COCHIS: Stable and coherent implicits

Under consideration for publication in J. Functional Programming

COCHIS: *Stable and Coherent Implicits*

TOM SCHRIJVERS
KU Leuven, Belgium
BRUNO C.D.S. OLIVEIRA
The University of Hong Kong
PHILIP WADLER
University of Edinburgh, UK
KOAR MARNTIROSIAN
KU Leuven, Belgium

Abstract

Implicit Progamming (IP) mechanisms infer values by type-directed resolution, making programs more compact and easier to read. Examples of IP mechanisms include Haskell's type classes, Scala's implicits, Agda's instance arguments, Coq's type classes, and Rust's traits. The design of IP mechanisms has led to heated debate: proponents of one school argue for the desirability of strong reasoning properties; while proponents of another school argue for the power and flexibility of local scoping or overlapping instances. The current state of affairs seems to indicate that the two goals are at odds with one another and cannot easily be reconciled.

This paper presents COCHIS, the Calculus Of CoHerent ImplicitS, an improved variant of the implicit calculus that offers flexibility while preserving two key properties: *coherence* and *stability of type substitutions*. COCHIS supports polymorphism, local scoping, overlapping instances, first-class instances, and higher-order rules, while remaining type safe, coherent and stable under type substitution.

We introduce a logical formulation of how to resolve implicits, which is simple but ambiguous and incoherent, and a second formulation, which is less simple but unambiguous, coherent and stable. Every resolution of the second formulation is also a resolution of the first, but not conversely. Parts of the second formulation bear a close resemblance to a standard technique for proof search called focusing. Moreover, key for its coherence is a rigorous enforcement of determinism.

1 Introduction

Programming language design is usually guided by two, often conflicting, goals: *flexibility* and *ease of reasoning*. Many programming languages aim at providing powerful, flexible language constructs that allow programmers to achieve reuse, and develop programs rapidly and concisely. Other programming languages aim at easy reasoning about programs, as well as at avoiding programming pitfalls. Often the two goals are at odds with each other, since highly flexible programming mechanisms make reasoning harder. Arguably the art of programming language design is to reconcile both goals.

A concrete case where this issue manifests itself is in the design of *Implicit Programming* (IP) mechanisms. Implicit programming denotes a class of language mechanisms which infer values by using type information. Examples of IP mechanisms include Haskell's type

classes (Wadler & Blott, 1989), C++'s concepts (Gregor *et al.*, 2006), JavaGI's generalized interfaces (Wehr *et al.*, 2007), Coq's type classes (Sozeau & Oury, 2008), Scala's implicits (Odersky, 2010), Agda's *instance arguments* (Devriese & Piessens, 2011), Rust's *traits* (Mozilla Research, 2017), and OCaml's *modular implicits* (White *et al.*, 2014). IP can also be viewed as a form of (type-directed) program synthesis (Manna & Waldinger, 1980). The programming is said to be *implicit*, because expressions (e.g., function arguments) can be omitted by the programmer. Instead the necessary values are provided automatically via a *type-directed resolution* process. These implicit values are either fetched by type from the current (implicit) environment or constructed by type-directed rules.

Currently there are two main schools of thought regarding the design of IP mechanisms. Haskell's type classes (Wadler & Blott, 1989) embody the first school of thought, which is guided by the *ease of reasoning* qualities of pure functional languages, and the *predictability* of programs. To ensure these goals the semantics of the language should satisfy a number of properties. One of those properties is *coherence* (Reynolds, 1991; Jones, 1992). The original definition of coherence in the literature is that any valid program must have exactly one meaning (that is, the semantics is not ambiguous). In fact Haskell type classes are supposed to support an even stronger property, the so-called *global uniqueness* of instances (Zhang, 2014). Global uniqueness ensures that *at any point in a program, and independently of the context* the type-directed resolution process always returns the same value for the same resolved type. This is a consequence of Haskell having a restriction of at most one instance of a type class per type in a program.

While both coherence and global uniqueness of instances are preserved in Haskell, this comes at a cost. Since the first implementations of type classes, Haskell imposes several restrictions to guarantee those properties. Various past work has indicated limitations of type classes (Kahl & Scheffczyk, 2001; Camarão & Figueiredo, 1999; Dijkstra & Swierstra, 2005; Dreyer *et al.*, 2007; Garcia *et al.*, 2007; Oliveira *et al.*, 2010; Morris & Jones, 2010; Oliveira *et al.*, 2012). In particular, type classes allow at most one instance per type (or severely restrict overlapping instances) to exist in a program. That is, all instances must be visible globally and local scoping of instances is not allowed. This form of global scoping goes against modularity. Other restrictions of type classes are that they are not first-class values and that the type-directed rules cannot be higher-order (Oliveira *et al.*, 2012).

Advanced features of type classes, such as overlapping instances (GHC, 2017), also pose severe problems, since they go against the principle of one instance per type. One issue is that "*when more specific overlapping instances are added, the proofs of some predicates will change to use the new instances*" (Morris & Jones, 2010). In essence special care (via restrictions) is needed to preserve coherence in the presence of overlapping instances. Another important property that is broken in the presence of overlapping instances (if special care is not taken) is the so-called *stability* of type substitutions. The issue is that the behaviour of resolution for an expression e can change if e gets a more specific type, leading to a different evaluation result. This is problematic because seemingly harmless inlinings will actually have a different semantics before and after the inlining. Because of this problem, the design of Haskell type classes significantly restricts the set of valid overlapping instances to ensure that stability holds, and the meaning of an expression does not change simply due to a more specific type. In other words, resolution should resolve implicit values using the same rules before and after instantiation of type variables.

As an orthogonal remark, in the Haskell community, the term coherence is often colloquially used to encompass several different properties, including global uniqueness, stability and the original coherence definition by Reynolds (Reynolds, 1991). In the context of an elaboration semantics (which the standard semantics style for IP mechanisms), the definition of coherence that follows from Reynolds one is that *all possible elaborations for a program possess the same meaning*. However such a definition of coherence is narrower than the colloquial term usually used in the Haskell community. It is important to note that the three properties are distinct. For example global uniqueness implies coherence, but not the converse. Furthermore it is also possible to have coherence, but not stability. In this paper we will use coherence in Reynolds' sense, and be precise about the different properties under discussion.

An alternative school of thought in the design of IP mechanisms favours *flexibility*. For example, Scala's implicits and Agda's instance arguments do not impose all of the type class restrictions. Scala supports local scoping of instances, which allows distinct instances to exist for the same type in different scopes of the same program. Scala also allows a powerful form of overlapping implicits (Oliveira *et al.*, 2010). The essence of this style of implicit programming is modelled by the *implicit calculus* (Oliveira *et al.*, 2012) or the more recent SI calculus (Odersky *et al.*, 2017). The implicit calculus supports a number of features that are not supported by type classes. Besides local scoping, in the implicit calculus *any type* can have an implicit value. In contrast Haskell's type class model only allows instances of classes (which can be viewed as a special kind of record) to be passed implicitly. Finally the implicit calculus supports higher-order instances/rules: that is rules whose requirements can themselves be other rules. The implicit calculus has been shown to be type-safe, and it also ensures coherence, but it lacks stability. The SI calculus lacks both coherence and stability, but the authors present a simply-typed subset that is coherent[1]. Unfortunately, while both the implicit/SI calculus and the various existing language mechanisms embody flexibility, the lack of important properties such as stability makes reasoning about the semantics of programs harder, and can prevent refactorings and compiler optimizations such as inlining.

The design of IP mechanisms has led to heated debate (Hulley, 2009; Zhang, 2014; Kmett, 2015) about the pros and cons of each school of thought: ease of reasoning versus flexibility. Proponents of the Haskell school of thought argue that coherence, stability and uniqueness of instances are extremely desirable, and flexibility should not come at the cost of those properties. Proponents of flexible IP mechanisms argue that flexibility is more important, and that uniqueness of instances goes against modularity. As far as we are aware there are no current designs that support local scoping, overlapping instances and first-class and higher-order rules, while at the same time ensuring both coherence and stability.

This paper presents COCHIS: the Calculus Of CoHerent ImplicitS. COCHIS is an improved variant of the implicit calculus that guarantees *coherence* and *stability*. COCHIS supports local scoping, overlapping instances, first-class instances and higher-order rules. Yet, in contrast to most previous work that supports these features, the calculus is not

[1] It makes no sense to talk about stability of type substitutions in a simply typed calculus, since this is a property that is only relevant for polymorphic calculi.

only type-safe, but also stable and coherent. Naturally, the unrestricted calculus does not support global uniqueness of instances, since this property depends on the global scoping restriction. Nevertheless, if retaining global uniqueness is desired, that can be modeled by the subset of COCHIS without local declarations. In fact, based on the resolution algorithm of an early version of COCHIS, GHC has been recently extended with *quantified class constraints* (Bottu *et al.*, 2017; GHC, 2017). Quantified class constraints enable higher-order rules in a setting with global scoping.

Ensuring coherence and stability in the presence of COCHIS's overlapping and higher-order rules is particularly challenging. We introduce a logical formulation of how to resolve implicits, which is simple but ambiguous and incoherent, and a second formulation, which is less simple but unambiguous and coherent. Every resolution of the second formulation is also a resolution of the first, but not conversely. Parts of the second formulation bear a close resemblance to a standard technique for proof search in logic called *focusing* (Andreoli, 1992; Miller *et al.*, 1991; Liang & Miller, 2009). However, unlike focused proof search, which is still essentially non-deterministic, COCHIS's resolution employs additional techniques to be entirely deterministic and thus obviously coherent. In particular, unlike focused proof search, our resolution uses a stack discipline to prioritize implicits, and removes any recursive resolutions from matching decisions. Moreover, further restrictions are needed to obtain stability.

In summary, our contributions are as follows:

- We present COCHIS, a *coherent*, *stable* and *type-safe* formal model for implicit programming that supports local scoping, overlapping implicits, first-class implicits and higher-order rules.
- We significantly improve the design of resolution over the existing work on the implicit calculus by Oliveira et al. (2012). The new design for resolution is more powerful and expressive; it is closely based on principles of logic and the idea of propositions as types (Wadler, 2015); and is related to the idea of focusing in proof search.
- COCHIS comes with a semantics in the form of a type-directed elaboration to System F.
- We provide a unification-based algorithm as an executable form of the declarative specification of resolution. A prototype implementation of this algorithm and of COCHIS is available at `https://bitbucket.org/tom_schrijvers/cochis/`.
- We establish key meta-theoretic properties of our system:
  - The elaboration is type-preserving.
  - Resolution is deterministic and thus obviously coherent.
  - Resolution is stable under type substitution, and type-directed elaboration is preserved by reduction of type application.
  - Our algorithm is sound and complete with respect to its declarative specification.

  The proofs are available in the appendix and at `https://bitbucket.org/KlaraMar/cochiscoq`

Organization Section 2 presents an informal overview of our calculus. Section 3 describes a polymorphic type system that statically excludes ill-behaved programs. Section 5

provides the elaboration semantics of our calculus into System F and correctness results. Section 6 discusses several of our design choices as well as some alternatives. Section 7 discusses related work and Section 8 concludes.

2 Overview

This section summarises the relevant background on type classes, IP and coherence, and introduces the key features of COCHIS. We first discuss Haskell type classes, the oldest and most well-established IP mechanism, then compare them to Scala implicits, and finally we introduce the approach taken in COCHIS.

2.1 Type Classes and Implicit Programming

Type classes enable the declaration of overloaded functions like comparison.

class *Ord* α **where**
  $(\leqslant) :: \alpha \to \alpha \to \mathit{Bool}$

A type class declaration consists of: a class name, such as *Ord*; a type parameter, such as α; and a set of method declarations, such as $(\leqslant)$. Each of the methods in the type class declaration should have at least one occurrence of the type parameter α in their signature.

Instances and Type-Directed Rules Instances implement type classes. For example, *Ord* instances for integers, characters, and pairs can be defined as follows:

instance *Ord Int* **where**
  $x \leqslant y = \mathit{primIntLe}\ x\ y$
instance *Ord Char* **where**
  $x \leqslant y = \mathit{primCharLe}\ x\ y$
instance $(\mathit{Ord}\ \alpha, \mathit{Ord}\ \beta) \Rightarrow \mathit{Ord}\ (\alpha, \beta)$ **where**
  $(x, x') \leqslant (y, y') = x \leqslant y \wedge (\neg\ (y \leqslant x) \vee x' \leqslant y')$

The first two instances provide the implementation of ordering for integers and characters, in terms of primitive functions. The third instance is more interesting, and provides the implementation of lexicographic ordering for pairs. In this case, the ordering instance itself *requires* an ordering instance for both components of the pair. These requirements are resolved by the compiler using the existing set of instances in a process called *resolution*. Using *Ord* we can define a generic sorting function

$$\mathit{sort} :: \mathit{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]$$

that takes a list of elements of an arbitrary type α and returns a list of the same type, as long as ordering is supported for type α[2].

[2] Note that, in Haskell, the single arrow ($\to$) denotes a function type constructor, whereas the double arrow ($\Rightarrow$) denotes a type with type class constraints (on the left of the arrow). In the *sort* function, for example, *Ord* α are the constraints.

Implicit Programming Type classes are an implicit programming mechanism because implementations of type class operations are automatically *computed* from the set of instances during the resolution process. For example, a call to *sort* only type checks if a suitable type class instance can be found. Other than that, the caller does not need to worry about the type class context, as shown in the following interaction with a Haskell interpreter:

$$\begin{array}{l} \mathit{Prelude} > \mathit{sort}\ [(3,\texttt{'a'}),(2,\texttt{'c'}),(3,\texttt{'b'})] \\ [(2,\texttt{'c'}),(3,\texttt{'a'}),(3,\texttt{'b'})] \end{array}$$

In this example, the resolution process combines the three *Ord* instances to find a suitable implementation for $\mathit{Ord}\ (\mathit{Int},\mathit{Char})$. The declarations given are sufficient to resolve an infinite number of other instances, such as $\mathit{Ord}\ (\mathit{Char},(\mathit{Int},\mathit{Int}))$ and the like.

One Instance Per Type A characteristic of (Haskell) type classes is that only one instance is allowed for a given type. For example, it is forbidden to include the alternative ordering model for pairs

$$\begin{array}{l} \textbf{instance}\ (\mathit{Ord}\ \alpha,\mathit{Ord}\ \beta) \Rightarrow \mathit{Ord}\ (\alpha,\beta)\ \textbf{where} \\ \quad (xa,xb) \leqslant (ya,yb) = xa \leqslant ya \wedge xb \leqslant yb \end{array}$$

in the same program as the previous instance because the compiler automatically picks the right type class instance based on the type parameter of the type class. If there are two type class instances for the same type, the compiler does not know which of the two to choose.

2.2 Coherence in Type Classes

An IP design is *coherent* if any valid program has exactly one meaning (that is, the semantics is not ambiguous). Haskell imposes restrictions to guarantee coherence. For example, Haskell rejects the expression:

$$\mathit{show}\ (\mathit{read}\ \texttt{"3"}) \equiv \texttt{"3"}$$

due to *ambiguity* of *type class resolution* (Jones, 1992). The functions *show* and *read* print and parse values of any type α that implements the classes *Show* and *Read*:

$$\begin{array}{l} \textbf{class}\ \mathit{Show}\ \alpha\ \textbf{where} \\ \quad \mathit{show} :: \alpha \rightarrow \mathit{String} \\ \textbf{class}\ \mathit{Read}\ \alpha\ \textbf{where} \\ \quad \mathit{read} :: \mathit{String} \rightarrow \alpha \end{array}$$

The program is rejected because there is more that one possible choice for α. For example α can be instantiated to *Int*, *Float*, or *Char*. Choosing $\alpha = \mathit{Float}$ leads to *False*, since showing the float 3 would result in `"3.0"`, while choosing $\alpha = \mathit{Int}$ leads to *True*. In other words if this expression was accepted then it could have multiple possible semantics. To ensure coherence instead of making an arbitrary choice Haskell rejects such programs.

