overleftarrow{:}} \tau \ \rightsquigarrow e}{Q \mid \Gamma \vdash z \mathrel{\overleftarrow{:}} \tau \ \rightsquigarrow e}\ \textsc{var}$$

where it elaborates z to e. For all other rules the elaboration is trivial as only the [VAR] rule qualifies names and adds implicit parameters – all other expressions just propagate the elaboration of their components in the obvious way. For example, the full rule for [FUN] with elaboration is:

$$\frac{Q \mid \Gamma, z : \tau_1 \vdash e \mathrel{\overleftarrow{:}} \tau_2 \ \rightsquigarrow e'}{Q \mid \Gamma \vdash \lambda z.e \mathrel{\overleftarrow{:}} \tau_1 \to \tau_2 \ \rightsquigarrow \lambda z.e'}\ \textsc{fun}$$

An interesting aspect is that almost all rules are checking rules ($\mathrel{\overleftarrow{:}}$) with just three exceptions for [INF], [GEN], and [IMPLICIT] which are inference rules ($\mathrel{\vec{:}}$). In the [LET] rule, the type of the binding x must be inferred and we use [GEN] to infer a most general type, which uses in turn [INF] and [IMPLICIT] to infer a mono type. The [INF] rule infers a type by switching to checking if the expression can be typed with a fresh monotype α.

The [APP-ARG] rule types applications where first the function expression is checked as $\alpha \to \tau$, and then the argument is checked where we propagate the discovered type information as $Q_1[\alpha]$. This order does not always work though as often we apply an overloaded variable and we need to first check the argument expression to discover enough type information to disambiguate the variable. Consider for example $\mathit{show}\ [1]$. This is where the [APP-VAR] rule comes in. This rule is always preferred over [APP-ARG] when it can apply, and it matches syntactically over variable applications $z\ e_1 \ldots e_n$.

$$\underset{\substack{\downarrow\\ \text{out}}}{Q} \mid \Gamma \vdash e \overset{\leftarrow}{:} \tau \quad , \quad \underset{\substack{\downarrow\ \uparrow\\ \text{out in}}}{Q} \mid \Gamma \vdash e \overset{\rightarrow}{:} \tau \quad , \quad \underset{\substack{\downarrow\ \uparrow\ \uparrow\\ \text{out in in}}}{Q} \mid \Gamma \vdash t \overset{\rightarrow}{:} \rho \quad , \quad \underset{\substack{\downarrow\ \uparrow\ \uparrow\ \downarrow\\ \text{out in in out}}}{Q} \mid \Gamma \vdash_{\text{GEN}} t \overset{\rightarrow}{:} \sigma \quad \text{with } \vDash Q$$

We use $\overset{\leftarrow}{:}\ \tau$ for checking type τ, and $\overset{\rightarrow}{:}\ \tau$ for infering type τ.

$$\frac{\exists! e.\ Q \mid \Gamma \Vdash z \overset{\leftarrow}{:} \tau \rightsquigarrow e}{Q \mid \Gamma \vdash z \overset{\leftarrow}{:} \tau}\text{VAR} \qquad \frac{Q_1 \mid \Gamma \vdash e \overset{\leftarrow}{:} \tau_1 \quad Q_2 \vdash \tau_1 \approx \tau_2}{Q_1, Q_2 \mid \Gamma \vdash (e : \tau_1) \overset{\leftarrow}{:} \tau_2}\text{ANN}$$

$$\frac{Q \mid \Gamma, z : \tau_1 \vdash e \overset{\leftarrow}{:} \tau_2}{Q \mid \Gamma \vdash \lambda z.e \overset{\leftarrow}{:} \tau_1 \to \tau_2}\text{FUN} \qquad \frac{Q_1 \mid \Gamma \vdash \lambda z.e \overset{\leftarrow}{:} \alpha_1 \to \alpha_2 \quad Q_2 \vdash \alpha_1 \to \alpha_2 \approx \alpha \ \text{ fresh } \alpha_1, \alpha_2}{Q_1, Q_2 \mid \Gamma \vdash \lambda z.e \overset{\leftarrow}{:} \alpha}\text{IFUN}$$

$$\frac{Q \mid \Gamma \vdash e \overset{\leftarrow}{:} \alpha \quad \text{fresh } \alpha}{Q \mid \Gamma \vdash e \overset{\rightarrow}{:} \alpha}\text{INF} \qquad \frac{Q \mid \Gamma, z : \alpha \vdash t \overset{\rightarrow}{:} \rho \quad \text{fresh } \alpha}{Q \mid \Gamma \vdash \lambda ?z.t \overset{\rightarrow}{:} ?\hat{z} : \alpha \to \rho}\text{IMPLICIT}$$

$$\frac{Q_1 \mid \Gamma \vdash_{\text{GEN}} t \overset{\rightarrow}{:} \sigma \quad Q_2 \mid \Gamma, z : \sigma \vdash e \overset{\leftarrow}{:} \tau}{Q_1, Q_2 \mid \Gamma \vdash \text{let } z = t \text{ in } e \overset{\leftarrow}{:} \tau}\text{LET} \qquad \frac{Q_0 \mid \Gamma \vdash t \overset{\rightarrow}{:} \rho \quad (Q, \sigma) = \text{gen}(Q_0, \Gamma, \rho)}{Q \mid \Gamma \vdash_{\text{GEN}} t \overset{\rightarrow}{:} \sigma}\text{GEN}$$

$$\frac{Q_1 \mid \Gamma \vdash e_1 \overset{\leftarrow}{:} \alpha \to \tau \quad Q_2 \mid \Gamma \vdash e_2 \overset{\leftarrow}{:} Q_1[\alpha] \quad \text{fresh } \alpha}{Q_1, Q_2 \mid \Gamma \vdash e_1\ e_2 \overset{\leftarrow}{:} \tau}\text{APP-ARG}$$

$$\frac{\begin{array}{l}\text{least } i \text{ with } 0 \leqslant i \leqslant n \quad \text{fresh } \alpha_{i+1}, \ldots, \alpha_n \\ Q_1 \mid \Gamma \vdash e_1 \overset{\rightarrow}{:} \tau_1 \quad \ldots \quad Q_i \mid \Gamma \vdash e_i \overset{\rightarrow}{:} \tau_i \quad Q = Q_1, \ldots, Q_i \\ Q_0 \mid \Gamma \vdash z \overset{\leftarrow}{:} Q[\tau_1] \to \ldots \to Q[\tau_i] \to \alpha_{i+1} \to \ldots \to \alpha_n \to \tau \\ Q_{i+1} \mid \Gamma \vdash e_{i+1} \overset{\leftarrow}{:} Q_0[\alpha_{i+1}] \quad \ldots \quad Q_n \mid \ \Gamma \vdash e_n \overset{\leftarrow}{:} Q_0[\alpha_n]\end{array}}{Q_0, Q_1, \ldots, Q_n \mid \Gamma \vdash z\ e_1 \ldots e_i \ldots e_n \overset{\leftarrow}{:} \tau}\text{APP-VAR}$$

$$\begin{array}{lll}\text{gen} : (Q, \Gamma, \sigma) \to (Q, \sigma) & & \\ \text{gen}(Q \cdot \alpha{=}\tau,\ \Gamma,\ \rho) & = \text{gen}(Q,\ \Gamma,\ [\alpha{:=}\tau]\rho) & \text{if } \alpha \notin \text{ftv}(Q, \Gamma) \\ \text{gen}(Q,\ \Gamma,\ \sigma) & = \text{gen}(Q,\ \Gamma,\ \forall\alpha.\sigma) & \text{if } \alpha \notin \text{ftv}(Q, \Gamma) \ \wedge \alpha \in \text{ftv}(\sigma) \\ \text{gen}(Q,\ \Gamma,\ \sigma) & = (Q, \sigma) & \text{if } (\text{dom}(Q) \cup \text{ftv}(\sigma)) \subseteq \text{ftv}(\Gamma)\end{array}$$

Fig. 5. Syntax-directed bidirectional type checking rules for static overloading with syntactic implicit parameters. We always prefer [APP-VAR] over [APP-ARGC] when applicable.

The [APP-VAR] rule looks a bit intimidating but essentially it just tries to infer the least amount of arguments i such that we can disambiguate z, and then propagates the remaining argument types into the remaining argument expressions. This strategy is straightforward to implement: first try to disambiguate z (without any inference of the arguments) and keep inferring one argument at a time until z can be disambiguated, and eventually use checking rules for the remaining arguments. Even though the rule has the drawback of a left-to-right bias, it seems to work well in practice with the Koka language (but as remarked by Leijen and Ye [2025] further refinements are possible).

These type rules are all fully determined by the syntax (and the shape of the propagated type in [FUN]/[IFUN]). Moreover, the rules are carefully constructed such that the (result) prefix Q only contains constraints induced by the structure of the program and types: at all leaf nodes in the derivation Q is empty, except for [EQ-VAR]. As a consequence, any well-typed program has only a

single possible derivation:

Theorem 2.3. (*Principal type derivations*)
For any derivations $Q \mid \Gamma \vdash_{\text{GEN}} e \overrightarrow{:} \sigma \rightsquigarrow e_1$ and $Q' \mid \Gamma \vdash_{\text{GEN}} e \overrightarrow{:} \sigma' \rightsquigarrow e_2$, we have $Q = Q'$, $\sigma = \sigma'$, and $e_1 = e_2$.

See the proof in Appendix B.3 of the supplement. This theorem is important as it also implies that the elaboration is deterministic with coherent semantics.

2.9 Scope Based Disambiguation

The current rules do not take the scope of a qualified name into account. Sometimes that can be inconvenient as we would like to be able to override definitions in an outer scope. For example,

```
fun secret-list()
  fun myint/show(i:int) = "*"
  show([1,2])
```

where we would like to elaborate to `list/show([1,2],myint/show)`. However, with the current rules this example would be rejected as both `int/show` and `myint/show` match. In such cases though, one might expect instead to prefer the definition that originated from an inner scope.

We can formalize this by changing the [VAR] rule to instead prefer a minimal solution instead of unique solution. Suppose we annotate every variable z with its scope depth i as z^i. The global scope depth is 0 and increments inside every let bound definition. We write $e \preccurlyeq e'$ if an elaborated expression e is either equal to e', or resolved to an inner scope relative to e'. We define this as:

$$z^i\ e_1 \ldots e_n \preccurlyeq z'^j\ e'_1 \ldots e'_k \quad = \quad i > j \ \vee (z = z' \ \wedge e_1 \preccurlyeq e'_1 \ \wedge \ldots \ \wedge e'_n \preccurlyeq e'_n)$$

(where we note that if $z = z'$ then $n = k$ for elaborated expressions). Let's write the condition on the current [VAR] as $P(Q, e) \ = \ Q \mid \Gamma \Vdash z \overleftarrow{:} \tau \rightsquigarrow e$. We can then then rephrase [VAR] as:

$$\frac{\exists! e.\ P(Q, e) \ \wedge (\forall Q' e'.\ P(Q', e') \implies e \preccurlyeq e')}{Q \mid \Gamma \vdash z \overleftarrow{:} \tau}\ \text{VAR-SCOPE}$$

Our Koka implementation uses this rule instead of [VAR] to prioritize qualified definitions in an inner scope.

2.10 Coherence and Stability

Schrijvers et al. [2019] present the Cochis calculus of coherent implicits. In this paper they argue that systems that support implicit programming should be *coherent* and *stable*.

Coherence is defined as a valid program always having exactly one meaning. Our system is coherent: this follows from the [VAR] rule as it requires a unique elaboration in combination with Theorem 2.3 which guarantees unique derivations. As an example of coherence, Schrijvers *et al* present a classic example from Haskell as *show* (*read* "3"). Such expressions are rejected in Haskell as the type class resolution is ambigious. For example, we could use instances for *Int* or *Bool* and many others to resolve the *read*. Similarly, assuming appropiate qualified *read* definitions, our system would reject this since the type context cannot resolve *read* unambiguously. Unlike Haskell though, we can disambiguate the example explicitly without needing a type signature, for example as *show* (*int/read* "3"). Since overloading in our system is just elaboration to fully qualified names we can always disambiguate manually in our system without needing extra type annotations.

Schrijvers et al. [2019] also argue that implicit type systems should be *stable*, where instantiation of type variables does not affect resolution. In essence, this means that using an overloaded variable at a more specific type does not change the semantics. This is not always the case for systems that would disambiguate based on a most specific type for example. We can show stability in our system formally over the variable resolution rules:

Theorem 2.4. (*Stability*)
If $Q_1 \mid \Gamma \vdash z \overset{\leftharpoonup}{:} \tau \;\rightsquigarrow e$ with $\vDash (\theta, Q_1)$, then we also have $Q_2 \mid \theta\Gamma \vdash z \overset{\leftharpoonup}{:} \theta\tau \;\rightsquigarrow e$ (with $Q_1 \sqsubseteq (\theta, Q_2)$).

See the proof in Appendix B.4 of the supplement. However, in the Cochis system the notion of stability is stronger than this. They show the following example in Haskell where overlapping instances are allowed:

class $Trans\ \alpha$ where $trans\ ::\ \alpha \to \alpha$
instance $Trans\ \alpha$ where $trans\ x\ =\ x$
instance $Trans\ Int$ where $trans\ x\ =\ x + 1$

with the definition

$bad\ ::\ \forall\alpha.\ \alpha \to \alpha$
$bad\ x\ =\ trans\ x$

Due to the type signature, the type class can be resolved unambiguously to the generic one. However, that also means that *bad* 1 evaluates to 1, while *trans* 1 evaluates to 2, and it is argued this breaks equational reasoning. We do not necessarily agree with this: as we saw before, a type annotation is essential to the Haskell semantics (as it does not have a dynamic untyped semantics), and we cannot just leave it out and replace *bad* with *trans* directly. In our system there is a similar situation where we can only do equational reasoning on fully qualified identifiers. Suppose we define:

$generic/trans\ x = \ x$ and $int/trans\ x = \ x + 1$

then the expression:

$bad\ =\ \lambda x.\ trans\ x$

would be rejected as it is ambigious. We need to either qualify the *trans* variable, as for example:

$generic/bad = \ \lambda x.\ generic/trans\ x$ or $int/bad = \ \lambda x.\ int/trans\ x$

or propagate *trans* explicitly as an implicit parameter:

$bad\ =\ \lambda ?trans.\ \lambda x.\ trans\ x$

In the last case, *bad* 1 indeed equals *trans* 1 (as it gets elaborated to *bad int/trans* 1).

But what about the original example in Cochis with the polymorphic type signature:

$bad\ :\ \forall\alpha.\ \alpha \to \alpha\ =\ \lambda x.\ trans\ x$

In this case, we can argue there are two acceptable solutions. The first design (I) is that *trans* should be resolved to *generic/trans* since the type of the argument is an abstract α and thus only the *generic/trans* can match. The other argument (II), as made in Cochis, is that this example should be rejected as the instantiation of $\forall\alpha.\ \alpha \to \alpha$ with *int* would make this example ambiguous again (i.e. it is not stable under type application).