Overlapping Instances Advanced features of type classes, such as overlapping instances, require additional restrictions to ensure coherence. The following program illustrates some of the issues:

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

This program declares a type class *Trans* α for defining transformations on some type α. The first instance provides a default implementation for any type, the identity transformation. The second instance defines a transformation for integers only.

An important question here is what happens if we write an expression like *trans* 3. Shall we pick the first or the second instance? Ideally this question should be answered by the language specification. One possible choice for the specification (not the Haskell choice) would be to allow any matching instance to be used, but this choice would lead to incoherence, since *trans* 3 could then both evaluate to 3 or 4. Instead, for overlapping instances, the GHC documentation (GHC, 2017) makes a different choice and declares that the *most specific instance* should be chosen. For the expression *trans* 3, the most specific instance is *Trans Int* and the expression evaluates to 4.

For the particular program *trans* 3, the Haskell specification manages to avoid incoherence by using the most specific instance, which ensures an unambiguous semantics. Thus, Haskell preserves coherence in the presence of a certain kind of overlapping instances, but there are other problematic overlapping instances that threaten the coherence property.

Incoherent Instances With overlapping instances, it is not always the case that a most specific instance exists. Consider the following type class and instance declarations:

class $C\ \alpha\ \beta$ **where**
  $m :: \alpha \to \beta \to Bool$
instance $C\ Bool\ \alpha$ **where**
  $m\ x\ y = x$
instance $C\ \alpha\ Bool$ **where**
  $m\ x\ y = y$

If we write the following program

incoherent $::$ *Bool*
incoherent $=$ *m True False* -- rejected without IncoherentInstances extension

then there is no most specific instance: both instances are equally specific. In this case, even with the overlapping instances extension activated, Haskell rejects the program.

However, Haskell also supports an additional extension, called `IncoherentInstances`, for allowing a more general kind of overlapping instances. With `IncoherentInstances` activated, Haskell accepts the *incoherent* definition. The (informal) language specification (GHC, 2017) for `IncoherentInstances` essentially says that in such a situation any matching instance could be picked. Thus, either of the two instances above can be picked, producing different evaluation results for the expression. Thus, as the name indicates, the expression *incoherent* leads to incoherence.

2.3 Stability in Type Classes

Another important property that is closely related to coherence is *stability*. Informally, stability ensures that instantiation of type variables does not affect resolution. Unfortunately overlapping instances threaten this property. Consider the following declaration, that uses the *trans* method from the type class *Trans* and the two instances declared previously:

$$bad :: \alpha \to \alpha$$
$$bad\ x = trans\ x \quad \text{-- unstable definition!}$$

Note here that the type of *bad* is $\alpha \to \alpha$ instead of $Trans\ \alpha \Rightarrow \alpha \to \alpha$. In Haskell both signatures can be accepted, but they are not equivalent. With $Trans\ \alpha \Rightarrow \alpha \to \alpha$ resolution is deferred to the call site of bad, allowing the instance of $Trans\ \alpha$ to be selected when α has been instantiated to a (potentially) more precise type. By contrast, with $\alpha \to \alpha$ resolution is applied eagerly, and an instance must be selected at the definition site. If Haskell were to accept this definition, it would have to implement *trans* using the first instance, since *trans* is applied at the arbitrary type α. Unfortunately this would mean that *bad* 3 returns 3 but *trans* 3 returns 4, even though *bad* and *trans* are defined to be equal, a nasty impediment to equational reasoning!

For this reason Haskell rejects the program by default. A programmer who really wants such behaviour can enable the `IncoherentInstances` compiler flag, which allows the program to type check.

Note that even though to allow *bad* we need the `IncoherentInstances` extension, which is suggestive of the definition breaking *coherence*, the issue here is not really coherence but rather stability. That is, type instantiation affects type class instantiation. As this example illustrates, instability is actually observable from a compiler implementation like GHC: we can observe that *bad* 3 and *trans* 3 behave differently. In contrast (in)coherence is not really observable from a compiler implementation: we need a language specification to understand whether there is incoherence or not. More concretely, with a single compiler implementation we can only observe the result of one possible elaboration of a program, but we cannot observe all the other possible elaborations allowed by the specification for that program. Therefore, we cannot observe incoherence.

The `IncoherentInstances` extension is understood to be highly problematic among Haskell programmers, since it can break both stability and coherence. Thus its use is greatly discouraged.

2.4 Global Uniqueness in Type Classes

A consequence of having at most one instance of a type class per type in a program is *global uniqueness* of instances (Zhang, 2014). That is, at any point in the program type class resolution for a particular type always resolves to the same value. Global uniqueness is a simple way to guarantee coherence, but it offers more than just coherence. The usefulness of this property is illustrated by a library that provides a datatype for sets that is polymorphic in the elements along with a *union* operation:

$$union :: Ord\ \alpha \Rightarrow Set\ \alpha \to Set\ \alpha \to Set\ \alpha$$

For efficiency reasons the sets are represented by a data structure that orders the elements in a particular way. It is natural to rely on the *Ord* type class to deal with ordering for the particular type α. To preserve the correct invariant, it is crucial that the ordering of elements in the set is always the same. The global uniqueness property guarantees this. If two distinct instances of *Ord* could be used in different parts of the program for the same type, then it would be possible to construct within the same program two sets using two different orderings (say ascending and descending order), and then break the ordering invariant by *union*-ing those two sets.

However, although global uniqueness is, in principle, a property that should hold in Haskell programs, Haskell implementations actually violate this property in various circumstances (nponeccop, 2012). In fact, it is acknowledged that providing a global uniqueness check is quite challenging for Haskell implementations (Marlow, 2012).

2.5 Scala Implicits and Stability

Scala implicits (Oliveira *et al.*, 2010) are an interesting alternative IP design. Unlike type classes, implicits are locally scoped. Consequently, Scala does not have the global uniqueness property, since different "instances" may exist for the same type in different scopes. Another important difference between implicits and type classes is that implicit parameters can be of any type, and there are no special constructs analogous to type class or instance declarations. Instead, implicits are modelled with ordinary types. They can be abstracted over and do not suffer from the second-class nature of type classes. These features mean that Scala implicits have a wider range of applications than type classes. Unlike Haskell type classes, however, with Scala implicits there is no way to enforce stability.

Modelling Type Classes with Implicits In Scala there is no special construct for defining the interface of a type class. Instead we can use regular interfaces to model type class interfaces. Scala models OO interfaces with traits (Scharli *et al.*, 2003). For example, the 3 interfaces presented in Section 2.1 can be modelled as:

$$\textbf{trait}\ \mathit{Ord}\ [T]\ \{\textbf{def}\ \mathit{le}\ (x:T, y:T):\mathit{Boolean}\}$$
$$\textbf{trait}\ \mathit{Show}\ [T]\ \{\textbf{def}\ \mathit{show}\ (x:T):\mathit{String}\}$$
$$\textbf{trait}\ \mathit{Read}\ [T]\ \{\textbf{def}\ \mathit{read}\ (x:\mathit{String}):T\}$$

Of course, by declaring traits like this, we still require explicit objects to call the methods on. To be able to use methods in the same way as Haskell type classes, the object (or dictionary) should be passed implicitly. This can be achieved by using Scala's implicits feature:

$$\textbf{def}\ \mathit{cmp}\ [A]\ (x:A, y:A)\ (\textbf{implicit}\ \mathit{ordD}:\mathit{Ord}\ [A]):\mathit{Boolean} = \mathit{ordD.le}\ (x,y)$$

The Scala definition of *cmp* plays the same role as $\leqslant$ in Haskell. The type of *cmp* states that *cmp* is parametrized in a type A, takes two (explicit) arguments of type A, and one implicit parameter (*ordD*). This is similar to a Haskell signature $\mathit{Ord}\ \alpha \Rightarrow \alpha \to \alpha \to \mathit{Bool}$, except that the implicit argument comes last. Additionally, unlike Haskell, at call sites it is possible to pass the implicit argument explicitly, if desired.

Context Bounds and Queries For the purposes of this paper we will, however, present *cmp* using *context bounds*: an alternative way to declare functions with constraints (or implicit arguments), supported in Scala. Context bounds are a simple syntactic sugar built on top of implicit parameters. With context bounds, the *cmp* definition is rewritten into:

$$\textbf{def}\ cmp\ [A : Ord]\ (x : A, y : A) : Boolean = ?[Ord\ [A]].le\ (x, y)$$

The notation $A : Ord$ is the so-called context bound. This notation enables us to declare a constraint on the type A. Namely, the type A should be an "instance" of $Ord\ [A]$. In the background the Scala compiler rewrites definitions with context bounds into definitions with implicit arguments. In the body of *cmp* an additional mechanism, called an implicit *query*, is now necessary to query the environment for a value of type $Ord\ [A]$. This query mechanism in Scala is nothing more than a simple function taking an implicit argument. The (slightly simplified) definition of the query operator is:[3]

$$\textbf{def}\ ?\ [T]\ (\textbf{implicit}\ w : T) : T = w$$

The operator ? takes a single *implicit* argument of type T and returns that value. Hence operationally it is just the identity function. The key point is that, when used, the implicit argument is filled in automatically by the compiler. In the definition of *cmp*, the expression $?[Ord\ [A]]$ triggers the compiler to look for a value of type $Ord\ [A]$ in the *implicit environment*. The implicit environment collects values that are declared to be implicit, and usable for automatic implicit resolution.

Implicit values, which correspond to type class instances in Haskell, are declared by using the **implicit** keyword. The following three examples capture the "instances" for *Ord*:

$$\begin{array}{l}
\textbf{implicit val}\ OrdInt = \textbf{new}\ Ord\ [Int]\ \{ \\
\quad \textbf{def}\ le\ (x : Int, y : Int) = primIntLe\ (x, y) \\
\} \\
\textbf{implicit val}\ OrdChar = \textbf{new}\ Ord\ [Char]\ \{ \\
\quad \textbf{def}\ le\ (x : Char, y : Char) = primCharLe\ (x, y) \\
\} \\
\textbf{implicit def}\ OrdPair\ [A : Ord, B : Ord] = \textbf{new}\ Ord\ [(A, B)]\ \{ \\
\quad \textbf{def}\ le\ (x : (A, B), y : (A, B)) = \\
\qquad cmp\ (fst\ (x), fst\ (y)) \wedge (!(cmp\ (fst\ (y), fst\ (x))) \vee cmp\ (snd\ (x), snd\ (y))) \\
\}
\end{array}$$

With those definitions it is now possible to declare functions, such as *sort*, that require *Ord* instances:

$$\textbf{def}\ sort\ [A : Ord]\ (x : List\ [A]) = ...$$

This Scala *sort* function can be used in a similar way to the corresponding Haskell function. For example the call $sort\ (List\ ((3, \texttt{'a'}), (2, \texttt{'c'}), (3, \texttt{'b'})))$ is valid and does not require an explicit argument of type $Ord\ [(Int, Char)]$. Instead this argument is computed from the implicit definitions for *Ord*.

[3] In Scala the operator is known by the longer name `implicitly`.

Wider Range of Applications for Implicits Scala implicits do allow for a wider range of applications than type classes. One example where implicits naturally address a problem that type classes do not address well is the problem of *implicit configurations* (Kiselyov & Shan, 2004). The following example, adapted from Kiselyov and Shan, illustrates this:

def *add* $(x : Int, y : Int)$ $(\textbf{implicit}\ modulus : Int) = (x + y)\ \%\ modulus$
def *mul* $(x : Int, y : Int)$ $(\textbf{implicit}\ modulus : Int) = (x * y)\ \%\ modulus$
implicit val *defMod* $: Int = 4$
def *test* $=$ *add* $(mul\ (3,3), mul\ (5,5))$ // *returns* 2

Here the idea is to model *modular arithmetic*, where numbers that differ by multiples of a given modulus are treated as identical. For example 3 * 3 = 1 (mod 4) because 9 and 1 differ by a multiple of 4. The code shows the definition of addition and multiplication in modular arithmetic, where in Scala % is modulo division. Both addition and multiplication include a third (implicit) parameter, which is the modulus of the division. Although the modulus could be passed explicitly this would be extremely cumbersome. Instead it is desirable that the modulus is passed implicitly. Scala implicits allow this, by simply marking the *modulus* parameter in *add* and *mul* with the **implicit** keyword. The third line shows how to set up an implicit value for the modulus. Adding **implicit** before **val** signals that the value being defined is available for synthesising values of type *Int*. Finally, *test* illustrates how expressions doing modular arithmetic can be defined using the implicit modulus. Because Scala also has local scoping, different modulus values can be used in different scopes.

Several other examples of applications that can be covered by implicits, but are harder to achieve with type classes are found in the existing literature (Oliveira *et al.*, 2010; Oliveira *et al.*, 2012; Odersky *et al.*, 2017). In particular, in a recent paper Odersky et al. (2017) introduce *implicit function types*, which are a generalization of the original Scala implicits (Oliveira *et al.*, 2010), and demonstrate several interesting use cases for implicits.

Instability in Scala Although Scala allows *nested* local scoping and overlapping implicits, *stability* is not guaranteed. Figure 1 illustrates the issue briefly, based on the example from Section 2.2. Line (1) defines a function *id* with type parameter α, which is simply the identity function of type $\alpha \Rightarrow \alpha$[4]. The **implicit** keyword in the declaration specifies that this value may be used to synthesise an implicit argument. Line (2) defines a function *trans* with type parameter α, which takes an implicit argument f of type $\alpha \Rightarrow \alpha$ and returns $f(x)$. Here the **implicit** keyword specifies that the actual argument should not be given explicitly; instead argument of the appropriate type will be synthesised from the available **implicit** declarations.

In the nested scope, line (3) defines function *succ* of type $Int \Rightarrow Int$ that takes argument x and returns $x + 1$. Again, the **implicit** keyword in the declaration specifies that *succ* may be used to synthesise implicit arguments. Line (4) defines a function *bad* with type parameter α which takes an argument x of type α and returns the value of function *trans* applied at type α to argument x. Line (5) shows that, as in the earlier example and for the same reason, *bad* (3) returns 3. As with the Haskell example, accepting this definition is

[4] Note that the $\alpha \Rightarrow \beta$ notation in Scala represents a function type, rather than a rule type.

```
trait A {
  implicit def id [α] : α ⇒ α = x ⇒ x                    // (1)
  def trans [α] (x : α) (implicit f : α ⇒ α) = f (x)     // (2)
}
object B extends A {
  implicit def succ : Int ⇒ Int = x ⇒ x + 1              // (3)
  def bad [α] (x : α) : α = trans [α] (x)                // (4) unstable definition !
  val v1 = bad [Int] (3)                                 // (5) evaluates to 3
  // val v2 = trans [Int] (3)                            // (6) substituting bad by trans is rejected
}
```

Fig. 1. Nested Scoping with Overlapping Rules in Scala

an equally nasty impediment to equational reasoning, since performing simple equational reasoning would lead to a different result. However unlike in Haskell, it is the intended behaviour: it is enabled by default and cannot be disabled. Interestingly the expression in line (6), which is accepted in Haskell, is actually rejected in Scala. Note that the expression in line (6) is simply the unfolding of the expression in line (5) (which is accepted in Scala). In the expression in line (6) the Scala compiler does detect two possible instances for $Int \Rightarrow Int$, but does not select the most specific one. In this case the call in line (6) is considered ambiguous because Scala accounts for other factors, when deciding whether there is ambiguity (Oliveira *et al.*, 2010; Odersky *et al.*, 2017). Rejecting line (6) has another unfortunate effect. Not only is the semantics not preserved under unfolding, but typing is not either: i.e. going from line (5) to line (6) using a simple unfolding step makes the program ill-typed! Clearly, satisfying desirable properties such as stability and type preservation is a subtle matter in the presence of implicits and deserves careful study.

2.6 An Overview of COCHIS

Like Haskell, our calculus COCHIS guarantees stability and coherence and, like Scala, it permits nested/overlapping declarations, and does not guarantee global uniqueness. COCHIS improves upon the implicit calculus (Oliveira *et al.*, 2012) by having stability and a better, more expressive design for resolution. Like the implicit calculus, the primary goal of COCHIS is to model *implicit resolution* and the *scoping* of implicit values used by resolution. Next we iterate over the key constructs and features of COCHIS.

Fetching Values by Type A central construct in COCHIS is a query. Queries allow values to be fetched by type, not by name. For example, in the following function call

foo ?*Int*

the query ?*Int* looks up a value of type *Int* in the implicit environment, to serve as an actual argument. Note that queries in COCHIS play the same role as the operator ? in Scala.

Constructing Values with Type-Directed Rules COCHIS constructs values, using programmer-defined, type-directed rules (similar to functions). A rule (or rule abstraction) defines how to compute, from implicit arguments, a value of a particular type. For example, here is a rule that given an implicit *Int* value, adds one to that value:

$\lambda_? Int.?Int + 1$

The rule abstraction syntax resembles a traditional λ expression. However, instead of having a variable as argument, a rule abstraction ($\lambda_?$) has a type as argument. The type argument denotes the availability of a value of that type (in this case *Int*) in the implicit environment inside the body of the rule abstraction. Thus, queries over the rule abstraction type argument inside the rule body will succeed.

The type of the rule above is the *rule type* $Int \Rightarrow Int$. This type denotes that the rule has type *Int* provided a value of type *Int* is available in the implicit environment. The implicit environment is extended through rule application (analogous to extending the environment with function applications). Rule application is expressed as, for example:

$(\lambda_? Int.?Int + 1)$ **with** 1

With syntactic sugar similar to a **let**-expression, a rule abstraction-application combination is more compactly denoted as:

implicit 1 **in** $(?Int + 1)$

Both expressions return 2.

The analog to rule abstractions in Scala are functions with arguments marked with the **implicit** keyword. However, in older versions of Scala, functions with implicit arguments were not first class and could not be abstracted over. In particular in older versions of Scala it was impossible to express the type of a function with an implicit argument. Recent versions of Scala, partly inspired by the implicit calculus, generalize the mechanism of implicits and make rule abstractions and types first class too, by what they call *implicit function types* (Odersky *et al.*, 2017).

Higher-Order Rules COCHIS supports higher-order rules. For example, the rule

$\lambda_? Int.\lambda_?(Int \Rightarrow Int \times Int).?(Int \times Int)$

when applied, will compute an integer pair given an integer and a rule to compute an integer pair from an integer. This rule is higher-order because another rule (of type $Int \Rightarrow Int \times Int$) is used as an argument. The following expression returns $(3,4)$:

$(\lambda_? Int.\lambda_?(Int \Rightarrow Int \times Int).?(Int \times Int))$
with 3
with $(\lambda_? Int.(?Int, ?Int + 1))$

Recursive Resolution Note that resolving the query $?(Int \times Int)$ above involves multiple implicits. The current environment does not contain the required integer pair. It does however contain the integer 3 and a rule $\lambda_? Int.(?Int, ?Int + 1)$ to compute a pair from an integer. Hence, the query is resolved with $(3,4)$, the result of applying the pair-producing rule to 3.

Polymorphic Implicits and Queries COCHIS features explicit polymorphism. For example, the rule $\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))$ abstracts over a type using standard type abstraction and then uses a rule abstraction to provide a value of type α in the implicit environment of the

rule body. This rule has type $\forall\alpha.\alpha \Rightarrow \alpha\times\alpha$ and can be instantiated to multiple rules of monomorphic types $Int \Rightarrow Int\times Int, Bool \Rightarrow Bool\times Bool, \ldots$.

Multiple monomorphic queries can be resolved by the same polymorphic rule. The following expression returns $((3,3),(True,True))$:

$$\textbf{implicit}\ 3\ \textbf{in implicit}\ True\ \textbf{in implicit}\ (\Lambda\alpha.(\lambda_?\alpha.(?\alpha,?\alpha)))\ \textbf{in}\ (?(Int\times Int),?(Bool\times Bool))$$

Queries can be polymorphic too. For instance, the following example extracts the polymorphic implicit with a polymorphic query.

$$\textbf{implicit}\ (\Lambda\alpha.(\lambda_?\alpha.(?\alpha,?\alpha)))\ \textbf{in}\ ?(\forall\beta.\beta\Rightarrow(\beta\times\beta)))$$

In practice, polymorphic queries are useful in combination with higher-kinded types where they occur as recursive resolvents of polymorphic rules. We cannot illustrate this with COCHIS as, to keep its definition small, it is not equipped with higher-kinded types. The interested reader can find examples in the work of Bottu et al. (2017).

Type-Directed Synthesis of Simple Programs One interesting feature of COCHIS is that it can synthesize simple programs of a given type. This feature can potentialy be useful when there is at most one possible implementation of a program. For example a variant of the polymorphic identity function could be synthesized with the following query:

$$?(\forall\alpha.\alpha\Rightarrow\alpha)$$

We have not yet explored the usefulness of this feature for practical programming in depth, but this seems to be related to programming with typed holes (Norell, 2008). Perhaps such a feature (or some extension of it) can be useful to automatically synthesize implementations for typed holes.

Combining Higher-Order and Polymorphic Rules The rule:

$$\lambda_?Int.\lambda_?(\forall\alpha.\alpha\Rightarrow\alpha\times\alpha).(?((Int\times Int)\times(Int\times Int)))$$

prescribes how to build a pair of integer pairs, inductively from an integer value, by consecutively applying the rule of type $\forall\alpha.\alpha\Rightarrow\alpha\times\alpha$ twice: first to an integer, and again to the result (an integer pair). For example, the following expression returns $((3,3),(3,3))$:

$$\textbf{implicit}\ 3\ \textbf{in implicit}\ (\Lambda\alpha.(\lambda_?\alpha.(?\alpha,?\alpha)))\ \textbf{in}\ ?((Int\times Int)\times(Int\times Int))$$

Locally and Lexically Scoped Implicits Implicits can be nested and resolution respects their lexical scope. Consider the following program:

$$\begin{array}{l}\textbf{implicit}\ 1\ \textbf{in}\\ \quad\textbf{implicit}\ True\ \textbf{in}\\ \quad\quad\textbf{implicit}\ (\lambda_?Bool.\ \textbf{if}\ ?Bool\ \textbf{then}\ 2\ \textbf{else}\ 0)\ \textbf{in}\\ \quad\quad\quad ?Int\end{array}$$

The query $?Int$ is not resolved with the integer value 1. Instead the rule that returns an integer from a boolean is applied to the boolean *True*, because that rule can provide an integer value and it is nearer to the query. So, the program returns 2 and not 1.

Care with Reduction Observe that some care is required with the inlining of **let**-bindings and other refactorings that perform variable substitutions. Consider for instance the following program.

implicit 1 **in**
  let $x = ?Int$ **in**
    implicit 2 **in**
      x

One might inline x and obtain the following.

implicit 1 **in**
  implicit 2 **in**
    $?Int$

But this would change its meaning: the first program returns 1, while the second returns 2. The solution is to never consider a term as a candidate for substitution until all of its implicits have been resolved.

This situation is not much different from that of regular **let**-bindings. For example, if we naively inline y in the following program:

let $x = 1$ **in**
  let $y = x$ **in**
    let $x = 2$ **in**
      y

we get

let $x = 1$ **in**
  let $x = 2$ **in**
    x

The second program clearly has a different meaning, since the variable x is now captured by the second binder rather than the first. While this problem can be remedied by renaming the second binder's variable, there is no analogous solution for implicits.

2.6.1 Encoding Simple Type Classes in COCHIS

A simple form of type classes can be encoded in COCHIS similarly to how type classes can be encoded in Scala. In this section we briefly (and informally) illustrate the encoding using examples. The simple encoding presented here does not deal with *superclasses*. We discuss superclasses in Section 6.

Next we illustrate how the encoding works on the examples from Section 2.1. To help with readability we assume a few convenient source language features not available in COCHIS (which is designed as a formal core calculus rather than a full-fledged source language). In particular COCHIS has no type-inference and requires explicit rule applications and queries. The design of a source language that supports type-inference, implicit rule applications, implicit polymorphism and that translates into a COCHIS-like calculus was

already explored in our previous work on the implicit calculus (Oliveira *et al.*, 2012). To better illustrate some of our examples here we will assume such source language features.

Firstly we use type synonyms to allow us to give a short name to a type. Secondly we use records. Using both of those constructs, the three type classes introduced in Sections 2.1 and 2.2 can be declared as:

type $Ord\ \alpha = \{le : \alpha \to \alpha \to Bool\}$
type $Show\ \alpha = \{show : \alpha \to String\}$
type $Read\ \alpha = \{read : String \to \alpha\}$

Similarly to the Scala encoding we define a *cmp* function that makes the argument of type $Ord\ \alpha$ implicit:

let $cmp : \forall\alpha.Ord\ \alpha \Rightarrow \alpha \to \alpha \to Bool = (?:Ord\ \alpha).le$ **in**

Here the query is annotated with a type $Ord\ \alpha$ triggering the resolution of a value of that type. Once that value is computed, the field *le* can be extracted from it.

The "instances" of *Ord* can be defined as record values or rule types returning an *Ord* record.

let $ordInt : Ord\ Int = \{le = \lambda x.\ \lambda y.\ primIntLe\ x\ y\}$ **in**
let $ordChar : Ord\ Char = \{le = \lambda x.\ \lambda y.\ primCharLe\ x\ y\}$ **in**
let $ordPair : \forall\alpha\ \beta.\ Ord\ \alpha \Rightarrow Ord\ \beta \Rightarrow Ord\ (\alpha,\beta) = \{le = \lambda x.\ \lambda y.$
$cmp\ (fst\ x)\ (fst\ y) \wedge ((\neg\ (cmp\ (fst\ y)\ (fst\ x))) \vee cmp\ (snd\ x)\ (snd\ y))\}$ **in**

Here the first two values denote instances for the base types *Int* and *Char*. The instance for pairs (*ordPair*) has two constraints ($Ord\ \alpha$ and $Ord\ \beta$), and those constraints are implicitly used by *cmp*.

Given a *sort* function:

let $sort : \forall\alpha.Ord\ \alpha \Rightarrow List\ \alpha \to List\ \alpha = ...$

we can now use **implicit** to introduce the "instances" into the implicit scope and have the $Ord\ (List\ Int)$ argument of the call $sort\ [(3,\texttt{'a'}),(2,\texttt{'c'}),(3,\texttt{'b'})]$ automatically inferred:

implicit *ordInt* **in**
  implicit *ordChar* **in**
    implicit *ordPair* **in**
      $sort\ [(3,\texttt{'a'}),(2,\texttt{'c'}),(3,\texttt{'b'})]$

2.7 Overlapping Implicits and Stability in COCHIS

As previously shown, the lexical scope imposes a natural precedence on implicits that ensures coherence. This precedence means that the lexically nearest implicit is used to resolve a query, and not necessarily the most specific implicit. For instance, the following COCHIS variation on the running *trans* example from Section 2.2

implicit $(\lambda n.n+1 : Int \to Int)$ **in**
  implicit $(\lambda x.x : \forall\alpha.\alpha \to \alpha)$ **in**
    $?(Int \to Int)\ 3$

yields the result 3 as the inner identity implicit has precedence over the more specific incrementation implicit in the outer scope. Yet, it is not always possible to statically select the nearest matching implicit. Consider the program fragment

$$\begin{array}{l}\textbf{let}\ \mathit{bad} : \forall\beta.\beta \to \beta = \\ \quad \textbf{implicit}\ (\lambda x.x : \forall\alpha.\alpha \to \alpha)\ \textbf{in} \\ \quad\quad \textbf{implicit}\ (\lambda n.n+1 : \mathit{Int} \to \mathit{Int})\ \textbf{in} \\ \quad\quad\quad ?(\beta \to \beta)\end{array}$$

Here we cannot statically decide whether $\mathit{Int} \to \mathit{Int}$ matches $\beta \to \beta$: it depends on whether β is instantiated to *Int* or not.

One might consider to force the matter by picking the lexically nearest implicit that matches all possible instantiations of the query, e.g., $\forall\alpha.\alpha \to \alpha$ in the example. While this poses no threat to type safety, this approach is nevertheless undesirable for two reasons. Firstly, it makes the behaviour of programs harder to predict, and, secondly, the behaviour of programs is not *stable under inlining*. Consider the call *bad Int* 3, which would yield the result 3. If instead we inline the function definition of *bad* at the call site and substitute the type argument, we obtain the specialised program

$$\begin{array}{l}\textbf{implicit}\ (\lambda x.x : \forall\alpha.\alpha \to \alpha)\ \textbf{in} \\ \quad \textbf{implicit}\ (\lambda n.n+1 : \mathit{Int} \to \mathit{Int})\ \textbf{in} \\ \quad\quad ?(\mathit{Int} \to \mathit{Int})\ 3\end{array}$$

Now $\mathit{Int} \to \mathit{Int}$ is the nearest lexical match and the program yields the result 4. Consequently, inlining definitions changes their behavior. To avoid this unpredictable behaviour, COCHIS rejects such unstable matchings. Technically speaking the key property that COCHIS guarantees is *stability of resolution* (see also Section 5). Essentially this property ensures that type instantiation does not affect the resolution of queries. That is, if some type variables appear free in a query that resolves, then, after instantiating any or all of those type variables, the query still resolves in the same way, i.e., using the same implicits. If it cannot be statically guaranteed that resolution behaves in the same way for *every* instantiation, then the program is rejected. The benefit of rejecting such potentially unstable programs is that the principle of substituting equals for equals is not affected by the interaction between resolution and instantiation. This makes reasoning about programs and code refactoring more predictable.

3 The COCHIS Calculus

This section formalizes the syntax and type system of COCHIS, while Section 5 formalises the type-directed translation to System F. To avoid duplication and ease reading, we present the type system and type-directed translation together, using grey boxes to indicate which parts of the rules belong to the type-directed translation. These greyed parts can be ignored in this section and will be explained in the next.