Cochis uses a separate stable predicate to check for this form of ambiguity, but it turns out that we can modularly describe either design in our system as two variants of the type scheme checking rule. First we extend the syntax to allow type scheme annotations for σ (with $\mathrm{ftv}(\sigma) \subseteq \mathrm{ftv}(\Gamma)$) as let $z : \sigma = t$ in e:

$$\frac{Q_1 \mid \Gamma \vdash t \overset{\leftharpoonup}{:} \sigma \qquad Q_2 \mid \Gamma, z : \sigma \vdash e \overset{\leftharpoonup}{:} \tau}{Q_1, Q_2 \mid \Gamma \vdash \mathsf{let}\ z : \sigma = t\ \mathsf{in}\ e \overset{\leftharpoonup}{:} \tau}\ \textsc{let-ann}$$

For checking mono-types, we already have the [ANN] rule. For ρ types, we extend this now to check implicit parameters:

$$\frac{\hat{z} = x \qquad Q \mid \Gamma, z : \tau \vdash t \overset{\leftharpoonup}{:} \rho}{Q \mid \Gamma \vdash \lambda ?z.t \overset{\leftharpoonup}{:} ?x{:}\tau \to \rho}\ \textsc{ann-implicit}$$

This leaves us with checking polymorphic quantifiers $\forall\alpha$. Usually, such checking rules are expressed algorithmically by replacing the polymorphic type variables with fresh type constants c (also called *skolem* constants [Peyton Jones et al. 2007]):

$$\frac{Q_1 \mid \Gamma \vdash\ t \mathrel{\tilde{:}} [\overline{\alpha}{:=}\overline{c}]\rho \qquad (Q_2, ()) = \mathsf{gen}(Q_1, \Gamma, ()) \qquad \overline{c} \notin Q_2 \qquad \mathsf{fresh}\ \overline{c}}{Q_2 \mid \Gamma \vdash\ t \mathrel{\tilde{:}} \forall\overline{\alpha}.\rho}\ \textsc{ann-skolem}$$

The condition $\overline{c} \notin Q_2$ ensures that the fresh type constants do not escape the scope of their binding. We need to use gen here to remove any unused constraints mentioning $\overline{c}$ (which can occur due to [inst-quantify]). Consider for example:

$$\frac{\dfrac{\dfrac{id : \forall\alpha.\alpha{\to}\alpha \in \Gamma \qquad \dfrac{\{\alpha{=}c\} \mid \Gamma \vdash\ \alpha{\to}\alpha \approx c{\to}c \qquad \mathsf{fresh}\ \alpha}{\{\alpha{=}c\} \mid \Gamma \vdash\ \forall\alpha.\alpha{\to}\alpha \sqsubseteq c{\to}c \ \rightsquigarrow\ \cdot}}{\exists!\, id.\ \{\alpha{=}c\} \mid \Gamma \Vdash\ id \mathrel{\tilde{:}} c{\to}c \ \rightsquigarrow\ id}\ \textsc{var}}{\{\alpha{=}c\} \mid \Gamma \vdash\ id \mathrel{\tilde{:}} c{\to}c \qquad (\varnothing, \_) = \mathsf{gen}(\{\alpha{=}c\}, \Gamma, ()) \qquad \overline{c} \notin \varnothing}\ \textsc{var}}{\varnothing \mid \Gamma \vdash\ id \mathrel{\tilde{:}} \forall\alpha.\alpha{\to}\alpha}$$

This follows design (I) where our example gets resolved to *generic/trans* as the integer alternative cannot match with the abstract skolem constant (i.e. $\not\vdash\ c \approx \mathit{int}$). This is the rule that is used in the Koka implementation.

However, we can also implement design (II) by not using skolem constants in the first place, and instead checking *afterwards* whether a type variable was indeed used polymorphically [Leijen 2008]:

$$\frac{Q_1 \mid \Gamma, \overline{\alpha} \vdash\ t \mathrel{\tilde{:}} \rho \qquad (Q_2, \forall\overline{\alpha}.\rho) = \mathsf{gen}(Q_1, \Gamma, \rho) \qquad \mathsf{fresh}\ \overline{\alpha}}{Q_2 \mid \Gamma \vdash\ t \mathrel{\tilde{:}} \forall\overline{\alpha}.\rho}\ \textsc{ann-scheme}$$

Here, we instantiate the polymorphic type as regular fresh type variables, but check afterwards that these did not unify with any type, nor escaped the scope through any other type constraints. To prevent generalizing early over the $\overline{\alpha}$ binders, we extend Γ to include type variable bindings as well as $\Gamma, \overline{\alpha}$ (which can be seen as a shorthand for a sequence of anonymous bindings as $\Gamma, \overline{\_ : \alpha}$).

Interestingly, for standard Hindley-Milner either approach is valid and it makes no difference. In our case though, by keeping $\overline{\alpha}$ as regular type variables, higher up in the derivation there can now be multiple matches for overloads, and we may no longer satisfy the unique $\exists!e$ condition in the [var] rule. In particular, for the *trans* example, both *generic/trans* and *int/trans* now match (since $\{\alpha{:=}\mathit{int}\} \vdash\ \alpha \approx \mathit{int}$), and the example is rejected due to ambiguity – corresponding to the second design (as advocated by Cochis).
In Cochis, the stability property is shown by translating from the core calculus into System-F and showing that static reduction of type application preserves typing. For our case this stategy does not quite work since once we elaborate to fully qualified names we already have an untyped dynamic semantics. We can show stability more directly though over the [ann-scheme] rule:

Theorem 2.5. (*Polymorphic Stability*)
If $Q_1 \mid \Gamma \vdash\ t \mathrel{\tilde{:}} \forall\overline{\alpha}.\rho \ \rightsquigarrow e$ (using [ann-scheme]), we also have $Q_2 \mid \Gamma \vdash\ t \mathrel{\tilde{:}} [\overline{\alpha}{:=}\overline{\tau}]\rho \ \rightsquigarrow e$ (for any $\overline{\tau}$ with $\mathsf{ftv}(\overline{\tau}) \subseteq \mathsf{ftv}(\Gamma)$).

See Appendix B.4 of the supplement for the proof.

2.11 Definition Stability

There is another form of stability that is quite important in practice, which we call definition stability. We call a system *definition stable* if the semantics of an existing function does not change

silently when adding a new definition. Consider for example the following Haskell program:

class *Trans* α where *trans* : $\alpha \to \alpha$
instance *Trans* α where *trans* $x = x$
foo = *trans* 1

where *foo* = 1. However, suppose we enable overlapping instances and add a more specific definition (maybe in an imported module):

instance *Trans Int* where *trans* $i = i + 1$

In this case, the *trans* is resolved now to the most specific instance and *foo* is now 2 (and thus Haskell with overlapping instances is not definition stable). Dually, Scala prefers the most general instance for implicits and is also not definition stable. In our system, adding either definition always results in an ambiguity error (and is therefore definition stable). Even with the extension to scope baced disambiguation (Section 2.9) our system is still definition stable as any outer definition is never preferred to an inner one.

3 TYPE INFERENCE

We essentially extended the standard type rules with just two new rules for implicit parameter instantiation ([INST-IMPLICIT]) and variable overloading ([VAR-QUALIFY]). We saw that even though each rule is simple, their recursive nature makes the rules quite expressive. Perhaps even *too* expressive since the potential recursion between these rules can easily lead to infinite derivations! Consider for example a function *foo* that requires itself as an implicit parameter, as *foo* : *?foo*:$(int \to int) \to int \to int$. This is a problem for type inference as trying to derive an application like *foo* 1 would lead to infinite recursion.

3.1 Undecidability of Type Checking

As it is, general type checking for our current rules turns out to be undecidable! In particular, we can show it is possible to encode a Turing machine on the type level, where type checking becomes equivalent to showing termination of the encoded Turing machine. See Appendix A of the supplement. for an example of encoding of a 3-state busy-beaver Turing machine [Rado 1962] in Koka. In particular, we can create types for the symbols 0 and 1, and the tape of the machine as:

```
type sym0                type cons<a,b>  // tape with head a and tail b
type sym1                type inf        // infinite tape of zeros
```

Furthermore, we can create machine states and a full Turing machine configuration as:

```
type state-a             type state-b              type state-c
abstract type config<s,l,r> = Start  // the machine config: <state, left tape, right tape>
val start : config<state-a,inf,inf> = Start  // starting machine configuration with all zeros
```

We can then encode the machine state transitions in the types. For example, for the 3-state busy-beaver, if we are in state A with the head being 0, we write 1 to the head, shift the tape to the right, and continue in state B:

```
fun ab/transition( st : config<state-a,cons<x,l>,cons<sym0,r>>,
                   ?transition : (config<state-b,l,cons<x,cons<sym1,r>>>) -> () ) : () = ()
```

We write a overloaded function for each transition, as well as a halting rule (in state C):

```
fun cend/transition( st : config<state-c,l,cons<sym1,r>> ) : () = ()
```

See the full example in Appendix A of the supplement where we also show how can expand the `inf` tape automatically to `cons<sym0,inf>` when so required. We can then start the machine as `transition(start)`, where static overloading and implicit parameters elaborate the plain `transition` to a full sequence of all state transitions – for the 3-state busy-beaver machine this elaborates to 14 state transitions.

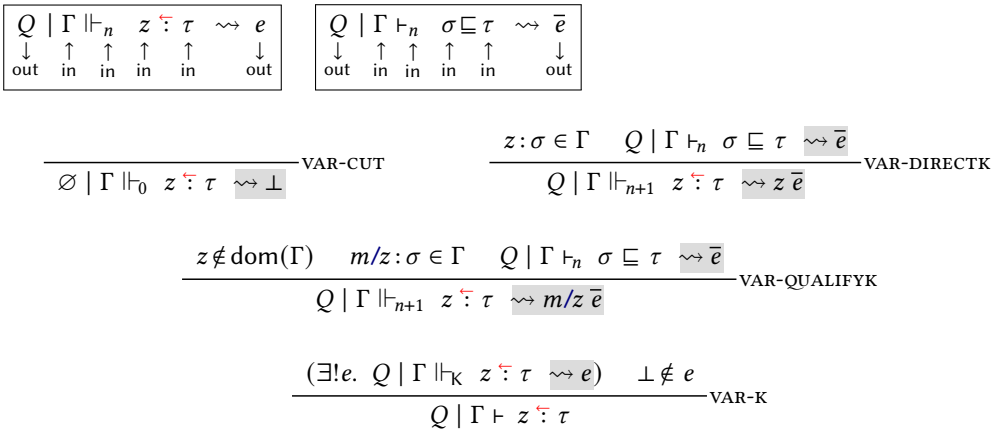

Fig. 6. Type rules with finite derivations (up to depth K). Instantiation passes n unchanged.

3.2 Recovering Decidability

In order to make the system decidable we need to impose some decreasing measure on the derivations to ensure termination of the type checking algorithm. One way to do this is to simply limit the depth of the derivation tree ending in a [VAR] to be of at most K deep for some fixed K.

Figure 6 gives a new set of rules for variable derivations (and instantiation) as $Q \mid \Gamma \Vdash_n z \Leftarrow \tau$ where we pass a decreasing bound $n \in \mathbb{N}$. Each application of [VAR-DIRECTK] or [VAR-QUALIFYK] decreases n (while n is passed unchanged through instantiation). Both [VAR-DIRECTK] or [VAR-QUALIFYK] can now only be applied for $n > 0$. In the case $n = 0$, the [VAR-CUT] rule applies which always elaborates to an "infinite" expression $\bot$. The new [VAR-K] rule is now extended to always reject any solutions that contain $\bot$ expressions.

These rules are clearly terminating since n strictly decreases. As such they are incomplete with respect to the original rules since some derivations are now rejected as too deep. We do have soundness though: any valid derivation using [VAR-K] is also valid in the original rules:

Theorem 3.6. (*Soundness of the Finite Rules*)
If $Q \mid \Gamma \vdash z \Leftarrow \tau \rightsquigarrow e$ with [VAR-K], then also $Q \mid \Gamma \vdash z \Leftarrow \tau \rightsquigarrow e$ with [VAR].

See the proof in Appendix B.2 of the supplement. It turns out that the soundness property is a bit subtle. In particular, in an earlier design we left out the rule [VAR-CUT] as it seems we can just not have any valid derivation that requires $n = 0$. This would be unsound with respect to the original rules though!

Currently, any potential infinite derivation has a solution containing $\bot$ using [VAR-CUT]. As such, it might be that the $\exists!e$ condition in the [VAR-K] rule is now not satisfied as valid finite solutions compete with such "infinite" solutions (which is also why the $\bot \notin e$ condition is outside the scope of the uniqueness condition). If we leave out the [VAR-CUT] rule though, there would be no valid derivation instead for such "infinite" ones, and in that case a particular finite solution may suddenly be unique and accepted. In such case though, the original rules may still find a finite derivation (of depth > K) for our "infinite" one and reject it (as it becomes ambigious). Although we are the first to formalize this, this issue with soundness was also remarked by White et al. [2015].

3.3 Maintaining History

Of course, our formalization based on a fixed depth K of [VAR-DIRECTK] or [VAR-QUALIFYK] applications is clear but perhaps also a bit naive in practice. However, we can see that in principle any

decreasing measure would suffice to limit the depth of the derivation tree.

As our approach is name based, we observe that for any infinite recursion to happen, it must be that we try to resolve the same qualified name more than once. Instead of a fixed K, we can instead pass a *history* H of each name $m/z : \tau$ in [VAR-QUALIFYK] (and $z : \tau$ in [VAR-DIRECTK]). One way to ensure termination is to simply reject any derivation that leads to a repeated name z in the history. That would be too restrictive though as it would reject for example *show* [[1]] (where *list*/*show* is resolved twice).

A better way is to only allow a derivation for $z \mathrel{\overset{\leftarrow}{:}} \tau$ to proceed if for the last $z : \tau'$ appearing in the history, we have that τ is "smaller" than τ'. We can define smaller here for example as the count of constructors in the type. This allows our earlier *show* [[1]] to be accepted again.

In the Koka implementation, we refine this a bit further where we require for $z \mathrel{\overset{\leftarrow}{:}} \tau$ that there are either fewer than N entries for z in the history, and otherwise that the τ is smaller than any one of the last N entries $z : \tau'$ in the history (where $N = 4$ in our case). This allows for some limited recursion where the size of the type stays can the same (or be larger) for some iterations. For example, our implementation can now accept the 3-state busy beaver problem we discussed earlier (but of course, still not arbitrary Turing machines).

3.4 Implementing Type Inference

As remarked by Leijen and Ye [2025], making the type rules syntax directed and under a prefix, we can essentially "read off" the inference algorithm. This make the formalism particularly useful not only for users to have a precise specification, but also for compiler implementors to ensure it is implemented accordingly. As shown by Leijen and Ye [2025], we can represent valid prefixes in a canonical form as a regular substitution, and implement unification and composition of two prefixes as $\mathit{unify}(\tau_1, \tau_2) : Q$ and $\mathit{compose}(Q_1, Q_2) : Q$ (see Appendix C of the supplement). These call an operation fail() of an effect handler [Leijen 2021] if no unification exists or when the union of the two prefixes does not have a valid solution. We can now directly implement the $Q \mid \Gamma \vdash_n \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ rules for instantiation as:

$$
\begin{array}{lll}
\mathit{inst} : (\Gamma, \mathbb{N}, \tau, \tau) \to (Q, \overline{e}) & & \\
\mathit{inst}(\Gamma, n, \tau_1, \tau_2) & = (\mathit{unify}(\tau_1, \tau_2), \cdot) & \\
\mathit{inst}(\Gamma, n, \forall \alpha.\sigma, \tau) & = \mathit{inst}(\Gamma, n, \sigma[\alpha{:=}\mathsf{fresh}()], \tau) & \\
\mathit{inst}(\Gamma, n, ?x{:}\tau_1 \to \rho, \tau_2) & = \mathsf{val}\ (Q_1, \overline{e}) = \mathit{inst}(\Gamma, n, \rho, \tau_2);\ (Q_2, e) = \mathit{resolve}(\Gamma, n, x, Q[\tau_1]) & \\
 & \quad \mathsf{in}\ (\mathit{compose}(Q_1, Q_2),\ e\ \overline{e}) &
\end{array}
$$

Here we assume an another operation fresh to generate fresh type variables. Each definition corresponds immediately to the instantiation rules [INST-MONO], [INST-QUANTIFY], and [INST-IMPLICIT]. That leaves the *resolve* function to implement the $Q \mid \Gamma \Vdash_n z \mathrel{\overset{\leftarrow}{:}} \tau \rightsquigarrow e$ variable rules:

$$
\begin{array}{lll}
\mathit{resolve} : (\Gamma, \mathbb{N}, z, \tau) \to (Q, e) & & \\
\mathit{resolve}(\Gamma, 0, z, \tau) & = \mathsf{infinite}() & \\
\mathit{resolve}(\Gamma, n, z, \tau) \mid z : \sigma \in \Gamma & = \mathsf{val}\ (Q, \overline{e}) = \mathit{inst}(\Gamma, n-1, \sigma, \tau)\ \mathsf{in}\ (Q, z\ \overline{e}) & \\
\mathit{resolve}(\Gamma, n, z, \tau) & = \mathsf{val}\ (m/z, \sigma) = \mathsf{forallq}(\Gamma, z);\ (Q, \overline{e}) = \mathit{inst}(\Gamma, n-1, \sigma, \tau)\ \mathsf{in}\ (Q, m/z\ \overline{e}) &
\end{array}
$$