3.1 Syntax

Here is the syntax of the calculus:

$$\begin{array}{lllll} \text{Types} & \rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha.\rho \mid \rho_1 \Rightarrow \rho_2 \\ \text{Monotypes} & \sigma & ::= & \alpha \mid \sigma \to \sigma \\ \text{Expressions} & e & ::= & x \mid \lambda(x:\rho).e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e\,\sigma \mid ?\rho \mid \lambda_?\rho.e \mid e_1 \textbf{ with } e_2 \end{array}$$

Types ρ comprise four constructs: type variables α; function types $\rho_1 \to \rho_2$; universal types $\forall\alpha.\rho$; and the novel *rule* types $\rho_1 \Rightarrow \rho_2$. In a rule type $\rho_1 \Rightarrow \rho_2$, type ρ_1 is called the *context* and type ρ_2 the *head*, following Haskell's terminology for type class instances. Observe that COCHIS types are similar to Haskell's type schemes with the exception that contexts are types rather than type class constraints. We also define a subset of types, called *monotypes* σ, which, like Haskell's monotypes, do not contain universal quantifiers or rule types.

Expressions e include three abstraction-elimination pairs. The form $\lambda(x:\rho).e$ abstracts over a value of type ρ in expression e, which can refer to it with variable x; the abstraction is eliminated by application $e_1\,e_2$. Similarly, $\Lambda\alpha.e$ abstracts over a type in expression e, which can refer to it with type variable α; the abstraction is eliminated by *predicative* type application $e\,\sigma$ (see Section 3.5.1). Finally, $\lambda_?\rho.e$ abstracts over implicit values of type ρ in expression e, which can refer to it with implicit query $?\rho$; the abstraction is eliminated by implicit application e_1 **with** e_2. For convenience we adopt the Barendregt convention (Barendregt, 1981), that variables in binders are distinct, throughout this article.

Using implicit abstraction and implicit application we can build the **implicit** sugar used in Section 2.

$$\textbf{implicit } e_1 : \rho \textbf{ in } e_2 \stackrel{\text{def}}{=} (\lambda_?\rho.e_2) \textbf{ with } e_1$$

For brevity we have kept the COCHIS calculus small. Examples may use additional syntax such as built-in integers, integer operators, and boolean literals and types.

3.2 Type System

Figure 2 presents the static type system of COCHIS. Our language is based on predicative System F, which is included in our system.

As in predicative System F, a type environment Γ records type variables α and variables x with associated types ρ that are in scope. New here is that it also records instances of implicits $?\rho$.

$$\text{Type Environments} \quad \Gamma \quad ::= \quad \varepsilon \mid \Gamma, x:\rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x$$

A typing judgment $\Gamma \vdash e : \rho$ holds if expression e has type ρ with respect to type environment Γ. The first five rules copy the corresponding predicative System F rules; only the last three deserve special attention. Firstly, rule (TY-IABS) extends the implicit environment with the type of an implicit instance. The side condition $\vdash_{\text{unamb}} \rho_1$ states that the type ρ_1 must be unambiguous; we explain this concept in Section 3.5. Secondly, rule (TY-IAPP) eliminates an implicit abstraction by supplying an instance of the required type. Finally, rule (TY-QUERY) resolves a given type ρ against the implicit environment. Again, a side-

$$\boxed{\Gamma \vdash e : \rho \rightsquigarrow E}$$

$$\text{(Ty-Var)}\ \frac{(x:\rho) \in \Gamma}{\Gamma \vdash x : \rho \rightsquigarrow x}$$

$$\text{(Ty-Abs)}\ \frac{\Gamma, x:\rho_1 \vdash e : \rho_2 \rightsquigarrow E}{\Gamma \vdash \lambda x:\rho_1.e : \rho_1 \to \rho_2 \rightsquigarrow \lambda x:|\rho_1|.E}$$

$$\text{(Ty-App)}\ \frac{\Gamma \vdash e_1 : \rho_1 \to \rho_2 \rightsquigarrow E_1 \qquad \Gamma \vdash e_2 : \rho_1 \rightsquigarrow E_2}{\Gamma \vdash e_1\, e_2 : \rho_2 \rightsquigarrow E_1\, E_2}$$

$$\text{(Ty-TAbs)}\ \frac{\Gamma, \alpha \vdash e : \rho \rightsquigarrow E_1}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E_1}$$

$$\text{(Ty-TApp)}\ \frac{\Gamma \vdash e : \forall\alpha.\rho \rightsquigarrow E}{\Gamma \vdash e\,\sigma : \rho[\sigma/\alpha] \rightsquigarrow E\ |\sigma|}$$

$$\text{(Ty-IAbs)}\ \frac{\Gamma, ?\rho_1 \rightsquigarrow x \vdash e : \rho_2 \rightsquigarrow E \quad \vdash_{\text{unamb}} \rho_1 \quad x\ \textit{fresh}}{\Gamma \vdash \lambda_?\rho_1.e : \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x:|\rho_1|.E}$$

$$\text{(Ty-IApp)}\ \frac{\Gamma \vdash e_1 : \rho_2 \Rightarrow \rho_1 \rightsquigarrow E_1 \qquad \Gamma \vdash e_2 : \rho_2 \rightsquigarrow E_2}{\Gamma \vdash e_1\ \textbf{with}\ e_2 : \rho_1 \rightsquigarrow E_1\ E_2}$$

$$\text{(Ty-Query)}\ \frac{\Gamma \vdash^{\text{a}}_{\text{r}} \rho \rightsquigarrow E \qquad \vdash_{\text{unamb}} \rho}{\Gamma \vdash ?\rho : \rho \rightsquigarrow E}$$

Fig. 2. Type System and Type-directed Translation to System F

condition states that ρ must be unambiguous. Resolution is defined in terms of the auxiliary judgment $\Gamma \vdash^{\text{a}}_{\text{r}} \rho$, which is explained next.

3.3 Resolution

Figure 3 provides a first, ambiguous definition of the resolution judgment. Its underlying principle is resolution in logic. Intuitively, $\Gamma \vdash^{\text{a}}_{\text{r}} \rho$ holds if Γ entails ρ, where the types in Γ and ρ are read as propositions, r stands for resolution and a for ambiguous. Following the "Propositions as Types" correspondence (Wadler, 2015), we read α as a propositional variable and $\forall\alpha.\rho$ as universal quantification. Yet, unlike in the traditional interpretation of types as propositions, we have two forms of arrows, function types $\rho_1 \to \rho_2$ and rule types $\rho_1 \Rightarrow \rho_2$, and the twist is that we choose to treat only rule types as implications, leaving function types as uninterpreted predicates.

Unfortunately, the definition in Figure 3 suffers from two problems. Firstly, it is *not syntax-directed*; several of the inference rules have overlapping conclusions. Hence, a

$$\boxed{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow E}$$

$$\text{(AR-IVAR)}\ \frac{?\rho \rightsquigarrow x \in \Gamma}{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow x}$$

$$\text{(AR-TABS)}\ \frac{\Gamma, \alpha \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow E}{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E} \qquad \text{(AR-IABS)}\ \frac{\Gamma, ?\rho_1 \rightsquigarrow x \vdash_{\mathrm{r}}^{\mathrm{a}} \rho_2 \rightsquigarrow E \qquad x\ \mathit{fresh}}{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E}$$

$$\text{(AR-TAPP)}\ \frac{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \forall\alpha.\rho \rightsquigarrow E}{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho[\sigma/\alpha] \rightsquigarrow E\ |\sigma|} \qquad \text{(AR-IAPP)}\ \frac{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho_1 \Rightarrow \rho_2 \rightsquigarrow E_2 \qquad \Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho_1 \rightsquigarrow E_1}{\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho_2 \rightsquigarrow E_2\ E_1}$$

Fig. 3. Ambiguous Resolution

deterministic resolution algorithm is non-obvious. Secondly and more importantly, the definition is *ambiguous*: a type can be derived in multiple different ways. As an example of both issues, consider that under the environment

$$\Gamma_0 = ?\mathit{Int}, ?\mathit{Bool}, ?(\mathit{Bool} \Rightarrow \mathit{Int})$$

there are two different derivations for resolving $\Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Int}$:

$$\text{(AR-IVAR)}\ \frac{?\mathit{Int} \in \Gamma_0}{\Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Int}}$$

and

$$\begin{array}{l}\text{(AR-IVAR)}\ \dfrac{?(\mathit{Bool} \Rightarrow \mathit{Int}) \in \Gamma_0}{\Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Bool} \Rightarrow \mathit{Int}} \qquad \text{(AR-IVAR)}\ \dfrac{?\mathit{Bool} \in \Gamma_0}{\Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Bool}} \\ \text{(AR-IAPP)}\ \hline \Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Int}\end{array}$$

This example illustrates the first issue; in particular the inference rules (AR-IVAR) and (AR-IAPP) overlap as both can be used to conclude $\Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Int}$. It also shows the second issue as there are two fully different derivation trees for $\Gamma_0 \vdash_{\mathrm{r}}^{\mathrm{a}} \mathit{Int}$. While this may seem harmless at the type-level, at the value-level each derivation corresponds to a (possibly) different value. Hence, ambiguous resolution may render the meaning of a program ambiguous. In other words, if both resolutions are allowed then the semantics is not coherent.

We next address these two issues one by one. Readers who are keen to see the end result may wish to skip the gradual developments and jump straight to Section 3.5.5.

3.4 Type-Directed Resolution with Focusing

To obtain a type-directed formulation of resolution, we adopt a solution from proof search known as *focusing* (Andreoli, 1992). This solution makes sure that only one inference rule applies at any given point and thereby rules out gratuitous forms of nondeterminism.

$$\Sigma ::= \varepsilon \mid \rho \rightsquigarrow x, \Sigma$$

$$\boxed{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E} \qquad \textit{Focusing}$$

$$\text{(FR-TABS)}\ \frac{\Gamma, \alpha \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\forall \alpha.\rho] \rightsquigarrow \Lambda\alpha.E} \qquad \text{(FR-IABS)}\ \frac{\Gamma, ?\rho_1 \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho_2] \rightsquigarrow E \qquad x \textit{ fresh}}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho_1 \Rightarrow \rho_2] \rightsquigarrow \lambda x : |\rho_1|.E}$$

$$\text{(FR-SIMP)}\ \frac{?\rho \rightsquigarrow x \in \Gamma \qquad \Gamma; [\rho] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \bar{\rho}' \rightsquigarrow \bar{x}'; \tau \rightsquigarrow E \qquad \Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho'] \rightsquigarrow E' \ \ (\forall \rho' \in \bar{\rho}')}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\tau] \rightsquigarrow E[\bar{E}'/\bar{x}']}$$

$$\boxed{\Gamma; [\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'} \qquad \textit{Matching}$$

$$\text{(FM-TAPP)}\ \frac{\Gamma; [\rho[\sigma/\alpha]] \rightsquigarrow E\,|\sigma| \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}{\Gamma; [\forall \alpha.\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}$$

$$\text{(FM-IAPP)}\ \frac{\Gamma; [\rho_2] \rightsquigarrow E\,x \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E' \qquad x \text{ fresh}}{\Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \rho_1 \rightsquigarrow x, \Sigma; \tau \rightsquigarrow E'}$$

$$\text{(FM-SIMP)}\ \Gamma; [\tau] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \varepsilon; \tau \rightsquigarrow E$$

Fig. 4. Focusing Resolution

As an example of such gratuitous nondeterminism consider the following two ways of resolving α given $\Gamma = \alpha, ?\alpha \rightsquigarrow x$:

$$\text{(AR-IVAR)}\ \frac{?\alpha \rightsquigarrow x \in \Gamma}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow x}$$

versus

$$\text{(AR-IAPP)}\ \frac{\text{(AR-IABS)}\ \dfrac{\text{(AR-IVAR)}\ \dfrac{?\alpha \rightsquigarrow y \in (\Gamma, ?\alpha \rightsquigarrow y)}{\Gamma, ?\alpha \rightsquigarrow y \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow y}}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \Rightarrow \alpha \rightsquigarrow \lambda y.y} \qquad \text{(AR-IVAR)}\ \dfrac{?\alpha \rightsquigarrow x \in \Gamma}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow x}}{\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \alpha \rightsquigarrow (\lambda y.y)\,x}$$

While these are two different proofs, they use the information in the context Γ in essentially the same way. They are β-equivalent and infer System F expressions that are also semantically equivalent. Hence, unlike the nondeterminism in the previous example at the end of Section 3.3, where the context provides two ways of resolving the query, this form of nondeterminism serves no purpose. We will see that focusing provides a straitjacket that eliminates the gratuitous nondeterminism and allows only the first and more direct of these two proofs. In fact, it allows only derivations in β-reduced and η-expanded form.

The focusing approach refines the grammar of types to distinguish a special class of *simple* types:

$$\begin{array}{lrcl} \text{Context Types} & \rho & ::= & \forall\alpha.\rho \mid \rho_1 \Rightarrow \rho_2 \mid \tau \\ \text{Simple Types} & \tau & ::= & \alpha \mid \rho_1 \to \rho_2 \end{array}$$

Observe that simple types τ are those types that do not have corresponding pairs of introduction and elimination rules in the ambiguous resolution judgment.

The definition of resolution with focusing that uses this refined grammar is given in Figure 4. The main *focusing* judgment $\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E$ is defined with the help of the auxiliary *matching* judgment $\Gamma; [\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'$. Both definitions are syntax-directed on the type ρ enclosed in square brackets.

The focusing judgment $\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho] \rightsquigarrow E$ focuses on the type ρ that is to be resolved – we call this type the "goal". There are three rules, for the three possible syntactic forms of ρ. Rules (FR-IABS) and (FR-TABS) decompose the goal by applying implication and quantifier introductions respectively. Once the goal is stripped down to a simple type τ, Rule (FR-SIMP) selects an implicit ρ from the environment Γ to discharge it. The selected type must *match* the goal, a notion that is captured by the auxiliary judgment. Matching gives rise to a sequence Σ of new (and hopefully simpler) goals that are resolved recursively.

The matching judgment $\Gamma; [\rho] \rightsquigarrow E \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'$ focuses on the selected implicit ρ and checks whether it matches the simple goal τ; informally it captures that ρ can be instantiated to $\Sigma \Rightarrow \tau$. Again, there are three rules for the three possible forms the rule can take. Rule (FM-TAPP) handles universal quantification by instantiating the quantified variable α in a way that recursively yields a match. Rule (FM-IAPP) handles a rule type $\rho_1 \Rightarrow \rho_2$ by recursively checking whether ρ_2 matches the goal. At the same time it yields a new goal ρ_1 which needs to be resolved in order for the rule to apply. Finally, rule (FM-SIMP) expresses the base case where the axiom is identical to the goal and there are no new goals.

This type-directed formulation of entailment reduces the number of proofs for a given goal. For instance, for the example above there is only one proof:

$$\text{(FR-SIMP)}\ \frac{?\alpha \rightsquigarrow x \in \Gamma \qquad \text{(FM-SIMP)}\ \Gamma; [\alpha] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \varepsilon; \alpha \rightsquigarrow x}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\alpha] \rightsquigarrow x}$$

3.5 Deterministic and Stable Resolution

While focusing provides a syntax-directed definition of resolution, it does not make resolution entirely deterministic. There are still two sources of nondeterminism: 1) the *ambiguous* instantiation of type variable α with a monotype σ in rule (FM-TAPP), and 2) nondeterministic selection of an implicit ρ from the type environment Γ in rule (FR-SIMP). This section eradicates those two remaining sources of nondeterminism to obtain an entirely deterministic formulation of resolution. On top of that, it imposes an additional *stability* condition to make resolution "super"-deterministic: resolution is preserved under type substitution. First, though, we point out that our choice for predicative instantiation has pre-emptively avoided a further source of nondeterminism.

3.5.1 Predicative Instantiation

We have restricted COCHIS to predicative instantiation, i.e., type variables can only be instantiated with monotypes σ. Impredicativity is a known source of nondeterminism in other settings like type inference for the polymorphic λ-calculus (Boehm, 1985; Pfenning, 1993). It causes similar problems for COCHIS, in the rules (AR-TAPP) and (FM-TAPP) for ambiguous and focusing resolution that choose an instantiation of a type variable.

To see why the impredicative instantiation in those rules causes nondeterminism, consider two ways resolving $\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \Rightarrow \mathit{Int}$ against the environment $\Gamma_1 = ?(\forall\alpha.\alpha \Rightarrow \alpha)$:[5]

$$\text{(AR-TAPP)}\ \dfrac{\text{(AR-IVAR)}\ \dfrac{?(\forall\alpha.\alpha \Rightarrow \alpha) \in \Gamma_1}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.\alpha \Rightarrow \alpha}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \Rightarrow \mathit{Int}}$$

and

$$\text{(AR-TAPP)}\ \dfrac{\text{(AR-IAPP)}\ \dfrac{\text{(AR-TAPP)}\ \dfrac{\text{(AR-IVAR)}\ \dfrac{?(\forall\alpha.\alpha \Rightarrow \alpha) \in \Gamma_1}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.\alpha \Rightarrow \alpha}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} (\forall\beta.\beta \Rightarrow \beta) \Rightarrow (\forall\beta.\beta \Rightarrow \beta)} \quad \text{(AR-IVAR)}\ \dfrac{?(\forall\beta.\beta \Rightarrow \beta) \in \Gamma_1}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\beta.\beta \Rightarrow \beta}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\beta.\beta \Rightarrow \beta}}{\Gamma_1 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \Rightarrow \mathit{Int}}$$

The first proof only involves the instantiation of α with *Int*. Yet, the second proof contains an impredicative instantiation of α with $\forall\beta.\beta \Rightarrow \beta$.

We have adopted the standard solution from the outset, which only allows predicative instantiation and thus only accepts the first of the two derivations above.

Observe that we not only forbid instantiation with universally quantified types $\forall\alpha.\rho$, but also with rule types $\rho_1 \Rightarrow \rho_2$. The latter are also a source of ambiguity. Consider for instance resolving *Int* in the environment $\Gamma_2 = ?(\forall\alpha.(\alpha \to \alpha) \Rightarrow \alpha), ?\mathit{Bool}, ?((\mathit{Bool} \Rightarrow \mathit{Int}) \to (\mathit{Bool} \to \mathit{Int})), ?(\mathit{Int} \to \mathit{Int})$. There is one derivation that involves instantiating the first entry's α with a monotype, namely with *Int*:

$$\text{(AR-IAPP)}\ \dfrac{\text{(AR-TAPP)}\ \dfrac{\text{(AR-IVAR)}\ \dfrac{?(\forall\alpha.(\alpha \to \alpha) \Rightarrow \alpha) \in \Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \forall\alpha.(\alpha \to \alpha) \Rightarrow \alpha}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} (\mathit{Int} \to \mathit{Int}) \Rightarrow \mathit{Int}} \quad \text{(AR-IVAR)}\ \dfrac{?(\mathit{Int} \to \mathit{Int}) \in \Gamma_2}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int} \to \mathit{Int}}}{\Gamma_2 \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}}$$

However, instantiation with the non-monotype $\mathit{Bool} \Rightarrow \mathit{Int}$ also yields a derivation; for the sake of conciseness, we have abbreviated *Bool* and *Int* to *B* and *I* respectively.

[5] For the sake of compactness the example uses the ambiguous definition of resolution. Similarly problematic examples can be created for the focusing-based definition.

$$\dfrac{\dfrac{\dfrac{\dfrac{?(\forall\alpha.(\alpha\to\alpha)\Rightarrow\alpha)\in\Gamma_2}{\Gamma_2\vdash^{\mathrm{a}}_{\mathrm{r}}\forall\alpha.(\alpha\to\alpha)\Rightarrow\alpha}\,(\text{AR-IVAR}) \quad \dfrac{?((B\Rightarrow I)\to(B\Rightarrow I))\in\Gamma_2}{\Gamma_2\vdash^{\mathrm{a}}_{\mathrm{r}}(B\Rightarrow I)\to(B\Rightarrow I)}\,(\text{AR-IVAR})}{\Gamma_2\vdash^{\mathrm{a}}_{\mathrm{r}}((B\Rightarrow I)\to(B\Rightarrow I))\Rightarrow(B\Rightarrow I)}\,(\text{AR-TAPP}) \quad \dfrac{?B\in\Gamma_2}{\Gamma_2\vdash^{\mathrm{a}}_{\mathrm{r}}B}\,(\text{AR-IVAR})}{\Gamma_2\vdash^{\mathrm{a}}_{\mathrm{r}}B\Rightarrow I}\,(\text{AR-IAPP})}{\Gamma_2\vdash^{\mathrm{a}}_{\mathrm{r}}I}\,(\text{AR-IAPP})$$

By restricting ourselves to instantiation with monotypes, we disallow the second derivation and thus avoid this source of ambiguity.

3.5.2 Non-Ambiguity Constraints

Rule (FM-TAPP) does not explain how the predicative substitution $[\sigma/\alpha]$ for the type $\forall\alpha.\rho$ should be obtained. At first sight it seems that the choice of σ is free and thus a source of nondeterminism. However, in many cases the choice is not free at all, but is instead determined fully by the simple type τ that we want to match. However, the choice is not always forced by the matching. Take for instance the context type $\forall\alpha.(\alpha\to String)\Rightarrow(String\to\alpha)\Rightarrow(String\to String)$. This type encodes the Haskell type $\forall\alpha.(Show\ \alpha, Read\ \alpha)\Rightarrow String\to String$ of the ambiguous expression $read\circ show$ discussed in Section 2.2. The choice of α is ambiguous when matching against the simple type $String\to String$. Yet, the choice is critical for two reasons. Firstly, if we guess the wrong instantiation σ for α, then it may not be possible to recursively resolve $(String\to\alpha)[\sigma/\alpha]$ or $(\alpha\to String)[\sigma/\alpha]$, while with a lucky guess both can be resolved. Secondly, for different choices of σ, $(String\to\alpha)[\sigma/\alpha]$ and $(\alpha\to String)[\sigma/\alpha]$ can be resolved in completely different ways.

In order to avoid any problems, we conservatively forbid all ambiguous context types in the implicit environment with the $\vdash_{\text{unamb}}\rho_1$ side-condition in rule (TY-IABS) of Figure 2.[6] This judgment is defined in Figure 5 in terms of the auxiliary judgment $\bar{\alpha}\vdash_{\text{unamb}}\rho$ which takes an additional sequence of type variables α that is initially empty.

This auxiliary judgment expresses that all type variables $\bar{\alpha}$ are resolved when matching against ρ. Its base case, rule (UA-SIMP), expresses that fixing the simple type τ fixes its free type variables $\bar{\alpha}$. The inductive rule (UA-TABS) accumulates the bound type variables $\bar{\alpha}$ before the head. Rule (UA-IABS) skips over any contexts on the way to the head, but also recursively requires that these contexts are unambiguous. The latter is necessary because rule (FR-SIMP) resolves those contexts recursively when ρ matches the resolvent; as recursive resolvents they add their contexts to the implicit environment in rule (FR-IABS).

Finally, the unambiguity condition is imposed on the queried type ρ in rule (TY-QUERY) because this type too may extend the implicit environment in rule (FR-IABS).

Note that the definition rules out harmless ambiguity, such as that in the type $\forall\alpha.Int$. When we match the head of this type Int with the simple type Int, the matching succeeds without actually determining how the type variable α should be instantiated. Here the

[6] A more permissive design would allow quantified type variables that are not mentioned anywhere, such as α in $\forall\alpha.Int\Rightarrow Int$, and instantiate them to a dummy type, like GHC's `GHC.Prim.Any`, which is only used for this purpose. As such unused type variables serve little purpose, we have opted not to make an exception for them.

$$\boxed{\vdash_{\text{unamb}} \rho} \qquad \text{(UA-Main)}\ \frac{\varepsilon \vdash_{\text{unamb}} \rho}{\vdash_{\text{unamb}} \rho}$$

$$\boxed{\bar{\alpha} \vdash_{\text{unamb}} \rho} \qquad \text{(UA-Simp)}\ \frac{\bar{\alpha} \subseteq ftv(\tau)}{\bar{\alpha} \vdash_{\text{unamb}} \tau}$$

$$\text{(UA-TAbs)}\ \frac{\bar{\alpha}, \alpha \vdash_{\text{unamb}} \rho}{\bar{\alpha} \vdash_{\text{unamb}} \forall \alpha.\rho} \qquad \text{(UA-IAbs)}\ \frac{\vdash_{\text{unamb}} \rho_1 \qquad \bar{\alpha} \vdash_{\text{unamb}} \rho_2}{\bar{\alpha} \vdash_{\text{unamb}} \rho_1 \Rightarrow \rho_2}$$

Fig. 5. Unambiguous Context Types

ambiguity is harmless, because it does not affect the semantics. Yet, to keep the meta-theory simple, we have opted to not differentiate between harmless and harmful ambiguity.

3.5.3 Committed Choice

The other remaining source of nondeterminism is the nondeterministic choice $?\rho \in \Gamma$ that appears in rule (FR-Simp) of the focusing judgment. Consider the trivial example of resolving the goal *Int* against the environment $\Gamma = ?Int \rightsquigarrow x, ?Int \rightsquigarrow y$. Both implicits in the environment match the goal and yield different, i.e., incoherent, elaborations.

Our solution is to replace the nondeterministic relation $?\rho \in \Gamma$ by a deterministic one that selects the first matching implicit in the environment and commits to it. In fact, we encapsulate all three hypotheses of rule (FR-Simp) in a new *lookup* judgment $\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$ which resolves τ with the first matching implicit in the environment Γ' and performs any recursive resolutions against the environment Γ. Of course, the modified rule (FR-Simp') invokes this lookup judgment with two copies of the same environment, i.e., Γ and Γ' are identical.

$$\text{(FR-Simp')}\ \frac{\Gamma; [\Gamma] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\tau] \rightsquigarrow E}$$

The (still preliminary) definition of the judgment itself is syntax-directed with respect to the type environment Γ':

$$\boxed{\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E} \qquad \textit{Lookup}$$

$$\text{(DL-Match)}\ \frac{\Gamma; [\rho] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \overline{\rho' \rightsquigarrow x}; \tau \rightsquigarrow E \qquad \Gamma \vdash^{\mathrm{f}}_{\mathrm{r}} [\rho'] \rightsquigarrow E' \quad (\forall \rho' \in \overline{\rho}')}{\Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E[\bar{E}'/\bar{x}]}$$

$$\text{(DL-NoMatch)}\ \frac{\nexists E, \Sigma: \ \Gamma; [\rho] \rightsquigarrow x \vdash^{\mathrm{f}}_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E \qquad \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}{\Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}$$

$$\text{(DL-Var)}\ \frac{\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\Gamma; [\Gamma', x : \rho] \vdash_{\mathrm{r}} \tau \rightsquigarrow E} \qquad \text{(DL-TyVar)}\ \frac{\Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\Gamma; [\Gamma', \alpha] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}$$

Rule (DL-MATCH) concerns the case where the first entry in the environment matches the goal. Its behavior is the same as in the original definition of rule (FR-SIMP).

Rule (DL-NOMATCH) is mutually exclusive with the above rule: it skips the first entry in the environment only iff it does not match to look for a matching implicit deeper in the environment. This implements the committed choice semantics: the first matching implicit is committed to and further implicits are not considered.

Finally, rules (DL-VAR) and (DL-TYVAR) skip the irrelevant non-implicit entries in the type environment.

It is not difficult to see that with the above definition there is only one way to resolve the goal *Int* against the environment $\Gamma = ?Int \rightsquigarrow x, ?Int \rightsquigarrow y$. The first matching entry, which elaborates to y, is committed to and the second entry is not considered.

3.5.4 Stability

While the above committed-choice formulation of resolution is deterministic, it is a rather fragile, or *unstable*, notion of resolution. Consider for example resolving the goal *Int* against the environment $\Gamma = ?Int \rightsquigarrow x, \alpha, ?\alpha \rightsquigarrow y$. This scenario arises for instance when type checking the expression $e = \lambda_? Int.(\Lambda\alpha.\lambda_? \alpha.?Int)\, Int$. Our definition of resolution skips the first entry in the environment because α does not match *Int*, commits to the second entry because *Int* trivially matches *Int*, and elaborates to x.

However, this resolution is not stable. Consider what happens when we apply a seemingly innocuous refactoring to the expression e by β-reducing the type application. This yields the new, and supposedly equivalent, expression $e' = \lambda_? Int.\lambda_? Int.?Int$. The direct impact of this refactoring on the resolution problem is to substitute *Int* for the type variable α. As a consequence the resolution commits now to the first entry and elaborates to y instead of x. Hence, more generally, the above definition of resolution is not stable under type substitution. This is problematic because it defies the common expectation that simple refactorings like the reduction of type application above do not change a program's behavior.

To avoid this problem and obtain stability under type substitution, we tighten the requirement of rule (DL-NOMATCH): an implicit in the environment can only be skipped iff it does not match under any possible substitution of type variables. With this tightened requirement the scenario above simply does not resolve: unstable resolutions are invalid.

$$\text{(DL-NoMatch')}\ \frac{\mathsf{stable}(\Gamma; \rho \rightsquigarrow x; \tau) \qquad \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}{\Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}$$

where a first stab at a formalisation of the stability condition is:

$$\boxed{\mathsf{stable}(\Gamma; \rho \rightsquigarrow x; \tau)}$$

$$\text{(Stable')}\ \frac{\nexists \theta, E, \Sigma :\ \ \theta(\Gamma); [\theta(\rho)] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma; \theta(\tau) \rightsquigarrow E}{\mathsf{stable}(\Gamma; \rho \rightsquigarrow x; \tau)}$$

The above formulation of the condition is a bit too lax; we have to be more precise about the domain and range of the substitution θ. Indeed, substitution does not make sense for

every type variable in the environment. Consider for example resolving the type $\forall\beta.\beta \to \beta$ against the environment $\Gamma_0 = ?(\forall\gamma.\gamma \to \gamma) \rightsquigarrow x, \alpha, ?(\alpha \to \alpha) \rightsquigarrow y$. We would like this resolution of $\forall\beta.\beta \to \beta$ to succeed against $?(\forall\gamma.\gamma \to \gamma)$.

Unfortunately, the above formulation of stability unnecessarily throws a spanner in the works. Consider what happens: Using Rule (FR-TABS), we would recursively resolve $\beta \to \beta$ against the extended environment $\Gamma_1 = \Gamma_0, \beta$. Next we get stuck as neither rule (DL-MATCH) nor rule (DL-NOMATCH') applies. The former does not apply because $\alpha \to \alpha$ does not match $\beta \to \beta$. Also the latter does not apply because there are two substitutions such that $\theta(\alpha \to \alpha)$ matches $\theta(\beta \to \beta)$ and hence skipping $\alpha \to \alpha$ is deemed unstable.