Again, each definition corresponds to the variable rules [VAR-CUT], [VAR-DIRECTK], and [VAR-QUALIFYK]. However, we see two new operations here: infinite and forallq. The call to infinite() returns a bottom expression as $(\varnothing, \bot)$, but forallq(Γ, z) is special as it should return a list of *all* $m/z : \sigma \in \Gamma$. Here is where we can use an effect handler to neatly express this in a modular way while keeping the implementations of *inst* and *resolve* as close as possible to the type rules. In particular, we can

implement *resolve* under a handler that implements a list of successes [Philip Wadler 1985]:

$$
\begin{array}{l}
\mathit{explore} \,:\, (\Gamma, z, \tau) \to [(Q, e)] \\
\mathit{explore}(\Gamma, z, \tau) = \ \mathsf{handle}\ \mathit{resolve}(\Gamma, \mathsf{K}, z, \tau) \\
\quad \mathsf{return}\ x \quad\ \to [x] \\
\quad \mathsf{fail}() \quad\ \to [] \\
\quad \mathsf{infinite}() \quad \to \mathsf{resume}((\varnothing, \bot)) \\
\quad \mathsf{forallq}(\Gamma, z) \ \to [x \mid m/z : \sigma \in \Gamma,\ x \leftarrow \mathsf{resume}(m/z, \sigma)]
\end{array}
$$

Normal results (return x) are wrapped in a singleton list $[x]$, while failures return an empty list. The infinite() operation just resumes at the call site with a bottom expression (as $(\varnothing, \bot)$). The forallq(Γ, z) operation is the most interesting one as it resumes for each $m/z : \sigma \in \Gamma$ and appends all returned solutions. In the type checking algorithm *check*, we can now call *explore* to implement the variable case as:

$$
\begin{array}{l}
\mathit{check} \,:\, (\Gamma, e, \tau) \to (Q, e) \\
\ldots \\
\mathit{check}(\Gamma, z, \tau) \ = \ \mathsf{match}\ \mathit{explore}(\Gamma, z, \tau) \\
\quad [(Q, e)] \ \mid \bot \notin e \to (Q, e) \qquad\qquad \textit{unique and not } \bot \\
\quad [] \qquad\qquad\ \to \mathsf{fail}() \qquad\qquad \textit{unresolved} \\
\quad \_ \qquad\qquad\ \to \mathsf{fail}() \qquad\qquad \textit{ambigious or infinite}
\end{array}
$$

This implements the $\exists! e$ condition by essentially exploring all possible derivations and returning normally only if there is a single successful derivation. The connection between using a list of successes and the $\exists! e$ condition may not be immediately apparent, but we can show that the algorithm is sound and complete with respect to our finite type rules:

Theorem 3.7. (*Algorithmic Soundness*)
If $\mathit{check}(\Gamma, z, \tau) = (Q, e)$, then also $Q \mid \Gamma \vdash z \mathbin{\reflectbox{:}} \tau \rightsquigarrow e$ (with [VAR-K]).

Theorem 3.8. (*Algorithmic Completeness*)
If $Q \mid \Gamma \vdash z \mathbin{\reflectbox{:}} \tau \rightsquigarrow e$ (with [VAR-K]), then also $\mathit{check}(\Gamma, z, \tau) = (Q, e)$.

See the proofs in Appendix B.1 of the supplement.

3.5 Optimizing Exploration

Even though the previous theorems show our algorithm is correct, it is not the most efficient implementation. We can make various improvements to the *explore* function to optimize the exploration of the search space.

First, we observe that we never actually look at the individual solutions if there is more than one solution. As such, we only consider a valid singleton solution $[(Q, e)]$, an empty solution $[]$, or multiple solutions. We can represent this more efficiently using a specialized data type:

$$\mathsf{type}\ \mathit{sol}\ \alpha \ = \ \mathsf{One}(\alpha) \mid \mathsf{None} \mid \mathsf{Amb}$$

Furthermore, we also observe that we never accept any infinite solution with $\bot \in e$. Let's only use One$((Q, e))$ if $\bot \notin e$, and directly use Amb for infinite solutions as well. We can then define *check* as:

$$
\begin{array}{l}
\mathit{check}(\Gamma, z, \tau) \ = \ \mathsf{match}\ \mathit{explore}_{\mathit{opt}}(\Gamma, z, \tau) \\
\quad \mathsf{One}(x) \to x \\
\quad \_ \qquad\ \to \mathsf{fail}()
\end{array}
$$

We do not need to change the definitions of *resolve* or *inst* as these use abstract operations – instead we only need to modify the handler to use our new representation:

$$
\begin{array}{l}
\mathit{explore}_{\mathit{opt}}(\Gamma, z, \tau) \;=\; \mathsf{handle}\ \mathit{resolve}(\Gamma, \mathsf{K}, z, \tau) \\
\quad \mathsf{return}\ x \;\to \mathsf{One}(x) \\
\quad \mathsf{fail}() \quad\;\; \to \mathsf{None} \\
\quad \mathsf{infinite}() \to \mathsf{Amb} \\
\quad \ldots
\end{array}
$$

Here we optimized the infinite operation: instead of resuming we observe that any evaluation with infinite() will eventually have an Amb result, and thus we can directly return with an Amb result without resuming. The main advantage of our new representation is in the definition of forallq as we can now cut the evaluation short as soon as we encounter any Amb result without needing to explore all possible derivations:

$$
\begin{array}{l}
\mathsf{forallq}(\Gamma, z) \to \\
\quad \mathsf{let}\ \mathit{find}(\mathit{current}, \mathit{candidates}) \;=\; \mathsf{match}\ \mathit{candidates} \\
\qquad \mathsf{Nil} \to \mathit{current} \\
\qquad \mathsf{Cons}(m/z : \sigma,\ \mathit{rest}) \to \mathsf{match}\ \mathsf{resume}(m/z, \sigma) \\
\qquad\quad \mathsf{Amb} \quad\;\; \to \mathsf{Amb} \\
\qquad\quad \mathsf{None} \quad\;\; \to \mathit{find}(\mathit{current}, \mathit{rest}) \\
\qquad\quad \mathsf{One}(x) \to \mathsf{match}\ \mathit{current} \\
\qquad\qquad \mathsf{One}(y) \to \mathsf{Amb} \\
\qquad\qquad \mathsf{None} \quad\;\; \to \mathit{find}(\mathsf{One}(x), \mathit{rest}) \\
\quad \mathsf{in}\ \mathit{find}(\mathsf{None},\ \forall m\sigma.\ m/z : \sigma \in \Gamma)
\end{array}
$$

The $\mathit{find}(\mathit{current}, \mathit{candidates})$ function iterates through all candidates using the *current* result which is either None or $\mathsf{One}((Q, e))$. As soon as we find an Amb result, we can immediately return with Amb without trying more solutions. Similarly when already have a valid current solution, and we find another valid solution, we can immediately return with Amb as well.

In our experience, cutting the exploration short as soon an ambigious or infinite derivation is found is quite important in practice. In particular, the [App-var] rule often requires backtracking to get enough type information from the first argument expressions – failing fast is important here. As shown in Section 2.9, we can also prefer inner scope definitions instead of requiring unique solutions. This can actually be used to optimize the exploration even further. In particular, we can directly skip trying resolving any m/z from an outer scope as soon as we found a solution for a definition in an inner scope.

4 SYNTACTIC IMPLICITS AND STATIC OVERLOADING IN PRACTICE

The system described in this paper is fully implemented in the Koka language [Leijen 2019 2021], and it is quite heavily used in the standard library for functions like `show`, `(==)`, `map`, comparison, etc. In Koka, application is not curried and all arguments are in between parenthesis. In contrast to our calculus, the implicit arguments always come last which fits better with the uncurried syntax. In particular, the programmer can now pass implicit parameters explicitly by adding them as regular parameters. For example, we can write `show([1], fn(x) "*")` to display a list of integers where all elements are shown as $*$. In the case of multiple implicit parameters, we can pass a subset of them by name as well, `show((1,True), ?snd/show=fn(x) "*")` for example.

As described in Section 2.9, Koka prefers inner-scope definitions, and as discussed in Section 3.3, instead of cutting the search off at an arbitrary depth our implementation maintains a history where the size of the types of implicit parameters must decrease within 4 recursive elaborations. This makes it possible to still accept the 3-state busy beaver program (see Appendix A of the supplement).

A nice advantage of our system is that there is no need to declare overloaded functions in advance, we just have regular function definitions. This also puts no constraints on the type of these functions. This allows us in particular to overload tuple selectors for example:

```
fun tuple/fst ( x : (a,b)   ) : a
fun triple/fst( x : (a,b,c) ) : a
```

and more generally selectors for fields in any datatype. This is not possible with type classes [Odersky et al. 1995; P. Wadler and Blott 1989] as a specific type signature needs to be declared upfront.

4.1 Default Implementations

In the standard library, we have various definitions to compare basic types, like:

```
type order = Lt | Eq | Gt
fun bool/cmp( x : bool, y : bool ) : order
fun int/cmp( x : int, y : int ) : order
fun list/cmp( xs : list<a>, ys : list<a>, ?cmp : a -> order ) : order
```

Given these definitions, we can now write a generic equality operator:

```
fun default/(==)( x : a, y : a, ?cmp : (a,a) -> order ) : bool
  match cmp(x,y)
    Eq -> True
    _  -> False
fun (!=)( x : a, y : a, ?eq : (a,a) -> bool ) : bool
  !eq(x,y)
```

This way, for any new data type, we only need to define the comparison function `cmp`, and get (in)equality for free. However, there is a problem with this as well. In particular, we may have a new data type that can determine equality more efficiently without doing a full comparison. This already happens in the standard library where equality between integers has a fast primitive:

```
fun int/(==)( x : int, y : int ) : bool
  prim-int-eq(x,y)
```

Unfortunately, since these definitions overlap, an expression like `1==2` will always be rejected as both `int/(==)` and `default/(==)` could apply. Haskell type classes address this problem by allowing default implementations in class definitions that are used unless an instance provides an explicit implementation. In our case, one way around this is to name `default/(==)` differently, and at each new data type require the user to write an explicit `(==)` definition that can either use the default one based on `cmp` or be specialized for that data type.

In Koka we use a different approach though: we pretend that any definition in the `default/` namespace is defined in a scope outside the global module scope, i.e. at scope depth -1 instead of 0. This means that when there is a choice between a regular definition and a default one, the regular one is preferred due to the scope disambiguation (Section 2.9). Our `1==2` example is now unambiguous, and elaborated to `int/(==)(1,2)`.

Unfortunately, the addition of `default/` can cause definition instability (Section 2.11). We may have a function that uses a variable that is resolved to a `default/` definition. If we add a more specific version, that one is now considered to be in an inner scope relative to `default/`, and is preferred without an ambiguity error. This is not ideal and we would like extend the compiler to at least warn in such cases (while preserving the convenience of the outer `default/` scope).

4.2 Grouping Operations

Oftentimes, we need multiple operations that logically belong together. It can be cumbersome to pass each one individually as an implicit parameter in general. Haskell type classes naturally group operations together where for example the `Num` class has addition, multiplication, etc. No such thing exists in our system but instead we can pass structures directly as an implicit parameter.

For example, we can define:

```
struct num<a>
  (+): (a, a) -> a
  (*): (a, a) -> a
```

and provide some "instances" as:

```
val int/num : num<int> = Num( (+), (*) )
```

However, using such structures can be a bit cumbersome, for example:

```
fun fadd( x : a, y : a, z : a, ?num : num<a> ) : a
  match num
    Num((+),(*)) -> x + (y * z)
```

Fortunately, Koka provides dot notation to automatically unpack structures, and we can write:

```
fun fadd( x : a, y : a, z : a, .?num : num<a> ) : a
  x + (y * z)
```

which is essentially syntactic sugar for the previous explicit version. The dot notation also recursively unpacks any `base` members which allows for linear hierarchies, for example for `monad<m>` and `monadplus<m>`. This simple mechanism can work surprisingly well, but of course it is limited as well compared to more specialized mechanisms like type classes. We hope to gain more experience in practice to evaluate its limits.

4.3 Phantom Implicits

The value passed at the call site for a syntactic implicit parameters, is just the plain name of the parameter. If the name is not in scope at the call site the program is rejected. However, we can imagine having a special set of implicit parameter names (and associated types) for which the compiler can supply a more elaborate argument term. We call such special names *phantom implicits*. For example, our Koka implementation defines the special implicit parameter names `kk-line :int` and `kk-file :string` (in `std/core/debug`). If in [INST-IMPLICIT] these names cannot be resolved, the compiler can instead elaborate to specific value: in these cases the current source file line number and name respectively. For example:

```
fun assert-line( condition : bool, msg : string, ?kk-line : int ) : exn ()
  if condition then () else throw("at " ++ kk-line.show ++ ": " ++ msg)
```

and then use it as `assert-line(False,"failed")`. Since `kk-line` is not defined at the call site, the compiler instead provides the current source line number for the implicit parameter, and elaborate to `assert-line(False,"failed",42)`. Note that we can still define `kk-line` explicitly. Unlike a preprocessor macro, we can *abstract* over these special names. For example, we can define:

```
fun assert-fline(condition : bool, msg : string, ?kk-line : int, ?kk-file : string ) : exn ()
  assert-line(condition, kk-file ++ ": " ++ msg )
```

Here, `assert-line` is elaborated to `assert-line(condition, kk-file ++ ": " ++ msg, kk-line)`. Since `kk-line` is in scope the compiler does not need to provide another value. However, when `assert-fline` is used where `kk-line` and `kk-file` are not in scope, the current line number and file of the call site are provided implicitly again.

For both the `kk-line` and `kk-file` phantom implicits, the compiler can supply their argument value just based on their name. However, we can also imagine supplying the argument value of a phantom implicit based on the caller's type context. This would lead to a design that is more similar to how type classes get resolved: we can imagine letting the user define type classes and instances – when a "type class" phantom implicit needs to be resolved, for example $num : num\langle int \rangle$, the instance declarations can be used to construct the correct dictionary to pass.

4.3.1 Divergence. Koka has at the moment a single phantom implicit that is resolved based on its type, namely `hdiv : hdiv<h,a,e>`. The type `hdiv` is an abstract type and users can never create

a value of this type. Only the compiler is allowed to do so if the type contraints are satisfied. In particular, the type `hdiv<h,a,e>` signifies that if the type `a` can contain a reference to the heap `h`, then the effect `e` must contain the divergence effect (`div`). This is used in Koka to correctly infer potential divergence where one stores self referential functions in a mutable heap. Leijen [2014] gives the following example of Landin's knot:

```
fun landin() : div ()            // div is inferred!
  val r = ref(fn() ())           // initialize with a parameter-less function that returns unit
  fun self()
    (!r)()                       // self calls the function stored in r
  r := self                      // store self in r now
  self()                         // and call it
```

Here we create a reference `r` in the global heap, and later store function `self` in there, where `self` calls the function stored in `r`. Even though the function has no syntactic recursion, it diverges and Koka needs to infer the `div` effect in the type of `landin`. In order to detect such sneaky divergence, we define the dereference operation `(!)` with the `hdiv` phantom implicit:

```
fun ref/(!)( r : ref<h,a>,  ?hdiv : hdiv<h,a,e>) : <read<h>|e> a
```

This reads a value of type `a` from a reference `r` in heap `h`, and has a `read<h>` effect and possibly more effects `e`. By including the phantom implicit here, we make explicit that the `e` effect now must contain the divergence effect `div` as well whenever the value type `a` can contain the same heap `h` in its type, i.e. whenever it is self referential. In the `landin` example above, the name `hdiv` for the phantom implicit is not bound so the compiler tries to supply a value instead, with the type `hdiv<h,() -> <read<h>|e> (), e>`. Since `h` is clearly in the value type `() -> <read<h>|e>`, the compiler unifies `e` with `<div|_>` to satisfy the constraint on `hdiv` implicits, and then supplies an internal dummy value as the evidence for the `hdiv` phantom implicit. This ensures that the `landin` function indeed gets the `div` effect.

In most cases in practice, the value type is a simple type like `int` and it is clear that it does not contain `h`. In such cases, the compiler directly supplies an internal dummy value again as evidence, but now without unifying `e` with the divergence effect. Sometimes though, the value type is polymorphic and at the call site it may still be undetermined if `h` can be in `a`. Just like with the previous `kk-line` example, we can then further abstract over the `hdiv` implicit to delay its evaluation – the user may not be able to create `hdiv` evidence values, but they are able to pass them around:

```
fun read2( r : ref<h,a>, ?hdiv : hdiv<h,a,e> ) : <read<h>|e> (a,a)
  val x = !r in (x,x)
```

In this example, the polymorphic dereference `!r` is elaborated to `ref/(!)(r,hdiv)`, where we essentially defer resolving the `hdiv` evidence to the call site of `read2` instead.

5 RELATED WORK

Since the main idea of syntactic implicit parameters is so straightforward, it is not easy to directly compare against more sophisticated systems like type classes or Scala implicits. Instead, we try to highlight how particular usage scenarios are addressed in various systems.

Type classes [P. Wadler and Blott 1989] are a very elegant form of implicits that has seen widespread success in languages like Haskell and Lean for example. An important difference with our system is that type class constraints are automatically generalized if they cannot be resolved locally, essentially introducing an implicit parameter for the dictionary at runtime. For example, $parens = \lambda x.\ "(" +\!\!+\ show\ x +\!\!+\ ")"$ would result in the type $parens : Show\ \alpha \Rightarrow \alpha \to String$ in Haskell. In contrast, in our system this definition is rejected, and we need to explicitly state that we are abstracting over *show*, as $parens = \lambda ?show.\ \lambda x.\ "(" +\!\!+\ show\ x +\!\!+\ ")"$ (or qualify *show* manually to disambiguate). Kovács [2020] investigate how much to generalize the types of inferred dictionary parameters in dependently typed languages.

Another difference is that type classes require global uniqueness of instances to be coherent. As a consequence, instances live in another scope than regular values. Moreover, once we define say an instance for *Num Int*, we cannot locally override this with another instance to do, say, modular arithmetic; Sozeau and Oury [2008] and Dreyer et al. [2007] investigate how such first-class type classes could look like. Kiselyov and Shan [2004] show how to use phantom types and rank-2 polymorphism with type classes to add implicit configuration modularly in a program and obtain coherence without global uniqueness.

Scala [Odersky 2016; Odersky et al. 2017] is one of the first languages to use implicits extensively. Like type classes, the implicits have a separate scope, and are resolved by their type. For example:

```
implicit val number : Int = 1
def add(x : Int)(implicit y : Int) = x + y
add(2)
```

where the `add(2)` would be elaborated to `add(2,number)` as that is the only `Int` in the implicit scope. Scala users usually declare new nominal types to avoid ambiguities. B. C. Oliveira et al. [2010] show how Scala implicits can be used to express type classes as well. However, just like in our system, one still needs to explicitly declare the required implicit parameters. Both type classes and Scala style implicits resolve implicits by their type. B. C. d. S. Oliveira et al. [2012] and later Schrijvers et al. [2019] develop an implicit calculus that expresses the essence of these systems. The latter paper in particular studies the coherence and stability properties.

In contrast, a system that uses explicit names to resolve implicit parameters is presented by Lewis et al. [2000] (and later by Jones [1999]). This design of implicit parameters is based on type classes where using an implicit name $?x$ gives rise to an implicit constraint. For example, let $f = ?y + 2$ in (f with $?y = 1$) evaluates to 3. The $?y$ here is an implicit parameter, and, just like type classes, the type of f is generalized to $?y{:}Int \Rightarrow Int$. The with construct is used to resolve such constraints with a particular value. Since names like $?y$ do not live in the usual lexical scope but are instead type level constraints, this may lead to suprises. For example, consider the expression (let $f = ?y + 2$ in $f + (f$ with $?y = 1))$ with $?y = 2$ which evaluates to 7 (and not 8) even though lexically it may appear as if the outer binding for $?y$ binds the occurrence in f.

White et al. [2015] describe an extension to the OCaml language for ad-hoc polymorphism inspired by Scala implicits and modular type classes. Their modular implicits are based on type-directed implicit module parameters, and elaborate straightforwardly into OCaml's first-class functors. Devriese and Piessens [2011] present *instance arguments* for the Agda language. These are inspired by both Scala's implicits and Agda's existing implicit arguments.

Because we resolve implicits by name, we can use history (Section 3.3) to carefully avoid non-termination of the search. In type based systems this can be more challenging. Agda's search avoids any recursion in the first place at the cost of needing to be more explicit. Rocq [2025 Rocq Prover 2025], has a configurable recursion limit (`Typeclasses Depth`), which defaults to infinite depth-first search. Schrijvers et al. [2019], provides a termination property based on the type head (implicit constraints), which ensures that the size of the type is decreasing. White et al. [2015] provides a termination property similar to ours, where successive instantiations of the same implicit functor must be decreasing in the size of collected constraints.

6 CONCLUSION

We presented a formal system for syntactic implicit parameters and static overloading. Even though each feature is rather straightforward, their interaction turned out to be surprisingly expressive. With the implementation in Koka we hope to gain more experience in using this at scale and extend it in interesting ways. In particular, we would like to improve on grouping (Section 4.2), and study the application of phantom implicits in future work.

REFERENCES

Devriese, and Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. *ACM SIGPLAN Notices* 46 (9). ACM New York, NY, USA: 143–155.

Dreyer, Harper, Chakravarty, and Keller. 2007. Modular Type Classes. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 63–70.

Jones. 1999. *Exploring the Design Space for Type-Based Implicit Parameterization*. Technical report, Oregon Graduate Institute.

Kiselyov, and Shan. 2004. Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 33–44.

Kovács. 2020. Elaboration with First-Class Implicit Function Types. *Proceedings of the ACM on Programming Languages* 4 (ICFP). ACM New York, NY, USA: 1–29.

Leijen. Sep. 2008. HMF: Simple Type Inference for First-Class Polymorphism. In *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming*. ICFP'08. Victoria, Canada. doi:https://doi.org/10.1145/1411204.1411245.

Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. doi:https://doi.org/10.4204/EPTCS.153.8.

Leijen. 2019. Koka Repository. https://github.com/koka-lang/koka.

Leijen. 2021. The Koka Language. https://koka-lang.github.io.

Leijen, and Ye. Sep. 2024. *Principal Type Inference under a Prefix: A Fresh Look at Static Overloading (TR)*. MSR-TR-2024-34. Microsoft Research.

Leijen, and Ye. Jun. 2025. Principal Type Inference under a Prefix: A Fresh Look at Static Overloading. *Proceedings of ACM Programming Languages (PLDI)* 9 (PLDI). ACM. doi:https://doi.org/10.1145/3729308.

Lewis, Launchbury, Meijer, and Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 108–118. POPL'00. ACM, Boston, MA, USA.

Odersky. Dec. 2016. Implicit Function Types. https://www.scala-lang.org/blog/2016/12/07/implicit-function-types.html. Blog post.

Odersky, Blanvillain, Liu, Biboudis, Miller, and Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proceedings of the ACM on Programming Languages* 2 (POPL). ACM New York, NY, USA: 1–29.

Odersky, Wadler, and Wehr. 1995. A Second Look at Overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 135–146.

Bruno CdS Oliveira, Moors, and Odersky. 2010. Type Classes as Objects and Implicits. *ACM Sigplan Notices* 45 (10). ACM New York, NY, USA: 341–360.

Bruno C.d.S. Oliveira, Schrijvers, Choi, Lee, and Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 35–44. PLDI '12. ACM, Beijing, China.

Peyton Jones, Vytiniotis, Weirich, and Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17 (1): 1–82. doi:https://doi.org/10.1017/S0956796806006034.

Pierce. Feb. 2002. *Types and Programming Languages (TAPL)*. 1st edition. The MIT Press, Cambridge, Massachusetts 02142.

Rado. 1962. On Non-Computable Functions. *Bell System Technical Journal* 41 (3). Wiley Online Library: 877–884.

Rocq Prover. 2025. https://rocq-prover.org/. Accessed: 2025-11-13.

Schrijvers, Oliveira, Wadler, and Marntirosian. 2019. COCHIS: Stable and Coherent Implicits. *Journal of Functional Programming*. {.bib-url}.

Selsam, Ullrich, and Moura. 2020. Tabled Typeclass Resolution. arXiv:cs.PL/2001.04301.

Sozeau, and Oury. 2008. First-Class Type Classes. In *International Conference on Theorem Proving in Higher Order Logics*, 278–293. Springer.

P. Wadler, and Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 60–76. POPL'89. ACM, Austin, Texas, USA. doi:https://doi.org/10.1145/75277.75283.

Philip Wadler. 1985. How to Replace Failure by a List of Successes a Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages. In *Functional Programming Languages and Computer Architecture*, edited by Jean-Pierre Jouannaud, 113–128. Springer Berlin Heidelberg, Berlin, Heidelberg.

White, Bour, and Yallop. 2015. Modular Implicits. In *Proceedings ML Family/OCaml Users and Developers Workshops, Gothenburg, Sweden, September 4-5, 2014*, edited by Oleg Kiselyov and Jacques Garrigue, 198:22–63. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association. doi:https://doi.org/10.4204/EPTCS.198.2.

A TURING ENCODING OF A 3-STATE BUSY BEAVER

This example encodes a 3-state "busy beaver" Turing machine program by Ivars Peterson[1] in the type system. It has the following transition table:

current	state A			state B			state C		
symbol	write	move	next	write	move	next	write	move	next
0	1	R	B	1	L	A	1	L	B
1	1	L	C	1	R	B	1	N	HALT

We encode the symbols, tape, and states using abstract types. The tape is represented as a cons list in the type language using phantom type parameters. The tape is encoded finitely with type `inf` representing an infinite sequence of `sym0`.

```
type sym0
type sym1

// a tape
type cons<a,b>    // tape with head ‘a‘ and tail ‘b‘
type inf
          // all states also match on ‘inf‘ and produce ‘cons<sym0,inf>‘ to simulate infinite tape
type state-a
type state-b
type state-c
```

We encode the configuration as the current state of the finite state machine and the left and right tape using phantom type parameters in the `config` type. The machine can read or write the first symbol on the right tape (the head).

```
// the machine configuration: <state, left tape, right tape>
abstract type config<s,a,c>
  Start
```

We give an explicit type annotation to initialize the phantom types for the starting machine configuration.

```
// starting machine configuration with all zeros
val start : config<state-a,inf,cons<sym0,inf>> = Start
```

Transitions are encoded using functions with implicit `?trans` constraints with the final `halt` transition requiring no further implicits.

```
// halting state
fun halt/trans() : ()
  ()
```

When `state == A`, and `head == 0` we encode the transition to write `1`, shift the tape right and continue in state `B` as follows:

```
// stateA: if head==0 then write 1, shift the tape right, and continue in state B
fun a/b/trans( st : config<state-a,cons<x,l>,cons<sym0,r>>,
               ?trans : (config<state-b,l,cons<x,cons<sym1,r>>>) -> () ) : ()
  ()
```

Similar transitions are encoded for the other states:

[1] https://en.wikipedia.org/wiki/Turing_machine_examples#3-state_Busy_Beaver

```
// stateA: if head==1 then write 1, shift the tape left, and continue in state C
fun a/c/trans( st : config<state-a,l,cons<sym1,cons<x,r>>>,
               ?trans : (config<state-c,cons<sym1,l>,cons<x,r>>) -> () ) : ()
  ()

// stateB: if head==0 then write 1, shift the tape left, and continue in state A
fun b/a/trans( st : config<state-b,l,cons<sym0,cons<x,r>>>,
               ?trans : (config<state-a,cons<sym1,l>,cons<x,r>>) -> () ) : ()
  ()

// stateB: if head==1 then write 1, shift the tape right, and continue in state B
fun b/b/trans( st : config<state-b,cons<x,l>,cons<sym1,r>>,
               ?trans : (config<state-b,l,cons<x,cons<sym1,r>>>) -> () ) : ()
  ()

// stateC: if head==0 then write 1, shift the tape left, and continue in state B
fun c/b/trans( st : config<state-c,l,cons<sym0,cons<x,r>>>,
               ?trans : (config<state-b,cons<sym1,l>,cons<x,r>>) -> () ) : ()
  ()

// stateC: if head==1 then write 1, do not move, and continue in state HALT
fun c/end/trans( st : config<state-c,l,cons<sym1,r>>,
                 ?trans : () -> () ) : ()
  ()
```

Because the tape encoding is finite, at times we must expand the tape by turning the infinite portion into zeros. However, we cannot do so arbitrarily since that can lead to ambiguity. To ensure uniqueness, we only expand the side of the tape that needs expansion in order for a transition to happen. For example the `a/b/trans` shifts from the left tape to the right, so we need a corresponding function that if given `state-a` and right hand tape with head `sym0` expands `inf`. This is done in the `inf/a/b/trans` rule below, which then allows the original `a/b/trans` to proceed.

```
fun inf/a/b/trans( st : config<state-a,inf,cons<sym0,r>>,
                   ?trans : (config<state-a,cons<sym0,inf>,cons<sym0,r>>) -> () ) : ()
  ()
fun inf/a/c/trans( st : config<state-a,l,cons<sym1,inf>>,
                   ?trans : (config<state-a,l,cons<sym1,cons<sym0,inf>>>) -> () ) : ()
  ()
fun inf/b/b/trans( st : config<state-b,inf,cons<sym1,r>>,
                   ?trans : (config<state-b,cons<sym0,inf>,cons<sym1,r>>) -> () ) : ()
  ()
fun inf/b/a/trans( st : config<state-b,l,cons<sym0,inf>>,
                   ?trans : (config<state-b,l,cons<sym0,cons<sym0,inf>>>) -> () ) : ()
  ()
fun inf/c/b/trans( st : config<state-c,l,cons<sym0,inf>>,
                   ?trans : (config<state-c,l,cons<sym0,cons<sym0,inf>>>) -> () ) : ()
  ()
```

We can evaluate the program by using the following `main` function and the command: `koka -e busy-beaver.kk`.

```
fun main()
  trans( start )
  println("done.")
```

In VSCode we can hover over the call which shows us the inferred implicit chain representing the transitions. We show the resulting chain below where we have aligned the `inf` and corresponding non-`inf` rules for readability.

```
// expands to a 14 state expansion (with the Koka termination check disabled),
// and matches the execution of:
// <https://en.wikipedia.org/wiki/Turing_machine_examples#3-state_Busy_Beaver>
/*
inf/a/b/trans(?trans=a/b/trans(_,
  b/a/trans(_,
  inf/a/c/trans(_,a/c/trans(_,
  inf/c/b/trans(_,c/b/trans(_,
  inf/b/a/trans(_,b/a/trans(_,
  a/b/trans(_,
  b/b/trans(_,
  b/b/trans(_,
  b/b/trans(_,
  inf/b/b/trans(_,b/b/trans(_,
  b/a/trans(_,
  a/c/trans(_,
  c/end/trans(_,
  halt/trans))))))))))))))))
*/
```

To simplify the handling of `inf`inite expansion, we can recognize that only a single function for shifting left / shifting right is needed. Since infinite tapes are always expanded to add a `sym0` we need to know that there exists a rule that given the current head, state, and a `sym0` where there currently is `inf`, will shift that `sym0` off of that side. However, this requires us to not only know the shape of the head of the transition rule, but also the tail.