However, if we look more closely at these substitutions, we see that none of them make sense. Essentially, there are two groups of substitutions:

- Those substitutions that instantiate β, of which $\theta = [\alpha/\beta]$ is a prominent example. These substitutions do not make sense because code inlining cannot result in β being instantiated to α or to any other type, because β is not in scope at the point in the code where the query happens (i.e., β does not appear in Γ_0). Hence, considering substitutions of β does not make sense.
  Figure 7, which puts all the measures together to obtain a type-directed, deterministic and stable resolution, addresses the issue as follows. It introduces a top-level *main* judgment $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$ to handle a query that delegates to the focusing-based judgments we have described above. The only contribution of the main judgment, which is defined by the single rule (R-MAIN), is to gather the type variables $\bar{\alpha}$ that appear in the environment at the point of the query by means of the function $\mathsf{tyvars}(\Gamma)$, and to pass them on through the auxiliary judgments to the point where the stability check is performed. Hence, the auxiliary judgments $\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$ (focusing), $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$ (lookup) and $\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)$ now all feature an additional argument $\bar{\alpha}$ of type variables that can be substituted.
- The substitution $\theta' = [\beta/\alpha]$ also generates a match. However, this substitution does not make sense either because code inlining can only result in substitutions of α by types that are well-scoped in the prefix of the environment before α. In the case of the example this means that we can only consider substitutions $[\sigma/\alpha]$ where $?(\forall\gamma.\gamma \to \gamma) \rightsquigarrow x \vdash \sigma$. In other words, σ cannot have any free type variables. There is no such σ that matches β.

In summary, Figure 6 formalises our notion of valid substitutions with the judgment $\bar{\alpha}; \Gamma \vdash \theta$. Rule (S-EMPTY) covers the base case and states that the empty substitution ε is trivially valid. Rule (S-CONS) covers the inductive case $[\sigma/\alpha] \cdot \theta$. It says that the single variable substitution $[\sigma/\alpha]$ is valid if α appears in the sequence of substitutable type variables, expressed by the structural pattern $\bar{\alpha}, \alpha, \bar{\alpha}'$. Moreover, α must appear in the type environment, expressed by a similar structural pattern Γ, α, Γ'. Lastly, the type σ must be well-scoped with respect to the environment prefix Γ. In addition, the remainder θ must be valid with respect to the remaining type variables $\bar{\alpha}, \bar{\alpha}'$ and the type environment after substitution of α.

$$\theta ::= \varepsilon \mid [\sigma/\alpha] \cdot \theta$$

$$\boxed{\bar{\alpha}; \Gamma \vdash \theta}$$

$$\text{(S-EMPTY)}\ \bar{\alpha}; \Gamma \vdash \varepsilon$$

$$\text{(S-CONS)}\ \frac{\Gamma \vdash \sigma \qquad \bar{\alpha}, \bar{\alpha}'; \Gamma, \theta(\Gamma') \vdash \theta}{\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma' \vdash [\sigma/\alpha] \cdot \theta}$$

Fig. 6. Valid Substitutions

3.5.5 Summary

Figure 7 puts all the above measures together in our unambiguous, deterministic and stable definition of resolution.

The *main judgment* $\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$ resolves the query ρ against the type environment Γ. It is defined by a single rule, (R-MAIN), which delegates the task to the auxiliary *focusing judgment* $\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$.

The focusing judgment has one more index than the main judgment, namely the type variables $\bar{\alpha}$ that are recorded in the type environment, which are retrieved by the function $\mathsf{tyvars}(\Gamma)$ in rule (R-MAIN). Three rules define the focusing judgment. The first two, (R-IABS) and (R-TABS), strip the query type ρ until only a simple type τ remains, which is handled by rule (R-SIMP). Rule (R-IABS) strips the context ρ_1 from a rule type $\rho_1 \Rightarrow \rho_2$, adds it to the type environment as a new implicit and then recursively processes the head ρ_1. Rule (R-TABS) strips the quantifier from a universally quantified type $\forall \alpha.\rho$ and adds the type variable α to the type environment in which ρ is processed. Finally, rule (R-SIMP) delegates the job of processing the simple type τ to the auxiliary *lookup judgment* $\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$.

The lookup judgment takes an additional index, the type environment Γ', which is initialised to Γ in rule (R-SIMP). It pops entries from Γ' until it finds an implicit that matches the simple query type τ. Rule (L-MATCH) first uses the auxiliary *matching judgment* $\Gamma; [\rho] \rightsquigarrow x \vdash_{\mathrm{r}} \overline{\rho' \rightsquigarrow x}; \tau \rightsquigarrow E$ to establish that implicit ρ at the top of Γ' matches the query type τ and to receive new queries ρ', which it resolves recursively against the type environment Γ. Rule (L-NOMATCH) skips the implicit at the top of the environment when it is stable to do so according to the auxiliary *stability judgment* $\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)$. Rules (L-VAR) and (L-TYVAR) skip term and type variable entries.

Three rules define the matching judgment. In the first one, (M-SIMP), the implicit is a simple type τ that is identical to query type; there are no remaining queries. When the implicit is a rule type $\rho_1 \Rightarrow \rho_2$, rule (M-IAPP) defers querying ρ_1 and first checks whether ρ_2 matches the query τ. When the implicit is a universally quantified type $\forall \alpha.\rho$, rule (M-TAPP) instantiates it appropriately to match the query τ.

Finally, the stability judgment is defined by a single rule, (STABLE), which makes sure that there is no substitution θ of the type variables $\bar{\alpha}$ for which $\theta(\rho)$ matches $\theta(\tau)$.

$$\Sigma ::= \varepsilon \mid \rho \rightsquigarrow x, \Sigma$$

$$\boxed{\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E} \qquad \textit{Main}$$

$$\text{(R-MAIN)}\ \frac{\mathsf{tyvars}(\Gamma); \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E}{\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E}$$

$$\begin{array}{rclcrcl} \mathsf{tyvars}(\varepsilon) & = & \varepsilon & & \mathsf{tyvars}(\Gamma, \alpha) & = & \mathsf{tyvars}(\Gamma), \alpha \\ \mathsf{tyvars}(\Gamma, x : \rho) & = & \mathsf{tyvars}(\Gamma) & & \mathsf{tyvars}(\Gamma, ?\rho \rightsquigarrow x) & = & \mathsf{tyvars}(\Gamma) \end{array}$$

$$\boxed{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E} \qquad \textit{Focusing}$$

$$\text{(R-IABS)}\ \frac{\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x \vdash_{\mathrm{r}} [\rho_2] \rightsquigarrow E \qquad x\ \textit{fresh}}{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho_1 \Rightarrow \rho_2] \rightsquigarrow \lambda x : |\rho_1| . E} \qquad \text{(R-TABS)}\ \frac{\bar{\alpha}; \Gamma, \alpha \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\forall \alpha . \rho] \rightsquigarrow \Lambda \alpha . E}$$

$$\text{(R-SIMP)}\ \frac{\bar{\alpha}; \Gamma; [\Gamma] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\tau] \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E} \qquad \textit{Lookup}$$

$$\text{(L-MATCH)}\ \frac{\Gamma; [\rho] \rightsquigarrow x \vdash_{\mathrm{r}} \overline{\rho' \rightsquigarrow x}; \tau \rightsquigarrow E \qquad \bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho'] \rightsquigarrow E' \quad (\forall \rho' \in \overline{\rho}')}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E[\bar{E}'/\bar{x}]}$$

$$\text{(L-NOMATCH)}\ \frac{\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau) \qquad \bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'}$$

$$\text{(L-VAR)}\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', x : \rho] \vdash_{\mathrm{r}} \tau \rightsquigarrow E} \qquad \text{(L-TYVAR)}\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', \alpha] \vdash_{\mathrm{r}} \tau \rightsquigarrow E}$$

$$\boxed{\Gamma; [\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'} \qquad \textit{Matching}$$

$$\text{(M-SIMP)}\ \Gamma; [\tau] \rightsquigarrow E \vdash_{\mathrm{r}} \varepsilon; \tau \rightsquigarrow E$$

$$\text{(M-IAPP)}\ \frac{\Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\, x \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E' \qquad x\ \textit{fresh}}{\Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash_{\mathrm{r}} \rho_1 \rightsquigarrow x, \Sigma; \tau \rightsquigarrow E'}$$

$$\text{(M-TAPP)}\ \frac{\Gamma; [\rho[\sigma/\alpha]] \rightsquigarrow E\, |\sigma| \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E' \qquad \Gamma \vdash \sigma}{\Gamma; [\forall \alpha . \rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'}$$

$$\boxed{\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)} \qquad \textit{Stability}$$

$$\text{(STABLE)}\ \frac{\nexists \theta, E, \Sigma : \quad \bar{\alpha}; \Gamma \vdash \theta \quad \theta(\Gamma); [\theta(\rho)] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma; \theta(\tau) \rightsquigarrow E}{\mathsf{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)}$$

Fig. 7. Deterministic Resolution and Translation to System F

4 Resolution Algorithm

This section presents an algorithm that implements the deterministic resolution rules of Figure 7. It differs from the latter in two important ways: firstly, it computes rather than guesses type substitutions in rule (M-TAPP); and secondly, it replaces explicit quantification over all substitutions θ in rule (STABLE) with a tractable approach to stability checking.

The definition of the algorithm, in Figure 8, is structured in the same way as the declarative specification: with one main judgment and three auxiliary ones that have similar roles (focusing, lookup, and matching). In fact, since the differences are not situated in the main and focusing judgment, these are actually identical.

4.1 Deferred Variable Instantiation

The first difference is situated in the matching judgment $\bar{\alpha};\Gamma;[\rho] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \Sigma';\tau \rightsquigarrow E'$. While its declarative counterpart immediately instantiates the quantified type variable in rule (M-TAPP), this algorithmic formulation defers the instantiation to the point where a deterministic choice can be made. As long as the type variables $\bar{\alpha}$ have not been instantiated, the judgment keeps track of them in its first argument. The actual instantiation happens in the base case, rule (ALG-M-SIMP). This last rule performs the deferred instantiation of type variables $\bar{\alpha}$ by computing the *most general unifier* $\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau',\tau)$. The unification algorithm, which we present below, computes a substitution θ that is valid (i.e., $\bar{\alpha};\Gamma \vdash \theta$) and that equates the two types (i.e., $\theta(\tau) = \theta(\tau')$).

In order to subject the recursive goals to this substitution, the algorithmic judgment makes use of an accumulating parameter Σ. This accumulator Σ represents all the goals collected so far in which type variables have not been substituted yet. In contrast, Σ' denotes all obligations with type variables already substituted.

Finally, observe that rule (ALG-L-MATCH) invokes the algorithmic judgment with an empty set of not-yet-instantiated type variables and an empty accumulator Σ.

The following example illustrates the differences between the declarative judgment:

$$\dfrac{\dfrac{\dfrac{\Gamma;[Int] \rightsquigarrow x\,Int\,y \vdash_{\text{r}} \varepsilon;Int \rightsquigarrow x\,Int\,y}{\text{(M-SIMP)}}}{\Gamma;[Int \Rightarrow Int] \rightsquigarrow x\,Int \vdash_{\text{r}} Int \rightsquigarrow y;Int \rightsquigarrow x\,Int\,y}\text{(M-IAPP)}}{\Gamma;[\forall\alpha.\alpha \Rightarrow \alpha] \rightsquigarrow x \vdash_{\text{r}} Int \rightsquigarrow y;Int \rightsquigarrow x\,Int\,y}\text{(M-TAPP)}$$

and its algorithmic counterpart:

$$\dfrac{\dfrac{\dfrac{[Int/\alpha] = \text{unify}_{\Gamma;\alpha}(\alpha,Int)}{\alpha;\Gamma;[\alpha] \rightsquigarrow x\,\alpha\,y;\alpha \rightsquigarrow y \vdash_{\text{alg}} \varepsilon;Int \rightsquigarrow x\,Int\,y}\text{(ALG-M-SIMP)}}{\alpha;\Gamma;[\alpha \Rightarrow \alpha] \rightsquigarrow x\,\alpha;\varepsilon \vdash_{\text{alg}} Int \rightsquigarrow y;Int \rightsquigarrow x\,Int\,y}\text{(ALG-M-IAPP)}}{\varepsilon;\Gamma;[\forall\alpha.\alpha \Rightarrow \alpha] \rightsquigarrow x;\varepsilon \vdash_{\text{alg}} Int \rightsquigarrow y;Int \rightsquigarrow x\,Int\,y}\text{(ALG-M-TAPP)}$$

4.2 Algorithmic Stability Check

The second difference can be found in (ALG-L-NOMATCH) of the lookup judgment. Instead of using the $\mathsf{stable}(\bar{\alpha};\Gamma;\rho \rightsquigarrow x;\tau)$ judgment, which quantifies over all valid sub-

stitutions, this rule uses the algorithmic judgment $\bar{\alpha};\Gamma;\rho \vdash_{\text{sta}} \tau$. This auxiliary judgment checks algorithmically whether the type ρ matches τ under any possible instantiation of the type variables $\bar{\alpha}$. We apply the same deferred-instantiation technique as with the first difference: Instead, of applying a substitution first and then checking whether the implicit matches the goal, we defer the instantiation to the end where we can deterministically pick one instantiation instead of considering all valid instantiations.

As a consequence of the similarity, in purpose and strategy, between the algorithmic stability, $\bar{\alpha};\Gamma;\rho \vdash_{\text{sta}} \tau$, and the matching judgment, $\bar{\alpha};\Gamma;[\rho] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \Sigma';\tau \rightsquigarrow E'$, the former is a variation of the latter, with two differences. Firstly, since the stability judgment is only concerned with matchability, no recursive resolvents Σ are collected nor are any elaborations tracked. Secondly, since the stability check considers the substitution of the type variables $\bar{\alpha}$ that occur in the environment at the point of the query, rule (ALG-L-NOMATCH) pre-populates the substitutable variables of the $\vdash_{\text{sta}}$ judgment with them. Contrast this with the matching judgment where only the implicit's quantified variables are instantiated, as witnessed by rules (ALG-M-TAPP) and (ALG-M-SIMP).

4.3 Scope-Aware Unification

The unification algorithm $\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\rho_1,\rho_2)$ is a key component of the two algorithmic changes explained above.

Figure 9 provides its definition, which is a hybrid between standard first-order unification (Martelli & Montanari, 1982) and polymorphic type instantiation (Dunfield & Krishnaswami, 2013) that is a restricted form of Miller's mixed-prefix unification (Miller, 1992). The domain restriction $\bar{\alpha}$ denotes which type variables are to be treated as unification variables; all other type variables are to be treated as constants. The returned substitution is a unifier of ρ_1 and ρ_2, i.e., $\theta(\rho_1) = \theta(\rho_2)$.

Validity The differences with standard first-order unification arise because, like polymorphic type instantiation, the algorithm has to account for the scope of type variables. Indeed, as we have already explained in Section 3.5, we expect that the returned substitution is valid, i.e., $\bar{\alpha};\Gamma \vdash \theta$. For instance, using standard first-order unification for $\text{unify}_{\Gamma;\beta}(\forall\alpha.\alpha \to \beta, \forall\alpha.\alpha \to \alpha)$ would yield the *invalid* substitution $[\beta/\alpha]$. The substitution is invalid because α is not in scope in Γ.