With higher-order implicit parameters (parameters whose types are contain implicit parameters), we can expect an implicit parameter `?trans` that shifts to the right (matching on the shape of both the head and tail of the transition), and also require the same implicit matching only the head of the transition rule, requiring further implicits to be handled by the caller.

```
// Expand the ‘inf‘ tape if the state would transition from the infinite side if it had a sym0.
fun inf/l/trans( st : config<s,inf,cons<h,r>>,
                 ?a/trans : (config<s,cons<sym0,inf>,cons<h,r>>,
                             ?trans: (config<s1,inf,r2>) -> ()) -> (),
                 ?b/trans: (config<s,cons<sym0,inf>,cons<h,r>>) -> ()) : ()
  ()
// Expand the ‘inf‘ tape if the state would transition from the infinite side if it had a sym0.
fun inf-r/trans( st : config<s,l,cons<h,inf>>,
                 ?a/trans : (config<s,l,cons<h,cons<sym0,inf>>>,
                             ?trans: (config<s1,l2,cons<h2,inf>>) -> ()) -> (),
                 ?b/trans: (config<s,l,cons<h,cons<sym0,inf>>>) -> () ) : ()
  ()
```

B PROOFS

B.1 Algorithmic Soundness and Completeness

To prove that the algorithm is sound and complete, we first inline the handler in *explore* to make the list of successes explicit without using operations.

$$
\begin{array}{lcl}
\mathit{resolve}(\Gamma, 0, z, \tau) & = & [(\varnothing, \bot)] \\
\mathit{resolve}(\Gamma, n, z, \tau) \mid z:\sigma \in \Gamma & = & [(Q, z\ \overline{e}) \mid (Q, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n-1, \sigma, \tau)] \\
\mathit{resolve}(\Gamma, n, z, \tau) & = & [(Q, m/z\ \overline{e}) \mid m/z:\sigma \in \Gamma,\ (Q, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n-1, \sigma, \tau)]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{inst}(\Gamma, n, \tau_1, \tau_2) & = & [(Q, \cdot) \mid Q \leftarrow \mathit{unify}(\tau_1, \tau_2)] \\
\mathit{inst}(\Gamma, n, \forall\alpha.\sigma, \tau) & = & \mathit{inst}(\Gamma, n, \sigma[\alpha{:=}\mathsf{fresh}()], \tau) \\
\mathit{inst}(\Gamma, n, ?x{:}\tau_1{\to}\rho, \tau_2) & = & \\
\multicolumn{3}{l}{\quad [(Q, e\ \overline{e}) \mid (Q_1, \overline{e}){\leftarrow}\mathit{inst}(\Gamma, n, \rho, \tau_2),\ (Q_2, e){\leftarrow}\mathit{resolve}(\Gamma, n, x, Q_1[\tau_1]),\ Q{\leftarrow}\mathit{compose}(Q_1, Q_2)]}
\end{array}
$$

Then *check* directly calls *resolve*, and does a case analysis on the list of results.

$$
\begin{array}{l}
\mathit{check} : (\Gamma, e, \tau) \to (Q, e) \\
\ldots \\
\mathit{check}(\Gamma, z, \tau) = \mathsf{match}\ \mathit{resolve}(\Gamma, \mathsf{K}, z, \tau) \\
\quad [(Q, e)] \mid \bot \notin e \to (Q, e) \qquad \textit{unique and not } \bot \\
\quad [] \to \mathsf{fail}() \qquad \textit{unresolved} \\
\quad \_ \to \mathsf{fail}() \qquad \textit{ambigious or infinite}
\end{array}
$$

For soundness and completeness, we will need a uniqueness Lemma stating that the rules and algorithm only produce corresponding unique results.

Lemma B.9. (*Uniqueness*)
$\exists!e.\ Q \mid \Gamma \Vdash_n z \stackrel{\leftarrow}{:} \tau \rightsquigarrow e$ iff $\mathit{resolve}(\Gamma, n, z, \tau) = [(Q, e)]$.

If *resolve* were not complete, it could produce one solution instead of two, which could return a value from *check* when there should be none. If it were not sound, it could produce one solution instead of zero, which could return a value from *check* when the implicit is not derivable using the rules. So before proving the uniqueness lemma and soundness and completeness of *check* we first need to prove soundness and completeness of *resolve*.

Because *resolve* and *inst* are mutually recursive, we need to prove their soundness and completeness together by mutual induction. To ensure completeness and termination for *resolve* we can use stepwise induction on n, since n always decreases. However, instantiation does not affect n, and we can simply prove by induction on the derivation given the soundness and completeness of *resolve* at step n, this is well-founded because each inductive step of *inst* makes the type scheme smaller.

For the purposes of these proofs we consider $\Vdash_0$ as $\Vdash_n$ with a premise $n = 0$ and $\Vdash_{n+1}$ as $\Vdash_n$ with premises $n = n' + 1 \ \wedge n > 0 \ \wedge n' = n - 1$ and n replaced by n' within the rule. This allows for cleaner proofs where the conclusions are all in terms of n, and the inductive case can be in terms of $n - 1$, matching the algorithm.

Additionally, we generalize the proofs over all solutions $\mathsf{S} = [(Q, e)]$ (resolve) and $Ss = [(Q, \overline{e})]$ (instantiation) satisfying the inference rules, since the algorithm works via list comprehension over all solutions.

Lemma B.10. (*Resolution Soundness*)
If $\mathit{resolve}(\Gamma, n, z, \tau) = \mathsf{S}$ at step n and *inst* is sound (**1**) at step $n - 1$ where $n > 0$, then also
$\forall (Q, e) \in \mathsf{S}.\ Q \mid \Gamma \Vdash_n z \stackrel{\leftarrow}{:} \tau \rightsquigarrow e$.

Proof. Proceeding by cases on n first with $n = 0$.

Case We have $\mathit{resolve}(\Gamma, 0, z, \tau) = [(\varnothing, \bot)]$ where $n = 0$, and need to show that $(\varnothing, \bot)$ is derivable in the rules. We can derive $(\varnothing, \bot)$ via [VAR-CUT] since $n = 0$.

Case When $n > 0$ (**2**), we have two subcases.

subcase $\mathit{resolve}(\Gamma, n, z, \tau) \mid z : \sigma \in \Gamma$ (**4**) $= [(Q, z\ \overline{e}) \mid (Q, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n-1, \sigma, \tau)]$ (**3**). By soundness of *inst* (1), we have for each element in (3), $Q \mid \Gamma \vdash_{n-1} \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ (**5**). And by [VAR-DIRECTK] and (2,5), for each $(Q,\ z\ \overline{e})$ we have $Q \mid \Gamma \Vdash_n z \overset{\leftarrow}{:} \tau \rightsquigarrow z\ \overline{e}$.

subcase $\mathit{resolve}(\Gamma, n, z, \tau) = [(Q, m/z\ \overline{e}) \mid m/z : \sigma \in \Gamma,\ (Q, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n-1, \sigma, \tau)]$. From the algorithm we have $(Q, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n-1, \sigma, \tau)$ (**3**), and $m/z : \sigma \in \Gamma$ (**4**) with $z \notin \Gamma$ (**5**) (implicitly by falling through the match). We can derive $Q \mid \Gamma \vdash_{n-1} \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ (**6**) by soundness of *inst* (1,3). For each $(Q,\ m/z\ \overline{e})$ we can show $Q \mid \Gamma \Vdash_n z \overset{\leftarrow}{:} \tau \rightsquigarrow m/z\ \overline{e}$ by [VAR-QUALIFYK] using (4,5,6).

Lemma B.11. (*Instantiation Soundness*)
If $\mathit{inst}(\Gamma, n, \tau_1, \tau_2) = Ss$ at step n and *resolve* is sound (**1**) at step n then also
$\forall (Q, \overline{e}) \in Ss.\ Q \mid \Gamma \vdash_n \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$.

Proof. We prove by induction on the derivation.

Case The first case applies to mono types: $\mathit{inst}(\Gamma, n, \tau_1, \tau_2) = [(Q, \cdot) \mid Q \leftarrow \mathit{unify}(\tau_1, \tau_2)]$. We derive $Q \vdash \tau_1 \approx \tau_2$ (**2**) by the soundness of *unify* (Lemma C.28). We then can show $Q \mid \Gamma \vdash_n \tau_1 \sqsubseteq \tau_2 \rightsquigarrow \cdot$ for all $(Q,\ \overline{e})$ in the resulting list by [INST-MONO] and (2).

Case This case applies to type schemes with quantifiers: $\mathit{inst}(\Gamma, n, \forall \alpha.\sigma, \tau) = \mathit{inst}(\Gamma, n, \sigma[\alpha{:=}\mathsf{fresh}()], \tau)$. Follows directly the inductive hypothesis (given that fresh is correct).

Case The last case applies to implicits types:

$\mathit{inst}(\Gamma, n, ?x{:}\tau_1 \to \rho, \tau_2) =$
$[(Q, e\ \overline{e}) \mid (Q_1, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n, \rho, \tau_2),\ (Q_2, e) \leftarrow \mathit{resolve}(\Gamma, n, x, Q_1[\tau_1]),\ Q \leftarrow \mathit{compose}(Q_1, Q_2)]$

From the algorithm we have $(Q_1, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n, \rho, \tau_2)$ (**3**), $(Q_2, e) \leftarrow \mathit{resolve}(\Gamma, n, x, Q_1[\tau_1])$ (**4**), and $Q \leftarrow \mathit{compose}(Q_1, Q_2)$ (**5**). By induction (2,3), we have $Q_1 \mid \Gamma \vdash_n \rho \sqsubseteq \tau_2 \rightsquigarrow \overline{e}$ (**6**). By soundnes of *resolve* (1,4), we have $Q_2 \mid \Gamma \Vdash_n x \overset{\leftarrow}{:} Q_1[\tau_1] \rightsquigarrow e$ (**7**). We can derive from the soundness of *compose* (Lemma C.29) and (5), that we have $\vDash (Q_1, Q_2)$ (**8**) We can apply [INST-IMPLICIT] now by (6,7,8).

We also need completeness of *resolve* and *inst*.

Lemma B.12. (*Resolution Completeness*)
If $Q \mid \Gamma \Vdash_n z \overset{\leftarrow}{:} \tau \rightsquigarrow e$ at step n and *inst* is complete at step $n-1$ where $n \geqslant 1$ then
$\mathit{resolve}(\Gamma, n, z, \tau) = \mathsf{S} \wedge (Q, e) \in \mathsf{S}$.

Proof. We proceed by case analysis of the derivation. Because the rules have mutually exclusive premises we can consider each case with respect to a single line of the algorithm.

Case From [VAR-CUT] we have $\varnothing \mid \Gamma \Vdash_0 z \overset{\leftarrow}{:} \tau \rightsquigarrow \bot$ and $n = 0$ which requires us to include $(\varnothing, \bot)$ in the result when $n = 0 : \mathit{resolve}(\Gamma, 0, z, \tau) = [(\varnothing, \bot)]$. This is mutually exclusive with the other cases where the conclusions require $n > 0$ which contracticts $n = 0$.

Case From [VAR-DIRECTK] we have $z : \sigma \in \Gamma$ (**2**), and $Q \mid \Gamma \vdash_{n-1} \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ (**3**). The corresponding line in the algorithm is mutually exclusive with the other cases since the conclusion only applies for $n > 0$, and ensures (2). From the inductive hypothesis and (3) we have
$\mathit{inst}(\Gamma, n-1, \sigma, \tau) = Ss \wedge (Q, \overline{e}) \in Ss$ (**4**). From (4) we are required to include $(Q, z\ \overline{e})$ in the results in this case:
$\mathit{resolve}(\Gamma, n, z, \tau) \mid z : \sigma \in \Gamma = [(Q, z\ \overline{e}) \mid (Q, \overline{e}) \leftarrow \mathit{inst}(\Gamma, n-1, \sigma, \tau)]$

Case From [VAR-QUALIFYK] we have that $z \notin \mathrm{dom}(\Gamma)$ (**2**) $m/z : \sigma \in \Gamma$ (**3**) and $Q \mid \Gamma \vdash_{n-1} \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ (**4**) with $n > 0$ (**5**). We do not match the first (5) or second (2) cases of the algorithm making this mutually exclusive. From the inductive hypothesis and (4), we have $\mathit{inst}(\Gamma, n-1, \sigma, \tau) = Ss \wedge (Q, \overline{e}) \in Ss$

(**6**). From (6) we must include $(Q, m/z\ \overline{e})$ in the results:
$resolve(\Gamma, n, z, \tau) \ = \ [(Q, m/z\ \overline{e}) \mid m/z : \sigma \in \Gamma,\ (Q, \overline{e}) \leftarrow inst(\Gamma, n-1, \sigma, \tau)]$.

Lemma B.13. (*Instantiation Completeness*)
If $Q \mid \Gamma \vdash_n \ \sigma \sqsubseteq \tau \ \rightsquigarrow \overline{e}$ at step n and *resolve* (**1**) is complete at step n where $n \geqslant 0$ then also $inst(\Gamma, n, \tau_1, \tau_2) = Ss \ \wedge (Q, \overline{e}) \ \in Ss$.

Proof. We proceed by induction on the derivation tree.

Case From $Q \mid \Gamma \vdash_n \ \tau_1 \sqsubseteq \tau_2 \ \rightsquigarrow \cdot$ we have $Q \vdash \ \tau_1 \approx \tau_2$ and by the completeness of unification, we are required to include $(Q, \ .)$ in our result: $inst(\Gamma, n, \tau_1, \tau_2) \ = \ [(Q, \cdot) \mid Q \leftarrow unify(\tau_1, \tau_2)]$.

Case From $Q \mid \Gamma \vdash_n \ \forall \alpha.\sigma \sqsubseteq \tau \ \rightsquigarrow \overline{e}$ we have $Q \mid \Gamma \vdash_n \ \sigma \sqsubseteq \tau \ \rightsquigarrow \overline{e}$ (**3**) and fresh α. By the induction hypothesis and (3) we have that $Q \mid \Gamma \vdash_n \ \sigma \sqsubseteq \tau \ \rightsquigarrow \overline{e}$ requires us to include results from $inst(\Gamma, n, \sigma[\alpha{:=}\mathsf{fresh}()], \tau)$, given fresh type variables.

Case From $Q_1, Q_2 \mid \Gamma \vdash_n ?x{:}\tau_1 \to \rho \sqsubseteq \tau_2 \ \rightsquigarrow e\ \overline{e}$ we have $Q_1 \mid \Gamma \vdash_n \ \rho \sqsubseteq \tau_2 \ \rightsquigarrow \overline{e}$ (**3**) and $Q_2 \mid \Gamma \Vdash_n \ x \overset{\leftarrow}{:} Q_1[\tau_1] \ \rightsquigarrow e$ (**4**). By the induction hypothesis and (3) we have that we must consider each $(Q_1, \overline{e}) \leftarrow inst(\Gamma, n, \rho, \tau_2)$ and for each of those we derive $(Q_2, e) \leftarrow resolve(\Gamma, n, x, Q_1[\tau_1])$ via the induction hypothesis and (4). Finally due to the completeness of *compose* we have that (Q_1, Q_2) exists, requiring us to include $(Q, \ e\ \overline{e})$ in the results for this case:

$inst(\Gamma, n, ?x{:}\tau_1 \to \rho, \tau_2) \ =$
$\quad [(Q, e\ \overline{e}) \mid (Q_1, \overline{e}) \leftarrow inst(\Gamma, n, \rho, \tau_2),\ (Q_2, e) \leftarrow resolve(\Gamma, n, x, Q_1[\tau_1]),\ Q \leftarrow compose(Q_1, Q_2)]$

Finally we need that uniqueness holds for both directions.

Proof. (*Of Lemma B.9 (Uniqueness)*) As a reminder, to prove soundness, we must first prove that unique derivations correspond to unique resolutions in the algorithm:
$\exists! e\ Q \mid \Gamma \Vdash_n \ z \overset{\leftarrow}{:} \tau \ \rightsquigarrow e$ *iff* $resolve(\Gamma, n, z, \tau) = [(Q, e)]$.
Case In the forwards direction we have $Q \mid \Gamma \Vdash_n \ z \overset{\leftarrow}{:} \tau \ \rightsquigarrow e$ (**1**) and $\exists! e$ (**2**). From completeness and (1) we have that $resolve(\Gamma, n, z, \tau) = \mathrm{S}$ where $(Q, e) \ \in \mathrm{S}$. Assume there are multiple (Q, e) in S: that immediately contradicts (2). So we only have one result, and can show $resolve(\Gamma, n, z, \tau) = [(Q, e)]$

Case In the backwards direction we have $resolve(\Gamma, n, z, \tau) = [(Q, e)]$ (**1**). By soundness and (1) we have that $Q \mid \Gamma \Vdash_n \ z \overset{\leftarrow}{:} \tau \ \rightsquigarrow e$ (**2**). Furthermore, assume that there is some other $Q' \mid \Gamma \Vdash_n \ z \overset{\leftarrow}{:} \tau \ \rightsquigarrow e'$, then via completeness and (1) we have a contradiction. Therefore the solution (2) is unique $\exists! e$ (**3**). With (2) and (3) we are able to show the conclusion.

$check \ : \ (\Gamma, e, \tau) \to (Q, e)$
$\ldots$
$check(\Gamma, z, \tau) \ = \ \mathsf{match}\ resolve(\Gamma, \mathsf{K}, z, \tau)$
$\quad [(Q, e)] \ \mid \bot \notin e \to (Q, e)$
$\quad [] \qquad\qquad \to \mathsf{fail}()$ *unresolved*
$\quad \_ \qquad\qquad \to \mathsf{fail}()$ *ambigious or infinite*

Proof. (*Of Theorem 3.7 (Algorithmic Soundness)*) We must prove that given $check(\Gamma, z, \tau) = (Q, e)$, then also
$Q \mid \Gamma \vdash \ z \overset{\leftarrow}{:} \tau \ \rightsquigarrow e$ (**1**) (with [VAR-K]). The only case where *check* produces a result is when *resolve* returns a unique result e with an associated Q (**2**). We can show (1) via the uniqueness lemma and (2).

Proof. (*Of Theorem 3.8 (Algorithmic Completeness)*) We must prove that given $Q \mid \Gamma \vdash z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$ also $\mathit{check}(\Gamma, z, \tau) = (Q, e)$ (**1**). The rule only produces a result if $\Vdash_n$ has a unique result e with an associated Q (**2**). We can show (1) via the uniqueness lemma and (2).

B.2 Soundness of the Finite Type Rules

Proof. (*Of Theorem 3.6*) We must show that given $Q \mid \Gamma \vdash z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$ (**1**) with [VAR-K], then $Q \mid \Gamma \vdash z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$ with [VAR].

This reduces trivially to showing that if $Q \mid \Gamma \Vdash_n z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$ (**1**), $\exists! e$ (**2**), and $\bot \notin e$ (**3**) then $\exists! e,\ Q \mid \Gamma \Vdash z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$ (**4**).

We prove by cases on the number and kinds of solutions from the finite rules $\Vdash_n$.

Case None or multiple solutions: Contradiction that we have a solution (1) and that it is unique (2).

Case One solution:

subcase $\bot \in e$: Contradiction that the solution is not infinite (3)

subcase $\bot \notin e$: Straightforward proof by induction on the derivation, the rules are identical except for passing n. We can proceed by case analysis on the derivation of $\bot \notin e$ in the inductive cases, which allows us to eliminate rule [VAR-CUT] which is the only other difference.

More detailed proof of last subcase.

Lemma B.14.
If $Q \mid \Gamma \Vdash_n z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e \wedge \bot \notin e$ then also $Q \mid \Gamma \Vdash z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$

Lemma B.15.
If $Q \mid \Gamma \vdash_n \sigma \sqsubseteq \tau \rightsquigarrow \overline{e} \wedge \bot \notin \overline{e}$ then also $Q \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$

Proof. (*Of Lemma B.14*) Induction on the structure of the derivation, where we have $Q \mid \Gamma \Vdash_n z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow e$ (**1**) and $\bot \notin e$ (**2**).

Case $Q \mid \Gamma \Vdash_0 z \mathrel{\overleftarrow{:}} \tau \rightsquigarrow \bot$: by contradiction with (2).

Case $Q \mid \Gamma \Vdash_n z : \tau \rightsquigarrow z\ \overline{e}$

subcase $z : \sigma \in \Gamma$ (**3**) We have that only [VAR-DIRECTK] is the only match, and we can derive [VAR-DIRECT] via (3) and the inductive hypothesis.

subcase $z : \sigma \notin \Gamma$ (**3**) We have that only [VAR-QUALIFYK] matches since (3) contradicts with the other subcase and thus $m/z : \sigma \in \Gamma$ (**4**), and we can derive [VAR-QUALIFY] from (3,4) and the inductive hypothesis.

Proof. ((*Of Lemma B.15*)) Induction on the structure of the derivation where we have $Q \mid \Gamma \vdash_n \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ (**1**) and $\bot \notin \overline{e}$ (**2**). Trivial by case analysis on the derivation of $\bot \notin e$ and passing n for the inductive hypotheses.

B.3 Principal Derivations

Here we prove Theorem 2.3: For any derivations $Q_1 \mid \Gamma \vdash_{\text{GEN}} e \mathrel{\overrightarrow{:}} \sigma \rightsquigarrow e_1$ and $Q_2 \mid \Gamma \vdash_{\text{GEN}} e \mathrel{\overrightarrow{:}} \sigma' \rightsquigarrow e_2$, we have $Q_1 = Q_2$, $\sigma = \sigma'$, and $e_1 = e_2$.

Leijen and Ye [2024] already show that type equivalence is principal (Theorem D.28):

Lemma B.16. (*Principal type equivalence*)
If $Q_1 \vdash \tau_1 \approx \tau_2$, then for any other derivation $Q_2 \vdash \tau_1 \approx \tau_2$, we have $Q_1 = Q_2$.

Next we show that variable resolution is principal (and coherent) as well:

Lemma B.17. (*Variable resolution is principal*)
If $Q_1 \mid \Gamma \vdash z \coloneqq \tau \rightsquigarrow e_1$ and also $Q_2 \mid \Gamma \vdash z \coloneqq \tau \rightsquigarrow e_2$, then $Q_1 = Q_2$ and $e_1 = e_2$.

Proof. For the elaboration, the equality $e_1 = e_2$ is by construction due to the $\exists! e$ premise. We still need to show though that we also have $Q_1 = Q_2$. Suppose we have $Q_1 \mid \Gamma \Vdash z \coloneqq \tau \rightsquigarrow e$ and also $Q_2 \mid \Gamma \Vdash z \coloneqq \tau \rightsquigarrow e$. We proceed by induction over the variable rules and instantiation:

Case [VAR-DIRECT]: by induction over the premise, for $Q_1 \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ and $Q_2 \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}$ we have $Q_1 = Q_2$.

Case [VAR-QUALIFY]: there is a choice here which $m/z : \sigma$ to pick from Γ. However, since they rewrite to the same $m/z\ \overline{e}$ both derivations use the same $m/z : \sigma$ and thus by induction over the instantiation premises, we have $Q_1 = Q_2$.

Case [INST-MONO]: We have $Q_1 \vdash \tau_1 \approx \tau_2$ and $Q_2 \vdash \tau_1 \approx \tau_2$, and by Lemma B.16, we have $Q_1 = Q_2$.

Case [INST-IMPLICIT]: We have $Q_{11} \mid \Gamma \vdash \rho \sqsubseteq \tau_2 \rightsquigarrow \overline{e}$ and $Q_{21} \mid \Gamma \vdash \rho \sqsubseteq \tau_2 \rightsquigarrow \overline{e}$ and by induction $Q_{11} = Q_{21}$ (**1**). We also have $Q_{12} \mid \Gamma \Vdash x \coloneqq Q_{11}[\tau] \rightsquigarrow e_1$ and $Q_{22} \mid \Gamma \Vdash x \coloneqq Q_{21}[\tau] \rightsquigarrow e_2$. However from (1) we have $Q_{11}[\tau] = Q_{21}[\tau]$, and thus by induction $e_1 = e_2$ and $Q_{12} = Q_{22}$, and thus $Q_{11}, Q_{12} = Q_{21}, Q_{22}$.

Case [INST-QUANTIFY]: From the premise we directly have by induction $Q_1 = Q_2$.

□

With Lemma B.17 we can now follow the proof of Leijen and Ye [2024] (Appendix C) to show that the full rules are principal:

Proof. (*Of Theorem 2.3*) Since the rules in Figure 5 are syntax directed (with preference for [APP-VAR] over [APP-ARG]), the structure of the two derivations must match exactly. Furthermore, by Lemma B.16 and Lemma B.17, the premise $Q \vdash \tau_1 \approx \tau_2$ (in [ANN]) and the [VAR] rule are also principal. Therefore both derivations are exactly equal with no derivation choices. In particular, in a [VAR] leaf where $Q \mid \Gamma \vdash z \coloneqq \tau \rightsquigarrow e'$ we always have the same τ for any derivation over $e[z]$, and it always elaborates to the same e'. □

B.4 Stability

First we show monomorphic substitution, then monomorphic stability, and finally polymorphic stability.

B.4.1 Monomorphic Substitution. Note: we first want to show other derivations exist and only later reason about unique derivations, so the lemmas are stated over $\Vdash$ for now. Moreover, since we need to reason about both the instance and variable rules together, we generally need to establish a stronger extended lemma first to have strong enough invariants.

The following Lemma is the inverse of stability (Lemma B.21) which we need in order to show that derivations are still unique in the stability theorem (Theorem 2.4).

Lemma B.18. (*Extended Monomorphic Substitution*)
If $Q_1 \mid \theta\Gamma \Vdash z \coloneqq \theta\tau \rightsquigarrow e_1$, then for any $\tau' \sqsubseteq \tau$ (**I**) where $\tau = \theta'\tau'$ such that $\theta' \sqsubseteq (Q_1, \theta)$ (**II**), we also have $Q_2 \mid \Gamma \Vdash z \coloneqq \tau' \rightsquigarrow e_2$ with $e_1 = e_2$ and $Q_2 \sqsubseteq (\theta, Q_1)$.

Lemma B.19. (*Monomorphic Substitution*)
If $Q_1 \mid \theta\Gamma \Vdash z \coloneqq \theta\tau \rightsquigarrow e_1$, then we also have $Q_2 \mid \Gamma \Vdash z \coloneqq \tau \rightsquigarrow e_2$ with $e_1 = e_2$ and $Q_2 \sqsubseteq (\theta, Q_1)$.

Proof. This follows directly from Lemma B.18 where $\tau' = \tau$. □

Note that if $Q_1 \mid \theta\Gamma \vdash z \mathrel{\overset{\leftarrow}{:}} \theta\tau \rightsquigarrow e$, then we *cannot* conclude $Q_2 \mid \Gamma \vdash z \mathrel{\overset{\leftarrow}{:}} \tau \rightsquigarrow e$ also holds.

We have by the premise $\exists!e.\ Q_1 \mid \theta\Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \theta\tau \rightsquigarrow e$. By Lemma B.19, for all such derivations, we also have $Q_2 \mid \Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \tau \rightsquigarrow e$ with the same e (satisfying the $\exists e$ part). However, we also need to show that there are no *other* derivations $Q_2 \mid \Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \tau \rightsquigarrow e'$ with $e' \neq e$ (satisfying the unique part). Unfortunately, that is not generally the case. Consider the example from Section 2.10 where we have that $\varnothing \mid [\alpha{:=}c]\Gamma \vdash \mathit{trans} \mathrel{\overset{\leftarrow}{:}} [\alpha{:=}c](\alpha{\to}\alpha) \rightsquigarrow \mathit{generic}/\mathit{trans}$ holds, but $\_ \mid \Gamma \not\Vdash \mathit{trans} \mathrel{\overset{\leftarrow}{:}} \alpha{\to}\alpha$ does not as it is ambiguous.

Proof. (*Of Lemma B.18*) We proceed by induction over the variable and instance rules. For the instance rules, if $Q_1 \mid \theta\Gamma \vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_1$, then also $Q_2 \mid \Gamma \vdash \sigma \sqsubseteq \tau' \rightsquigarrow \overline{es}_2$ with $e_1 = e_2$ and $Q_2 \sqsubseteq (\theta, Q_1)$.

Case [VAR-DIRECT]: By the premise, $z : \theta\sigma \in \theta\Gamma$ (**1a**) and $Q_1 \mid \theta\Gamma \Vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow e_1$ (**1b**). From (1a), we also have $z : \sigma \in \Gamma$. By induction on (1b), we have $Q_2 \mid \Gamma \Vdash \sigma \sqsubseteq \tau' \rightsquigarrow e_2$ with $e_1 = e_2$, and we can conclude $Q_2 \mid \Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \tau' \rightsquigarrow e_2$.

Case [VAR-QUALIFY]: By the premise, $m/z : \theta\sigma \in \theta\Gamma$ (**1a**) and $Q_1 \mid \theta\Gamma \Vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow e_1$ (**1b**). From (1a), we also have $m/z : \sigma \in \Gamma$. By induction on (1b), we have $Q_2 \mid \Gamma \Vdash \sigma \sqsubseteq \tau' \rightsquigarrow e_2$ with $e_1 = e_2$, and we can conclude $Q_2 \mid \Gamma \Vdash m/z \mathrel{\overset{\leftarrow}{:}} \tau' \rightsquigarrow e_2$.

Case [INST-IMPLICIT]: We have $Q_{11} \mid \theta\Gamma \vdash \theta\rho \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_1$ (**1a**) and $Q_{12} \mid \theta\Gamma \Vdash x \mathrel{\overset{\leftarrow}{:}} Q_{11}[\theta\tau_1] \rightsquigarrow e_1$ (**1b**). By induction on (1a), $Q_{21} \mid \Gamma \vdash \rho \sqsubseteq \tau' \rightsquigarrow \overline{es}_2$ (**2a**) with $\overline{es}_1 = \overline{es}_2$ (**2b**) and $Q_{21} \sqsubseteq (Q_{11}, \theta)$ (**2c**).

By Lemma B.23, $Q_{11} \circ \theta = (Q_{11}, \theta)$, and by composition, $= \theta \circ (Q_{11}, \theta)$. From (1b) we therefore have $Q_{12} \mid \theta\Gamma \vdash x \mathrel{\overset{\leftarrow}{:}} \theta((Q_{11}, \theta)[\tau_1]) \rightsquigarrow e_1$. By (2c), we also have $(\tau' =) Q_{21}[\tau_1] \sqsubseteq (Q_{11}, \theta)[\tau_1] (= \tau)$ (I) (and thus $(Q_{11}, \theta)[\tau_1] = (Q_{11}, \theta)[Q_{21}[\tau_1]]$ where $(Q_{11}, \theta) \sqsubseteq (Q_{11}, \theta)$ (II)). We can now use induction on (1b) to conclude $Q_{22} \mid \Gamma \vdash x \mathrel{\overset{\leftarrow}{:}} Q_{21}[\tau_1] \rightsquigarrow e_2$ (**3a**), with $e_1 = e_2$ (**3b**) and $Q_{22} \sqsubseteq (\theta, Q_{12})$ (**3c**). We can now derive from (2a,3a), $(Q_{21}, Q_{22}) \mid \Gamma \vdash ?x{:}\tau_1{\to}\rho \sqsubseteq \tau' \rightsquigarrow e_2\ \overline{es}_2$ with $e_1\ \overline{es}_1 = e_2\ \overline{es}_2$ (2b,3b), and $(Q_{21}, Q_{22}) \sqsubseteq ((Q_{11}, Q_{12}), \theta)$ (2c,3c).