Most General Unifier Secondly, traditional unification computes the most general unifier, i.e., any other unifier can be expressed as its composition with another substitution. Yet, the most general unifier may not be a valid substitution, while more specific unifiers may be valid. Consider for instance unifying α with $\beta \to \beta$ where $\Gamma = \alpha,\beta$ and both α and β are unification variables. The most general unifier is $[\beta \to \beta/\alpha]$. However this unifier is not valid, as α appears before β in the environment. In contrast, there are infinitely many more specific unifiers that are valid, all of the form $[\rho \to \rho/\alpha, \rho/\beta]$ where ρ is a closed type.

Fortunately, by a stroke of luck, the above is not a problem for either of our two use cases:

- The first use case is that in rule (ALG-M-SIMP) where this is not a problem because the scenario never arises. In $\text{unify}_{\Gamma;\bar{\alpha}}(\tau',\tau)$ only τ' contains unification variables and hence the range of the substitution never contains any unification variables. As a consequence the above example and others like it cannot occur.
- The second use case, in rule (STA-SIMP), is only interested in the existence of a valid substitution. We neither care which one it is nor whether it is the most general one. Moreover, as illustrated above, whenever there is a most general substitution that is invalid due to the relative position of unification variables in the environment, we can always construct a more specific valid substitution by substituting the remaining unification variables by closed types.

Definition With the above issues in mind we can consider the actual definition in Figure 8. The main unification judgment $\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\rho_1,\rho_2)$ is defined by rule (U-MAIN). This rule computes the unifier in terms of the auxiliary judgment $\theta = \text{unify}'_{\bar{\alpha}}(\rho_1,\rho_2)$, which is essentially standard unification, and then checks the above validity concerns. Indeed, for any type variable β that appears in the image of a type variable α, either β must appear before α in the environment Γ (regular validity), or β must itself be a unification variable (the exceptional case). The relative position of variables is checked with the auxiliary judgment $\beta >_\Gamma \alpha$ whose one rule verifies that β appears before α in the environment Γ;[7] a similar check on relative positions can be found in Dunfield and Krishnaswami's algorithm (Dunfield & Krishnaswami, 2013).

The auxiliary judgment $\text{unify}'_{\bar{\alpha}}(\rho_1,\rho_2)$ computes the actual unifier. Rule (U-VAR) is the standard reflexivity rule for type variables. Rules (U-INSTL) and (U-INSTR) are two symmetric base cases; they only create a substitution $[\sigma/\alpha]$ if α is one of the unification variables and if α does not occur in σ, which is the well-known occurs-check. Rules (U-FUN), (U-RUL) and (U-UNIV) are standard congruence rules that combine the unification problems derived for their subterms.

4.4 Termination of Resolution

If we are not careful about which implicits are added to the environment, then the resolution process may not terminate. This section describes how to impose a set of modular syntactic restrictions that prevents non-termination. As an example of non-termination consider

$$?(\mathit{Char} \Rightarrow \mathit{Int}), ?(\mathit{Int} \Rightarrow \mathit{Char}) \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Int}$$

which loops, using alternatively the first and second implicit in the environment. The source of this non-termination is the recursive nature of resolution: a simple type can be resolved in terms of an implicit type whose head it matches, but this requires further resolution of the implicit type's context.

The problem of non-termination has been widely studied in the context of Haskell's type classes, and a set of modular syntactic restrictions has been imposed on type class instances

[7] If type variables are represented by de Bruijn indices, this can be done by checking whether one index is greater than the other.

$$\boxed{\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E} \qquad \textit{Main}$$

$$(\textsc{Alg-R-Main})\ \frac{\mathsf{tyvars}(\Gamma); \Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E}{\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E} \qquad \textit{Focusing}$$

$$(\textsc{Alg-R-IAbs})\ \frac{\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x \vdash_{\text{alg}} \rho_2 \rightsquigarrow E \qquad x \textit{ fresh}}{\bar{\alpha}; \Gamma \vdash_{\text{alg}} \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda(x : |\rho_1|).E}$$

$$(\textsc{Alg-R-TAbs})\ \frac{\bar{\alpha}; \Gamma, \alpha \vdash_{\text{alg}} \rho \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\text{alg}} \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E} \qquad (\textsc{Alg-R-Simp})\ \frac{\bar{\alpha}; \Gamma; [\Gamma] \vdash_{\text{alg}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{\text{alg}} \tau \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E} \qquad \textit{Lookup}$$

$$(\textsc{Alg-L-Match})\ \frac{\varepsilon; \Gamma; [\rho] \rightsquigarrow x; \varepsilon \vdash_{\text{alg}} \bar{\rho}' \rightsquigarrow \vec{x}'; \tau \rightsquigarrow E \qquad \bar{\alpha}; \Gamma \vdash_{\text{alg}} \rho' \rightsquigarrow E' \ \ (\forall \rho' \in \bar{\rho}')}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{alg}} \tau \rightsquigarrow E[\bar{E}'/\vec{x}']}$$

$$(\textsc{Alg-L-NoMatch})\ \frac{\bar{\alpha}; \Gamma; \rho \not\vdash_{\text{sta}} \tau \qquad \bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{alg}} \tau \rightsquigarrow E'}$$

$$(\textsc{Alg-L-Var})\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', x : \rho] \vdash_{\text{alg}} \tau \rightsquigarrow E} \qquad (\textsc{Alg-L-TyVar})\ \frac{\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; [\Gamma', \alpha] \vdash_{\text{alg}} \tau \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma; [\rho] \rightsquigarrow E; \Sigma \vdash_{\text{alg}} \Sigma'; \tau \rightsquigarrow E'} \qquad \textit{Matching}$$

$$(\textsc{Alg-M-Simp})\ \frac{\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')}{\bar{\alpha}; \Gamma; [\tau'] \rightsquigarrow E; \Sigma \vdash_{\text{alg}} \theta(\Sigma); \tau \rightsquigarrow |\theta|(E)}$$

$$(\textsc{Alg-M-IApp})\ \frac{\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\,x; \Sigma, \rho_1 \rightsquigarrow x \vdash_{\text{alg}} \Sigma'; \tau \rightsquigarrow E' \qquad x \textit{ fresh}}{\bar{\alpha}; \Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E; \Sigma \vdash_{\text{alg}} \Sigma'; \tau \rightsquigarrow E'}$$

$$(\textsc{Alg-M-TApp})\ \frac{\bar{\alpha}, \alpha; \Gamma, \alpha; [\rho] \rightsquigarrow E\,\alpha; \Sigma \vdash_{\text{alg}} \Sigma'; \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; [\forall\alpha.\rho] \rightsquigarrow E; \Sigma \vdash_{\text{alg}} \Sigma'; \tau \rightsquigarrow E'}$$

$$\boxed{\bar{\alpha}; \Gamma; \rho \vdash_{\text{sta}} \tau} \qquad \textit{Stability}$$

$$(\textsc{Sta-TApp})\ \frac{\bar{\alpha}, \alpha; \Gamma, \alpha; \rho \vdash_{\text{sta}} \tau}{\bar{\alpha}; \Gamma; \forall\alpha.\rho \vdash_{\text{sta}} \tau} \qquad (\textsc{Sta-IApp})\ \frac{\bar{\alpha}; \Gamma; \rho_2 \vdash_{\text{sta}} \tau}{\bar{\alpha}; \Gamma; \rho_1 \Rightarrow \rho_2 \vdash_{\text{sta}} \tau}$$

$$(\textsc{Sta-Simp})\ \frac{\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')}{\bar{\alpha}; \Gamma; \tau' \vdash_{\text{sta}} \tau}$$

Fig. 8. Resolution Algorithm

$$\boxed{\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)}$$

$$\text{(U-MAIN)}\ \frac{\theta = \text{unify}'_{\bar{\alpha}}(\rho_1, \rho_2) \qquad \beta \in \bar{\alpha} \ \vee\ \beta >_\Gamma \alpha \quad (\forall [\sigma/\alpha] \in \theta, \forall \beta \in \mathit{ftv}(\sigma))}{\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)}$$

$$\boxed{\theta = \text{unify}'_{\bar{\alpha}}(\rho_1, \rho_2)}$$

$$\text{(U-VAR)}\ \frac{}{\varepsilon = \text{unify}'_{\bar{\alpha}}(\beta, \beta)}$$

$$\text{(U-INSTL)}\ \frac{\alpha \in \bar{\alpha} \qquad \alpha \notin \mathit{ftv}(\sigma)}{[\sigma/\alpha] = \text{unify}'_{\bar{\alpha}}(\alpha, \sigma)} \qquad \text{(U-INSTR)}\ \frac{\alpha \in \bar{\alpha} \qquad \alpha \notin \mathit{ftv}(\sigma)}{[\sigma/\alpha] = \text{unify}'_{\bar{\alpha}}(\sigma, \alpha)}$$

$$\text{(U-FUN)}\ \frac{\theta_1 = \text{unify}'_{\bar{\alpha}}(\rho_{11}, \rho_{21}) \qquad \theta_2 = \text{unify}'_{\bar{\alpha}}(\theta_1(\rho_{12}), \theta_1(\rho_{22}))}{\theta_2 \cdot \theta_1 = \text{unify}'_{\bar{\alpha}}(\rho_{11} \to \rho_{12}, \rho_{21} \to \rho_{22})}$$

$$\text{(U-RUL)}\ \frac{\theta_1 = \text{unify}'_{\bar{\alpha}}(\rho_{11}, \rho_{21}) \qquad \theta_2 = \text{unify}'_{\bar{\alpha}}(\theta_1(\rho_{12}), \theta_1(\rho_{22}))}{\theta_2 \cdot \theta_1 = \text{unify}'_{\bar{\alpha}}(\rho_{11} \Rightarrow \rho_{12}, \rho_{21} \Rightarrow \rho_{22})}$$

$$\text{(U-UNIV)}\ \frac{\theta = \text{unify}'_{\bar{\alpha}}(\rho_1, \rho_2)}{\theta = \text{unify}'_{\bar{\alpha}}(\forall \beta.\rho_1, \forall \beta.\rho_2)}$$

$$\boxed{\beta >_\Gamma \alpha} \qquad \overline{\beta >_{\Gamma_1,\beta,\Gamma_2,\alpha,\Gamma_3} \alpha}$$

Fig. 9. Unification Algorithm

to avoid non-termination (Sulzmann *et al.*, 2007). This paper adapts those restrictions to the setting of COCHIS.

Overall Approach We show termination by characterising the resolution process as a (resolution) tree with goals in the nodes. The initial goal sits at the root of the tree. A multi-edge from a parent node to its children represents a rule type from the environment that matches the parent nodes goal; the node's children are the recursive goals.

Resolution terminates if the tree is finite. Hence, if it does not terminate, there is an infinite path from the root in the tree, that denotes an infinite sequence of matching rule type applications. To show that there cannot be such an infinite path, we use a norm $\|\cdot\|$ (defined at the bottom of Figure 10) that maps the head of every goal ρ to a natural number, its size. While types like *Int* and *Bool* are not formally part of our calculus, we assume that their size is 1. Similarly, the size of type constructor applications like *List* ρ is $1 + \|\rho\|$.

If we can show that this size strictly decreases from any parent goal to its children, then we know that, because the order on the natural numbers is well-founded, on any path from the root there is eventually a goal that has no children.

Termination Condition It is trivial to show that the size strictly decreases, if we require that every implicit binding in the environment makes it so. This requirement is formalised as the termination condition $\vdash_{\text{term}} \rho$ in Figure 10. This condition should be imposed on

every type added to the environment, namely to ρ_1 in rule (TY-IABS) of Figure 2 and to ρ_1 in rule (R-IABS) of Figure 7. However, because the latter concerns only a part of a resolved type, we feel that it is easier to follow for the programmer if we impose the condition instead on the whole resolved type in rule (TY-QUERY) of Figure 2.

The judgment is defined by case analysis on the type ρ. Rule (T-SIMP) states that simple types trivially satisfy the condition. Next, rule (T-FORALL) is the congruence rule for universally quantified types. Finally, rule (T-RULE) enforces the actual condition on rule types $\rho_1 \Rightarrow \rho_2$, which requires that the head τ_1 of ρ_1 is strictly smaller than the head τ_2 of ρ_2.

To account for polymorphism and the fact that the type variables in rule types can be instantiated, rule (T-RULE) ensures that the $\|\tau_1\| < \|\tau_2\|$ property is stable under substitution. Declaratively, we can formulate this stability under substitution as:

$$\forall \theta.dom(\theta) \subseteq ftv(\tau_1) \cup (ftv)(\tau_2) : \ \|\theta(\tau_1)\| < \|\theta(\tau_2)\|$$

Consider for instance the type $\forall a.(a \to a) \Rightarrow (a \to Int \to Int)$. In this example, the head's size 5 is strictly greater than the context constraint's size 3. Yet, if we instantiate α to $(Int \to Int \to Int)$, then the head's size becomes 9 while the context constraint's size becomes 11.

The declarative formulation above is not suitable in an algorithm because it enumerates all possible substitutions. Rule (T-RULE) uses instead an equivalent algorithmic formulation which states that, in addition to $\|\tau_1\| < \|\tau_2\|$, the number of occurrences of any free type variable α may not be larger in τ_1 than in τ_2. The first condition expresses that for the empty substitution, the size strictly decreases, say from $\|\tau_2\| = n$ to $\|\tau_1\| = m$. If we instantiate a type variable α to a type σ of size k, then the sizes change to $\|[\sigma/\alpha]\tau_1\| = m + k \times \mathrm{occ}_\alpha(\tau_1)$ and $\|[\sigma/\alpha]\tau_2\| = n + k \times \mathrm{occ}_\alpha(\tau_2)$ where the auxiliary function $\mathrm{occ}_\alpha(\rho)$ determines the number of occurrences of α in ρ. The second condition guarantees that $k \times \mathrm{occ}_\alpha(\tau_1) \leqslant k \times \mathrm{occ}_\alpha(\tau_2)$ and thus that the strict decrease is preserved under substitution. In our example above, we do have that $\|\alpha \to \alpha\| = 3 < 5 = \|\alpha \to Int \to Int\|$, the second condition is not satisfied, i.e., $\mathrm{occ}_\alpha(\alpha \to \alpha) = 2 \not\leqslant 1 = \mathrm{occ}_\alpha(\alpha \to Int \to Int)$.

Finally, as the types have a recursive structure whereby their components are themselves added to the environment, rule (T-RULE) also enforces the termination condition recursively on the components.

Discussion Above we have adapted termination conditions for Haskell's type class resolution to COCHIS. While our adapted conditions are sufficient for termination, they are not necessary. In fact, they can be rather restrictive. For instance, $\not\vdash_{\text{term}} Int \Rightarrow Bool$ because $\|Int\| \not< \|Bool\|$. Indeed, resolving $Bool$ in the context $\Gamma_1 = ?(Bool \Rightarrow Int), ?(Int \Rightarrow Bool)$ is problematic. Yet, it is not in the context $\Gamma_2 = ?(Int), ?(Int \Rightarrow Bool)$. The problem is that the conditions are not context sensitive. We leave exploring more permissive, context-sensitive conditions to future work.