Case [INST-MONO]: We have $Q_1 \vdash \theta\tau_1 \approx \theta\tau$ from the premise, and thus by Theorem B.25, $Q \vdash \tau_1 \approx \tau$ (**1a**) with $(Q, \theta) = (Q_1, \theta)$ (**1b**) and also $Q \sqsubseteq (Q_1, \theta)$ (**1c**). We can now derive:

$$
\begin{array}{lll}
 & (Q_1, \theta)[\tau_1] & \\
= & (Q_1, \theta)[Q[\tau_1]] & \{\ (1c)\ \} \\
= & (Q_1, \theta)[Q[\tau]] & \{\ (1a), \textit{Theorem 2.1}\ \} \\
= & (Q_1, \theta)[\tau] & \{\ (1c)\ \} \\
= & (Q_1, \theta)[\theta'\tau'] & \{\ (I)\ \} \\
= & (Q_1, \theta)[\tau'] & \{\ (II)\ \}
\end{array}
$$

And by Theorem 2.2, $Q_2 \vdash \tau_1 \approx \tau'$ with $Q_2 \sqsubseteq (Q_1, \theta)$. □

Case [INST-QUANTIFY]: We have $Q_1 \mid \theta\Gamma \vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_1$. By induction, $Q_2 \mid \Gamma \vDash \sigma \sqsubseteq \tau' \rightsquigarrow \overline{es}_2$ with $\overline{es}_1 = \overline{es}_2$ and $Q_2 \sqsubseteq (Q_1, \theta)$. With the same fresh α, we can conclude $Q_2 \mid \Gamma \vDash \forall\alpha.\sigma \sqsubseteq \tau' \rightsquigarrow \overline{es}_2$.

B.4.2 Monomorphic Stability.

To prove stability, we again first establish an extended lemma:

Lemma B.20. (*Extended Monomorphic Stability*)
If $Q_1 \mid \Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \tau' \rightsquigarrow e_1$, and $\vDash (\theta, Q_1)$ (**I**), then for any $\tau' \sqsubseteq \tau$ (**II**) where $\tau = \theta'\tau'$ such that $\theta' \sqsubseteq (Q_1, \theta)$ (**III**), we have $Q_2 \mid \theta\Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \theta\tau \rightsquigarrow e_2$ with $Q_1 \sqsubseteq (\theta, Q_2)$.

Lemma B.21. (*Monomorphic Stability over Derivations*)
If $Q_1 \mid \Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \tau \rightsquigarrow e$ with $\vDash (\theta, Q_1)$, then we also have $Q_2 \mid \theta\Gamma \Vdash z \mathrel{\overset{\leftarrow}{:}} \theta\tau \rightsquigarrow e$ with $Q_1 \sqsubseteq (\theta, Q_2)$.

Proof. This follows directly from Lemma B.20 with $\tau' = \tau$. □

Proof. (*Of Lemma B.20*) We proceed by induction over the variable and instance rules. For the instance rules, if $Q_1 \mid \theta\Gamma \vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_1$, then also $Q_2 \mid \Gamma \vdash \sigma \sqsubseteq \tau' \rightsquigarrow \overline{es}_2$ with $e_1 = e_2$ and $Q_2 \sqsubseteq (\theta, Q_1)$.

Case [VAR-DIRECT]: By the premise, $z : \sigma \in \Gamma$ (**1a**) and $Q_1 \mid \Gamma \Vdash \sigma \sqsubseteq \tau' \rightsquigarrow e_1$ (**1b**) with $\vDash (Q_1, \theta)$ (**1c**). From (1a), we also have $z : \theta\sigma \in \theta\Gamma$. By induction on (1b,1c), we have $Q_2 \mid \theta\Gamma \Vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow e_2$ with $e_1 = e_2$, and we can conclude $Q_2 \mid \theta\Gamma \Vdash z \overset{\leftarrow}{:} \theta\tau \rightsquigarrow e_2$.

Case [VAR-QUALIFY]: By the premise, $m/z : \sigma \in \Gamma$ (**1a**) and $Q_1 \mid \Gamma \Vdash \sigma \sqsubseteq \tau' \rightsquigarrow e_1$ (**1b**) with $\vDash (Q_1, \theta)$ (**1c**). From (1a), we also have $m/z : \theta\sigma \in \theta\Gamma$. By induction on (1b,1c), we have $Q_2 \mid \theta\Gamma \Vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow e_2$ with $e_1 = e_2$, and we can conclude $Q_2 \mid \theta\Gamma \Vdash m/z \overset{\leftarrow}{:} \theta\tau' \rightsquigarrow e_2$.

Case [INST-IMPLICIT]: We have $Q_{11} \mid \Gamma \vdash \rho \sqsubseteq \tau' \rightsquigarrow \overline{es}_1$ (**1a**), $Q_{12} \mid \Gamma \Vdash x \overset{\leftarrow}{:} Q_{11}[\tau_1] \rightsquigarrow e_1$ (**1b**), and $\vDash (Q_{11}, Q_{12}, \theta)$ (**1c**). By induction on (1a,1c), $Q_{21} \mid \theta\Gamma \vdash \theta\rho \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_2$ (**2a**) with $\overline{es}_1 = \overline{es}_2$ (**2b**) and $Q_{11} \sqsubseteq (Q_{21}, \theta)$ (**2c**).

By (2c), we also have $(\tau' =) Q_{11}[\tau_1] \sqsubseteq (Q_{21}, \theta)[\tau_1] (= \tau)$ (II), (and thus $(Q_{21}, \theta)[\tau_1] = (Q_{21}, \theta)[Q_{11}[\tau_1]]$ where $(Q_{21}, \theta) \sqsubseteq (Q_{21}, \theta)$ (III)).

By induction on (1b,1c) we now have $Q_{22} \mid \theta\Gamma \vdash x \overset{\leftarrow}{:} \theta((Q_{21}, \theta)[\tau_1]) \rightsquigarrow e_2$ (**3a**), with $e_1 = e_2$ (**3b**) and $Q_{12} \sqsubseteq (\theta, Q_{22})$ (**3c**). By composition, $\theta \circ (Q_{21}, \theta) = (Q_{21}, \theta)$, and by Lemma B.23, $(Q_{21}, \theta) = Q_{21} \circ \theta$, and thus from (3a), $Q_{22} \mid \theta\Gamma \vdash x \overset{\leftarrow}{:} Q_{21}[\theta\tau_1] \rightsquigarrow e_2$ (**3d**).

We can now derive from (2a,3d), $(Q_{21}, Q_{22}) \mid \theta\Gamma \vdash \theta(?x{:}\tau_1 \to \rho) \sqsubseteq \theta\tau \rightsquigarrow e_2\ \overline{es}_2$ with $e_1\ \overline{es}_1 = e_2\ \overline{es}_2$ (2b,3b), and $(Q_{11}, Q_{12}) \sqsubseteq ((Q_{21}, Q_{22}), \theta)$ (2c,3c).

Case [INST-MONO]: We have $Q_1 \vdash \tau_1 \approx \tau'$ (**1a**) from the premise with $\vDash (Q_1, \theta)$. We can now derive:

$$
\begin{array}{lll}
 & (Q_1, \theta)[\theta\tau_1] & \\
= & (Q_1, \theta)[\tau_1] & \{\ \theta \sqsubseteq (Q_1, \theta)\ \} \\
= & (Q_1, \theta)[Q_1[\tau_1]] & \{\ Q_1 \sqsubseteq (Q_1, \theta)\ \} \\
= & (Q_1, \theta)[Q_1[\tau']] & \{\ (1a), \textit{Theorem 2.1}\ \} \\
= & (Q_1, \theta)[\tau'] & \{\ \} \\
= & (Q_1, \theta)[\theta'\tau'] & \{\ (III)\ \} \\
= & (Q_1, \theta)[\tau] & \{\ def.\ \} \\
= & (Q_1, \theta)[\theta\tau] & \{\ \theta \sqsubseteq (Q_1, \theta)\ \}
\end{array}
$$

And by Theorem 2.2, $Q_2 \vdash \theta\tau_1 \approx \theta\tau$ with $Q_2 \sqsubseteq (Q_1, \theta)$.

Case [INST-QUANTIFY]: We have from the premise $Q_1 \mid \Gamma \vdash \sigma \sqsubseteq \tau' \rightsquigarrow \overline{es}_1$ with $\vDash (\theta, Q_1)$. By induction, $Q_2 \mid \theta\Gamma \vDash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_2$ with $\overline{es}_1 = \overline{es}_2$ and $Q_1 \sqsubseteq (Q_2, \theta)$. With the same fresh α, we can derive $Q_2 \mid \theta\Gamma \vdash \theta(\forall\alpha.\sigma) \sqsubseteq \theta\tau \rightsquigarrow \overline{es}_2$.

□

Now we are able to establish stability over variable resolution: if $Q_1 \mid \Gamma \vdash z \overset{\leftarrow}{:} \tau \rightsquigarrow e$ with $\vDash (\theta, Q_1)$ (**I**), then we also have $Q_2 \mid \theta\Gamma \vdash z \overset{\leftarrow}{:} \theta\tau \rightsquigarrow e$ with $Q_1 \sqsubseteq (\theta, Q_2)$.

Proof. (*Of Theorem 2.4*) From the premise of [VAR], we have $\exists! e$ (**1a**) with $Q_1 \mid \Gamma \vdash z \overset{\leftarrow}{:} \tau \rightsquigarrow e$ (**1b**). By Theorem B.21 (1a,1b), we also have $Q_2 \mid \theta\Gamma \vdash z \overset{\leftarrow}{:} \theta\tau \rightsquigarrow e$ (**2a**) with $Q_1 \sqsubseteq (Q_2, \theta)$ (**2b**) satisfying $\exists e$. Now, we also need to show there are no *other* derivations $Q \mid \theta'\Gamma \vdash z \overset{\leftarrow}{:} \theta'\tau \rightsquigarrow e'$ for some θ' with $e \neq e'$. However, if such derivation exists, we also have by Lemma B.19, $Q' \mid \Gamma \vdash z \overset{\leftarrow}{:} \tau \rightsquigarrow e'$ – but that contradicts (1a,1b). □

B.4.3 Polymorphic Stability. We prove Theorem 2.5: If using the [ANN-SCHEME] rule, we have $Q \mid \Gamma \vdash t \overset{\leftarrow}{:} \forall\overline{\alpha}.\rho \rightsquigarrow e$, and for any $\overline{\tau}$ with $\mathrm{ftv}(\overline{\tau}) \subseteq \mathrm{ftv}(\Gamma)$ (**I**), then also $Q' \mid \Gamma \vdash t \overset{\leftarrow}{:} [\overline{\alpha}{:=}\overline{\tau}]\rho \rightsquigarrow e$.

Proof. (*Of Theorem 2.5*) By the premise, $Q_1 \mid \Gamma, \overline{\alpha} \vdash t \overset{\leftarrow}{:} \rho \rightsquigarrow e$ (**1a**), with $(Q, \forall\overline{\alpha}.\rho) = \mathrm{gen}(Q_1, \Gamma, \rho)$ (**1b**) and fresh $\overline{\alpha}$ (**1c**). By (1b), we must have $\overline{\alpha} \notin \mathrm{dom}(Q_1)$ (**1d**) and $\overline{\alpha} \notin \mathrm{ftv}(Q)$.

We can now show that we can also derive $Q' \mid \Gamma \vdash t \coloneqq [\overline{\alpha}{:=}\overline{\tau}]\rho \rightsquigarrow e$ by exactly following the derivation of (1a). The interesting cases are at the [VAR] leaf nodes. In the original derivation, we have a leaf $Q_3 \mid \Gamma, \overline{\alpha}, \Gamma' \vdash z \coloneqq \tau \rightsquigarrow e_3$ (**2a**) and in the new one we need $Q_4 \mid \Gamma[\overline{\alpha}{:=}\overline{\tau}]\Gamma' \vdash z \coloneqq [\overline{\alpha}{:=}\overline{\tau}]\tau \rightsquigarrow e_3$. We must have $\overline{\alpha} \notin \mathrm{dom}(Q_3)$ by (1d), and therefore $\vDash ([\overline{\alpha}{:=}\overline{\tau}], Q_3)$ (**2b**). We can now use Theorem 2.4 with (2a,2b), to derive $Q_4 \mid [\overline{\alpha}{:=}\overline{\tau}]\Gamma, \overline{\alpha}, \Gamma' \vdash z \coloneqq [\overline{\alpha}{:=}\overline{\tau}]\tau \rightsquigarrow e_3$ (**3a**). Since the $\overline{\alpha}$ bindings are anonymous, and fresh $\overline{\alpha}$, we have $Q_4 \mid \Gamma, [\overline{\alpha}{:=}\overline{\tau}]\Gamma' \vdash z \coloneqq [\overline{\alpha}{:=}\overline{\tau}]\tau \rightsquigarrow e_3$ (**3b**).

(As an aside, this is the situation where using skolem constants are unstable: with skolems, we may originally have a leaf node in type equivalence as $\not\vdash \tau' \approx \tau[\overline{\alpha}{:=}\overline{c}]$. But if we later instantiate the $\overline{\alpha}$, we get $\_ \vdash \tau' \approx [\overline{\alpha}{:=}\overline{\tau}]$ and now this may hold, which can lead to a new valid derivation, and later on the $\exists! e$ condition in [VAR] cannot be satisfied.) □

B.4.4 Supporting Lemmas.

Lemma B.22. (*Prefix composition can be composed*)
If $\mathrm{ftv}(Q_1) \not\cap \mathrm{dom}(Q_2)$, then $(Q_1, Q_2) = Q_1 \circ Q_2$.

Lemma B.23.
If $Q \mid \theta\Gamma \vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow \overline{e}$, then $\mathrm{ftv}(Q) \not\cap \mathrm{dom}(\theta)$, and by Lemma B.22, $(Q, \theta) = Q \circ \theta$.

Lemma B.24.
If $Q \mid \theta\Gamma \vdash \theta\sigma \sqsubseteq \theta\tau \rightsquigarrow \overline{e}$, then also $Q' \mid \Gamma \vdash \sigma \sqsubseteq \tau \rightsquigarrow \overline{e}'$

We have [Leijen and Ye 2024,D.27)]:

Theorem B.25. (*Type equivalent substitution*)
If $Q_1 \vdash Q[\tau_1] \approx Q[\tau_2]$, then also $Q_2 \vdash \tau_1 \approx \tau_2$ with $(Q, Q_1) = (Q, Q_2)$.

and also [Leijen and Ye 2024,Lemma 3.9]:

Lemma B.26. (*Extraction corresponds to composition of prefix solutions*)
If $\vDash Q$ and $Q = Q' \cdot \alpha{=}\tau$, then $\langle Q \rangle = \langle Q' \rangle \circ [\alpha{:=}\tau]$.

where

$$\frac{Q = Q' \cup \{\alpha{=}\tau\} \qquad \alpha \notin \mathrm{ftv}(Q', \tau)}{Q \;=\; Q' \cdot \alpha{=}\tau}\ \textsc{extract}$$

We can also simplify duplicate bindings [Leijen and Ye 2024,Theorem 2.7]:

Theorem B.27. (*Simplify*)
If $Q' \vdash \tau_1 \approx \tau_2$, then $Q \cup \{\alpha{=}\tau_1,\ \alpha{=}\tau_2\} \;=\; Q \cup Q' \cup \{\alpha{=}\tau_1\}$

C UNIFICATION AND PREFIX COMPOSITION

Leijen and Ye [2025] show how to implement prefix composition (*compose*) and type equivalence (*unify*). Unfortunately, in the original publication their formulation is not technically complete unless we assume prefixes are always *well-formed*.

The reason for this is rather subtle, and requires careful reasoning about equivalent substitutions. We say two substitutions θ_1, θ_2 are equivalent whenever each is an instance of the other: $\theta_1 \sqsubseteq \theta_2 \wedge \theta_2 \sqsubseteq \theta_1$ (where $\theta \sqsubseteq \theta'$ iff $\theta' = \theta'' \circ \theta$ for some θ''). This means that a substitution from a type variable to another has no direction: $[\alpha{:=}\beta] \equiv [\beta{:=}\alpha]$ since each is an instance of the other.

Since equality of prefixes is defined as equivalence of the minimal solutions, this is also the case for prefixes, where $\{\alpha{=}\beta\} = \{\beta{=}\alpha\}$. For the *solve* algorithm, we need to define an ordering on such type variable equalities in order to make it easier to correctly detect cycles. We assume there is some lexical ordering of type variables such that for every type variable equality $\alpha{=}\beta$, we have

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{unify} : (\tau,\tau) \to \theta} \\
\mathit{unify}(\alpha,\alpha) & & = \mathit{id} \\
\mathit{unify}(\alpha,\beta) & \mid \alpha \neq \beta & = \text{if } \alpha < \beta \text{ then } [\alpha{:=}\beta] \text{ else } [\beta{:=}\alpha] \\
\mathit{unify}(\alpha,\tau) \text{ or } (\tau,\alpha) & \mid \alpha \notin \mathsf{ftv}(\tau) & = [\alpha{:=}\tau] \\
\mathit{unify}(\tau_1 \to \tau_2,\ \tau_1' \to \tau_2') & & = \text{let } \theta_1 = \mathit{unify}(\tau_1,\tau_1');\ \theta_2 = \mathit{unify}(\tau_2,\tau_2') \text{ in } \mathit{solve}(\theta_1 \cup \theta_2) \\
\mathit{unify}(\_,\_) & & = \mathsf{fail}()
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{solve} : Q \to \theta} \\
\mathit{solve}(\varnothing) & = \mathit{id} & \\
\mathit{solve}(Q \uplus \{\alpha{=}\tau\}) & = \mathit{solve}(Q) \circ [\alpha{:=}\tau] & \text{if } \alpha \notin \mathsf{ftv}(Q,\tau) \\
\mathit{solve}(Q \uplus \{\alpha{=}\tau_1, \alpha{=}\tau_2\}) & = \mathit{solve}(Q \cup Q' \cup \{\alpha{=}\tau_1\}) & \text{if } Q' = \mathit{unify}(\tau_1,\tau_2) \ \wedge \alpha \notin \mathsf{ftv}(\tau_1,\tau_2,\mathsf{rng}(Q)) \\
\mathit{solve}(\_) & = \mathsf{fail}() &
\end{array}
$$

$$
\begin{array}{l}
\mathit{compose} : (Q,Q) \to \theta \\
\mathit{compose}(Q_1,Q_2) \ = \ \mathit{solve}(Q_1 \cup Q_2)
\end{array}
$$

Fig. 7. Unification and solving of prefixes (where we use $\uplus$ for disjoint union).

that $\alpha < \beta$ (and $\beta{=}\alpha$ is never present). Moreover, we also assume we never have $\alpha{=}\alpha$ constraints. We call such prefixes well formed.

We can easily ensure well-formed prefixes by adapting the *unify* algorithm [Leijen and Ye 2025] to include one extra case for type variable equalities, which ensures the constraint $\alpha{=}\beta$ is always in the correct order with $\alpha < \beta$. The full rules for *unify*, *solve*, and *compose* are given in Figure 7.

Essentially the *solve* algorithm picks non-dependent bindings and composes them recursively, while simplifying duplicate bindings away by unifying their types using the *unify* function. Note that the extra case for variable equalities also ensures now that $\mathit{unify}(\tau_1,\tau_2) = \mathit{unify}(\tau_2,\tau_1)$.

As an aside, the need for well-formed prefixes can be illustrated by using the ill-formed prefix $\{\beta{=}\alpha, \alpha{=}\beta, \alpha{=}\mathit{int}\}$. In such case *solve* would fail while there actually exists a valid substitution (i.e. *solve* is incomplete for ill-formed prefixes). In contrast, the well-formed equivalent $\{\alpha{=}\beta, \alpha{=}\beta, \alpha{=}\mathit{int}\}$ solves indeed to $[\alpha{:=}\mathit{int}, \beta{:=}\mathit{int}]$.

C.1 Soundness

Lemma C.28. (*Unification is Sound*)
If $\langle Q \rangle = \mathit{unify}(\tau_1,\tau_2)$, then $Q \vdash \tau_1 \approx \tau_2$.

Lemma C.29. (*Solve is sound*)
If $\theta = \mathit{solve}(Q)$ (with a well-formed Q), then $\models Q$ and $\theta = \langle Q \rangle$.

We establish soundness of *unify* and *solve* together, since they are mutually recursive.

Proof. (*Of Lemma C.28 and Lemma C.29*) By induction on the rules of *unify* and *solve*.

Case $\mathit{unify}(\alpha,\alpha)$: we have $\langle Q \rangle = \mathit{id}$ and thus $Q = \varnothing$. By [EQ-ID], we have $\varnothing \vdash \alpha \approx \alpha$.

Case $\mathit{unify}(\alpha,\beta)$ with $\alpha \neq \beta$: With $\alpha < \beta$, we have $Q = \{\alpha{=}\beta\}$, and by [EQ-VAR] we have $Q \vdash \alpha \approx \beta$. For $\beta < \alpha$, we have $Q = \{\beta{=}\alpha\}$, and by [EQ-VAR] and [EQ-REFL], we also have $Q \vdash \beta \approx \alpha$.

Case $\mathit{unify}(\alpha,\tau)$,$\alpha \notin \mathsf{ftv}(\tau)$: we have $Q = \{\alpha{=}\tau\}$, and by [EQ-VAR] we have $Q \vdash \alpha \approx \tau$.

Case $\mathit{unify}(\tau,\alpha)$,$\alpha \notin \mathsf{ftv}(\tau)$: we have $Q = \{\alpha{=}\tau\}$, and by [EQ-VAR] and [EQ-REFL], we have $Q \vdash \tau \approx \alpha$.

Case $\mathit{unify}(\tau_1 \to \tau_2,\ \tau_1' \to \tau_2')$: From the premises, we have $Q_1 = \mathit{unify}(\tau_1,\tau_1')$ (**1a**), $Q_2 = \mathit{unify}(\tau_2,\tau_2')$ (**1b**), and $Q = \mathit{solve}(Q_1 \cup Q_2)$ (**1c**). By induction over (1a,1b), we also have $Q_1 \vdash \tau_1 \approx \tau_1'$ and $Q_2 \vdash \tau_2 \approx \tau_2'$.

Moreover, by induction on $solve(Q_1 \cup Q_2)$, we have $\vDash (Q_1 \cup Q_2)$ with $\langle (Q_1, Q_2) \rangle = solve(Q_1, Q_2)$. We can now use [EQ-FUN] to derive $(Q_1, Q_2) \vdash \tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'$.

Case $unify(\tau_1, \tau_2) = \mathsf{fail}()$: in this case the precondition is not satisfied.

For the rules of *solve*, we have:

Case $solve(\varnothing)$: we have trivially $\vDash \varnothing$, and $\theta = id = \langle \varnothing \rangle$.

Case $solve(Q \uplus \{\alpha{=}\tau\})$ with $\alpha \notin \mathsf{ftv}(Q)$ (**1a**): we have $\theta = solve(Q) \circ [\alpha{:=}\tau]$ (**1b**). By induction on $solve(Q)$, we have $solve(Q) = \langle Q \rangle$ (**2a**) with $\vDash Q$ (**2b**). By (1a,2b), we also have $\vDash (Q \cup \{\alpha{=}\tau\})$. By Theorem B.26, we have $\langle (Q \cup \{\alpha{=}\tau\}) \rangle = \langle Q \rangle \circ [\alpha{:=}\tau]$ (**3**). Therefore

$$
\begin{array}{lll}
 & \theta & \\
= & solve(Q) \circ [\alpha{:=}\tau] & \{\ (1b)\ \} \\
= & \langle Q \rangle \circ [\alpha{:=}\tau] & \{\ (2a)\ \} \\
= & \langle Q \cup \{\alpha{=}\tau\} \rangle & \{\ (3)\ \}
\end{array}
$$

Case $solve(Q \uplus \{\alpha{=}\tau_1,\ \alpha{=}\tau_2\})$: we have $\langle Q' \rangle = unify(\tau_1, \tau_2)$ (**1a**) and $\theta = solve(Q \cup Q' \cup \{\alpha{=}\tau_1\})$ (**1b**). By induction on (1a), $Q' \vdash_{\mathsf{T1}} \approx \tau_2$ (**2a**), and by induction on (1b), $\vDash (Q \cup Q' \cup \{\alpha{=}\tau_1\})$ (**2b**) where $\theta = \langle (Q \cup Q' \cup \{\alpha{=}\tau_1\}) \rangle$ (**2c**). From Theorem B.27 and (2a), we also have $(Q \cup \{\alpha{=}\tau_1\} \cup \{\alpha{=}\tau_2\}) = (Q \cup Q' \cup \{\alpha{=}\tau_1\})$, and thus $\theta = \langle (Q \cup \{\alpha{=}\tau_1\} \cup \{\alpha{=}\tau_2\}) \rangle$ (2c).

Case $solve(Q) = \mathsf{fail}()$: in this case the precondition is not satisfied.

□

C.2 Completeness

Lemma C.30. (*Unify is complete*)
If $Q \vdash \tau_1 \approx \tau_2$, then $\langle Q \rangle = unify(\tau_1, \tau_2)$.

Lemma C.31. (*Solve is complete*)
If $\vDash Q$ (with a well-formed Q), then $\theta = solve(Q)$ with $\theta = \langle Q \rangle$.

Again, we establish completeness of *unify* and *solve* together since they are mutually recursive. For clarity, we split the proofs and assume completeness of the other in each proof.

Proof. (*Of Lemma C.30*) We proceed by induction over the derivation (assuming *solve* is complete):

Case [EQ-ID]: We have $\varnothing \vdash \tau_1 \approx \tau_2$ with $\tau_1 = \tau_2$. If we only have function arrows and variables, we can repeatedly apply [EQ-FUN] (with $Q_1 = Q_2 = \varnothing$), ending in $\varnothing \vdash \alpha \approx \alpha$. In that case we have $unify(\alpha, \alpha) = id$ (with $id = \langle \varnothing \rangle$). For each [EQ-FUN] we can apply the $unify(\tau_1 \to \tau_2, \tau_3 \to \tau_4)$ rules where $solve(\varnothing, \varnothing) = id$ again.

Case [EQ-VAR]: We have $\{\alpha{=}\tau\} \vdash \alpha \approx \tau$ (**1a**) with $\alpha \notin \mathsf{ftv}(\tau)$ (**1b**). Suppose $\tau = \beta$ (with $\alpha \neq \beta$ (1b)). If $\alpha < \beta$, $unify(\alpha, \beta) = [\alpha{:=}\beta] = \langle \{\alpha{=}\beta\} \rangle$. If $\beta < \alpha$, we have $unify(\alpha, \beta) = [\beta{:=}\alpha]$ but that equals also $\beta{:=}\alpha$ and $= \langle \{\alpha{=}\beta\} \rangle$. Otherwise, $\tau \neq \beta$, and by (1b), we have $unify(\alpha, \tau) = [\alpha{:=}\tau] = \langle \{\alpha{=}\tau\} \rangle$.

Case [EQ-REFL]: We have $Q \vdash \tau_2 \approx \tau_1$ by the premise, and by induction $unify(\tau_2, \tau_1) = \langle Q \rangle$. We have $unify(\tau_2, \tau_1) = unify(\tau_1, \tau_2)$, and therefore we also have $\langle Q \rangle = unify(\tau_1, \tau_2)$.

Case [EQ-FUN]: We have $Q_1 \vdash \tau_1 \approx \tau_3$ (**1a**) and $Q_2 \vdash \tau_2 \approx \tau_4$ (**1b**) by the premises. The $unify(\tau_1 \to \tau_2, \tau_3 \to \tau_4)$ case applies, and we have by induction over (1a,1b), $\langle Q_1 \rangle = unify(\tau_1, \tau_3)$ and $\langle Q_2 \rangle = unify(\tau_2, \tau_4)$, and by Lemma C.31, $\langle (Q_1, Q_2) \rangle = solve(Q_1, Q_2)$.

Case If no derivation rules apply, we also have that none of the previous *unify* cases apply, and we have $unify(\tau_1, \tau_2) = \mathsf{fail}()$.

□

For showing completeness of *solve*, we also need to establish a decreasing measure for the induction.

For well-formed prefixes, we can define a stable *degree* of a prefix (which decreases in each recursive step). First we define the dependencies of α with respect to a Q as:

- $\beta \in \mathit{deps}(\alpha)$ if $\alpha{=}\tau \in Q$ and $\beta \in \mathsf{ftv}(\tau)$.
- $\gamma \in \mathit{deps}(\alpha)$ if $\beta \in \mathit{deps}(\alpha)$ and $\gamma \in \mathit{deps}(\beta)$ (transitive closure).

We say α is independent of β, $\alpha \prec \beta$, iff $\alpha \notin \mathit{deps}(\beta)$ (where we have both $\alpha \prec \beta$ and $\beta \prec \alpha$ for indepedent type variables). The degree of Q is now the number of occurrences of distinct type variables in the domain ordered by $\prec$ (in some order). For example, $\mathit{degree}(\{\beta{=}\gamma, \alpha{=}\mathit{int}{\to}\beta, \gamma{=}\mathit{int}, \alpha{=}\beta{\to}\mathit{int}\})$ $= (2,1,1)$ (for (α,β,γ)). We have that if $\alpha \notin \mathsf{codom}(Q)$, then $\forall \beta \in \mathsf{dom}(Q).\ \alpha \prec \beta$ (**I**).

Proof. (*Of Lemma C.31*) We proceed by induction on the degree and shape of Q:

Case $Q = \varnothing$: in that case $\mathit{solve}(\varnothing) = \mathit{id}$ applies where $\mathit{id} = \langle\varnothing\rangle$.

Otherwise $Q \neq \varnothing$. Suppose we have that for all $\alpha \in \mathsf{dom}(Q)$ that $\alpha \in \mathsf{codom}(Q)$.

Since all domain variables appear in the co-domain, by the pigeon hole principle such prefix must contain a cycle, for example, $\{\alpha{=}\mathit{int},\ \beta{=}\alpha{\to}\gamma,\ \gamma{=}\beta{\to}\beta\}$. Since we order type variable equalities in a well-formed prefix, it is not possible for such cycle to consist of only type variable equalities (like $\{\alpha{=}\beta, \beta{=}\alpha\}$). Therefore, in such cycle there must be some $\alpha{=}\tau$ with τ being a larger type (e.g. $\tau_1{\to}\tau_2$). However, that makes it no longer possible to create an idempotent substitution and $\not\models Q$, contradicting the assumption.

Therefore, we know there must be at least some $\alpha \in \mathsf{dom}(Q)$ where $\alpha \notin \mathsf{codom}(Q)$ (**1**). We proceed now by case analysis:

Case $Q = Q' \uplus \{\alpha{=}\tau\}$ with $\alpha \notin \mathsf{dom}(Q')$. In that case, with (1) we have $\alpha \notin \mathsf{ftv}(Q', \tau)$ (**2**) and the second case of *solve* applies. By induction on the decreasing degree, $\langle Q'\rangle = \mathit{solve}(Q')$ (**3**). By (2) and [EXTRACT], we have $Q = Q' \cdot \{\alpha{=}\tau\}$ and by Lemma B.26, $\langle Q\rangle = \langle Q'\rangle \circ [\alpha{:=}\tau]$. With (3) we now have $\langle Q\rangle = \mathit{solve}(Q') \circ [\alpha{:=}\tau]$.

Case Otherwise, we must have $Q = Q' \uplus \{\alpha{=}\tau_1, \alpha{=}\tau_2\}$. With (1), we have $\alpha \notin \mathsf{ftv}(\tau_1, \tau_2, \mathsf{rng}(Q))$ (**1a**), and the third case of *solve* applies. Since $\models Q$, by simplification (Theorem B.27), we have $Q' \cup \{\alpha{=}\tau_1,\ \alpha{=}\tau_2\} = Q' \cup Q'' \cup \{\alpha{=}\tau_1\}$ (**2a**), with $Q'' \vdash \tau_1 \approx \tau_2$ (**2b**).

From the completeness of *unify*, and (2b) we have $\langle Q''\rangle = \mathit{unify}(\tau_1, \tau_2)$ (**3**). By (2b) $\mathsf{ftv}(Q'') \subseteq \mathsf{ftv}(\tau_1, \tau_2)$, and thus by (1a), we have one less $\alpha \in \mathsf{dom}(Q' \cup Q'' \cup \{\alpha{=}\tau_1\})$. Moreover, since $\alpha \notin \mathsf{codom}(Q' \cup Q'')$ and (I), the degree decreases, and by induction $\langle Q\rangle = \mathit{solve}(Q' \cup Q'' \cup \{\alpha{=}\tau_1\})$.

Since this covers all forms of Q with $\models Q$, the fail case never applies. □