5 Type-Directed Translation to System F

In this section we explain the dynamic semantics of COCHIS in terms of System F's dynamic semantics, by means of a type-directed translation. This translation turns implicit

$$\boxed{\vdash_{\text{term}} \rho} \qquad \text{(T-SIMP)}\ \frac{}{\vdash_{\text{term}} \tau} \qquad \text{(T-FORALL)}\ \frac{\vdash_{\text{term}} \rho}{\vdash_{\text{term}} \forall\alpha.\rho}$$

$$\text{(T-RULE)}\ \frac{\vdash_{\text{term}} \rho_1 \quad \vdash_{\text{term}} \rho_2 \quad \tau_1 = \text{hd}(\rho_1) \quad \tau_2 = \text{hd}(\rho_2) \quad \|\tau_1\| < \|\tau_2\| \quad \forall\alpha \in ftv(\rho_1) \cup ftv(\rho_2) : \text{occ}_\alpha(\tau_1) \leqslant \text{occ}_\alpha(\tau_2)}{\vdash_{\text{term}} \rho_1 \Rightarrow \rho_2}$$

$$\text{hd}(\tau) = \tau \qquad \text{hd}(\forall\alpha.\rho) = \text{hd}(\rho) \qquad \text{hd}(\rho_1 \Rightarrow \rho_2) = \text{hd}(\rho_2)$$

$$\begin{array}{rclrcl} \text{occ}_\alpha(\beta) & = & \begin{cases} 1 & (\alpha = \beta) \\ 0 & (\alpha \neq \beta) \end{cases} & \text{occ}_\alpha(\forall\beta.\rho) & = & \text{occ}_\alpha(\rho) \quad (\alpha \neq \beta) \\ \text{occ}_\alpha(\rho_1 \to \rho_2) & = & \text{occ}_\alpha(\rho_1) + \text{occ}_\alpha(\rho_2) & \text{occ}_\alpha(\rho_1 \Rightarrow \rho_2) & = & \text{occ}_\alpha(\rho_1) + \text{occ}_\alpha(\rho_2) \end{array}$$

$$\begin{array}{rclrcl} \|\alpha\| & = & 1 & \|\forall\alpha.\rho\| & = & \|\rho\| \\ \|\rho_1 \to \rho_2\| & = & 1 + \|\rho_1\| + \|\rho_2\| & \|\rho_1 \Rightarrow \rho_2\| & = & 1 + \|\rho_1\| + \|\rho_2\| \end{array}$$

Fig. 10. Termination Condition

contexts into explicit parameters and statically resolves all queries, much like Wadler and Blott's dictionary passing translation for type classes (Wadler & Blott, 1989). The advantage of this approach is that we simultaneously provide a meaning to well-typed COCHIS programs and an effective implementation that resolves all queries statically.

The translation follows the type system presented in Section 3. The additional machinery that is necessary (on top of the type system) corresponds to the grayed parts of Figures 2, 3 and 7.

5.1 Type-Directed Translation

Figure 2 presents the translation rules that convert COCHIS expressions into System F expressions. The gray parts of the figure extend the type system with the necessary information for the translation.

The syntax of System F is as follows:

$$\begin{array}{lrcl} \text{Types} & T & ::= & \alpha \mid T \to T \mid \forall\alpha.T \\ \text{Expressions} & E & ::= & x \mid \lambda(x:T).E \mid E\ E \mid \Lambda\alpha.E \mid E\ T \end{array}$$

The gray extension to the syntax of type environments annotates every implicit's type with explicit System F evidence in the form of a term variable x.

Translation of Types The function $|\cdot|$ takes COCHIS types ρ to System F types T:

$$\begin{array}{rclrcl} |\alpha| & = & \alpha & |\forall\alpha.\rho| & = & \forall\alpha.|\rho| \\ |\rho_1 \to \rho_2| & = & |\rho_1| \to |\rho_2| & |\rho_1 \Rightarrow \rho_2| & = & |\rho_1| \to |\rho_2| \end{array}$$

It reveals that implicit COCHIS arrows are translated to explicit System F function arrows.

Translation of Terms The type-directed translation judgment, which extends the typing judgment, is

$$\Gamma \vdash e : \rho \rightsquigarrow E$$

This judgment states that the translation of COCHIS expression e with type ρ is System F expression E, with respect to type environment Γ.

Variables, lambda abstractions and applications are translated straightforwardly. Perhaps the only noteworthy rule is (TY-IABS). This rule associates the type ρ_1 with the fresh variable x in the type environment. This creates the necessary evidence that can be used by resolutions in the body of the rule abstraction to construct System F terms of type $|\rho_1|$.

Resolution The more interesting part of the translation happens when resolving queries. Queries are translated by rule (TY-QUERY) using the auxiliary resolution judgment $\vdash_r$:

$$\Gamma \vdash_r \rho \rightsquigarrow E$$

which is shown, in deterministic form, in Figure 7. The translation builds a System F term as evidence for the resolution.

The mechanism that builds evidence dualizes the process of peeling off abstractions and universal quantifiers: Rule (R-IABS) wraps a lambda binder with a fresh variable x around a System F expression E, which is generated from the resolution for the head of the rule (ρ_2). Similarly, rule (R-TABS) wraps a type lambda binder around the System F expression resulting from the resolution of ρ.

For simple types τ rule (R-SIMP) delegates the work of building evidence, when a matching implicit ρ is found in the environment, to rule (L-MATCH). The evidence consists of two parts: E is the evidence of matching τ against ρ. This match contains placeholders $\bar{x}$ for the contexts whose resolution is postponed by rule (M-IAPP). It falls to rule (L-MATCH) to perform these postponed resolutions, obtain their evidence $\bar{E}$ and fill in the placeholders.

Meta-Theory The type-directed translation of COCHIS to System F exhibits a number of desirable properties.

Theorem 5.1 (Type-Preserving Translation)**.** *Let e be a* COCHIS *expression, ρ be a type, Γ a type environment and E a System F expression. If $\Gamma \vdash e : \rho \rightsquigarrow E$, then $|\Gamma| \vdash_F E : |\rho|$.*

Here we define the translation of the type environment from COCHIS to System F as:

$$\begin{array}{rcl@{\qquad}rcl} |\varepsilon| & = & \varepsilon & |\Gamma, \alpha| & = & |\Gamma|, \alpha \\ |\Gamma, x : \rho| & = & |\Gamma|, x : |\rho| & |\Gamma, ?\rho \rightsquigarrow x| & = & |\Gamma|, x : |\rho| \end{array}$$

An important lemma in the theorem's proof is the type preservation of resolution.

Lemma 5.1 (Type-Preserving Resolution)**.** *Let Γ be a type environment, ρ be a type and E be a System F expression. If $\Gamma \vdash_r^a \rho \rightsquigarrow E$, then $|\Gamma| \vdash_F E : |\rho|$.*

Section A.1 of the Appendix points to the mechanized proofs of Theorem 5.1 and Lemma 5.1 stated above.

In addition, we can express three key properties of Figure 7's definition of resolution in terms of the generated evidence. Firstly, the deterministic version of resolution is a sound variation on the original ambiguous resolution.

Theorem 5.2 (Soundness)**.** *Figure 7's deterministic definition of resolution is sound (but incomplete) with respect to Figure 3's ambiguous definition.*

$$\forall \Gamma, \rho, E: \quad \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E \quad \Rightarrow \quad \Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow E$$

Section A.3 of the Appendix points to the mechanized proof of the above theorem. As a corollary (by use of this result with Lemma 5.1), the deterministic resolution process is also type-preserving. Secondly, the deterministic resolution is effectively deterministic:

Theorem 5.3 (Determinacy)**.** *Let Γ be a type environment, ρ be a type and E_1, E_2 be system F expressions. Assume that $\vdash_{\mathrm{unamb}} \rho$ and $\forall ?\rho_i \in \Gamma, \vdash_{\mathrm{unamb}} \rho_i$. Then*

$$\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_1 \ \wedge \ \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_2 \quad \Rightarrow \quad E_1 = E_2$$

A full proof of Theorem 5.3 can be found in Section A.4 of the Appendix.

It follows immediately that deterministic resolution is also coherent, which for our elaboration-based setting is formulated, following Biernacki and Polesiuk (2018), in terms of *contextual equivalence* (Morris, 1969).

Corollary 5.1 (Coherence)**.** *Let Γ be a type environment, ρ be a type and E_1, E_2 be system F expressions. Assume that $\vdash_{\mathrm{unamb}} \rho$ and $\forall ?\rho_i \in \Gamma, \vdash_{\mathrm{unamb}} \rho_i$. Then*

$$\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_1 \ \wedge \ \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_2 \quad \Rightarrow \quad |\Gamma| \models_{\mathrm{F}} E_1 \simeq_{\mathrm{ctx}} E_2 : |\rho|$$

Here, the well-studied (Chong, 2017) contextual equivalence judgment $\Gamma \models_{\mathrm{F}} E_1 \simeq_{\mathrm{ctx}} E_2 : T$ captures that E_1 and E_2 behave the same in any well-typed program context. We do not give the formal definition here as it is not required to prove the corollary; we only need to know that it is an equivalence relation, and, more specifically, that it is reflexive, i.e., any well-typed System F term is contextually equivalent to itself.

Thirdly, on top of the immediate coherence of deterministic resolution, an additional stability property holds.

Lemma 5.2 (Stability of Resolution)**.** *Resolution is stable under monotype substitution.*

$$\forall \Gamma, \alpha, \Gamma', \sigma, \rho, E: \quad \Gamma, \alpha, \Gamma' \vdash_{\mathrm{r}} \rho \rightsquigarrow E \ \wedge \ \Gamma \vdash \sigma \ \Rightarrow \ \Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

This is a key lemma to establish that static reduction of type application preserves typing and elaboration.

Lemma 5.3 (Stability of Typing under Type Application Reduction)**.** *Static reduction of type application preserves typing.*

$$\forall \Gamma, \alpha, \sigma, \rho, E: \quad \Gamma \vdash (\Lambda \alpha.e)\, \sigma : \rho \rightsquigarrow (\Lambda \alpha.E)\, |\sigma| \ \Rightarrow \ \Gamma \vdash e[\sigma/\alpha] : \rho \rightsquigarrow E[|\sigma|/\alpha]$$

This, together with another property of System F's contextual equivalence:

$$\forall \Gamma, \alpha, E, T, T': \quad \Gamma \models_{\mathrm{F}} (\Lambda \alpha.E)\, T : T' \ \Rightarrow \ \Gamma \models_{\mathrm{F}} (\Lambda \alpha.E)\, T \simeq_{\mathrm{ctx}} E[T/\alpha] : T'$$

allows us to conclude the correctness of static reduction of type application.

Theorem 5.4 (Correctness of Type Application Reduction)**.** *If a type application and its reduced form elaborate to two System F terms, those terms are contextually equivalent.*

$$\forall \Gamma, \alpha, e, \sigma, E1, E2 : \Gamma \vdash (\Lambda \alpha.e)\, \sigma : \rho \rightsquigarrow E_1 \wedge \Gamma \vdash e[\sigma/\alpha] : \rho \rightsquigarrow E_2 \ \Rightarrow \ |\Gamma| \models_{\mathrm{F}} E_1 \simeq_{\mathrm{ctx}} E_2 : |\rho|$$

The stability Lemmas 5.2 and 5.3 are proved in Section A.5 of the Appendix.

5.2 Evidence Generation in the Algorithm

The evidence generation in Figure 8 is largely similar to that in the deterministic specification of resolution in Figure 7. With the evidence we can state the correctness of the algorithm.

Theorem 5.5 (Partial Correctness)**.** *Let* Γ *be a type environment,* ρ *be a type and* E *be a System F expression. Assume that* $\vdash_{\text{unamb}} \rho$ *and also* $\vdash_{\text{unamb}} \rho_i$ *for all* $?\rho_i \in \Gamma$*. Then* $\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E$ *if and only if* $\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$*.*

The proof is split in two parts, one for each "direction" of the theorem. Section A.7 proves soundness of the algorithm with respect to deterministic resolution, while Section A.8 proves partial completeness. The latter correctness property is partial because it does not hold without the additional termination conditions for the queried type and the type environment, Γ.

5.3 Dynamic Semantics

Finally, we define the dynamic semantics of COCHIS as the composition of the type-directed translation and System F's dynamic semantics. Following Siek's notation (Siek & Lumsdaine, 2005), this dynamic semantics is:

$$eval(e) = V \qquad \textit{where } \varepsilon \vdash e : \rho \rightsquigarrow E \textit{ and } E \rightarrow^* V$$

with $\rightarrow^*$ the reflexive, transitive closure of System F's standard single-step call-by-value reduction relation (see Chapter 23 of (Pierce, 2002)).

Now we can state the conventional type safety theorem for COCHIS:

Theorem 5.6 (Type Safety)**.** *If* $\varepsilon \vdash e : \rho$*, then* $eval(e) = V$ *for some System F value* V*.*

The proof follows from Theorem 5.1 and System F's well-known normalization property.

6 Discussion

In this section we discuss and justify several of the design decisions made during the creation of COCHIS. Mostly, these choices are motivated by the design of Haskell type classes or Scala implicits.

6.1 Predicative Instantiation

System F is an impredicative calculus, allowing the instantiation of type variables with arbitrary (polymorphic) types. In contrast COCHIS is predicative: instantiation of type variables only allows *monotypes*. Our reasons for departing from System F are threefold:

- **Impredicative instantiation in resolution leads to additional ambiguity.** As discussed in Section 3, impredicative instantiations of type variables during resolution can lead to ambiguity. The restriction to predicative instantiation removes this ambiguity, and we see no way that preserves impredicativity to achieve the same.

- **Impredicativity is complex for implicit instantiation.** While System F (where all type instantiations are explicit) is simple, matters become much more complicated when some implicit instantiation is allowed. Indeed, the design of System F-like calculi with implicit instantiation and/or some form of type-inference (Odersky & Läufer, 1996; Jones *et al.*, 2007; Le Botlan & Rémy, 2003; Leijen, 2008; Vytiniotis *et al.*, 2008) is much more divided in terms of design choices regarding (im)predicativity. Notably, Rouvoet (2016) has shown that the ambiguous resolution from the implicit calculus (Oliveira *et al.*, 2012), which is the impredicative variant of our Figure 3, is undecidable. His proof proceeds by showing that the problem is equivalent to the System F type inhabitation problem, which is known to be undecidable (Barendregt *et al.*, 2013).

- **Predicative instantiation is not a big restriction in practice.** Due to the above complications brought by impredicativity, many practical languages with type-inference only allow predicative instantiation. For example, the key algorithm for type-inference currently employed by the GHC Haskell compiler is predicative (Jones *et al.*, 2007; Vytiniotis *et al.*, 2011). Worth noting is that the original Hindley-Milner (HM) type system (Hindley, 1969; Milner, 1978) is where the predicativity restriction on polymorphic type systems with type-inference was first put into place. Since COCHIS is intended as a target for languages with implicit polymorphism and type-inference, which have predicativity restrictions, restricting the core language to allow only predicative instantiation does not pose any problems.

Alternative Design Choices One alternative design choice worth mentioning for COCHIS would be to allow impredicative instantiation in explicit type applications, but still retain the predicativity restriction during resolution. This design would be less restrictive than the design of COCHIS, and we believe that it is a reasonable design. We decided against it for two reasons. Firstly, as already mentioned, COCHIS is aimed to be a target for source languages with type-inference. As these source languages have predicative restrictions anyway, there is little to be gained from having impredicative instantiation in the core. Secondly, and more importantly, some of the meta-theory would be more involved if impredicative instantiation on type applications were allowed. In particular, Lemmas 5.2, 5.3 and 5.4 would need to be generalized to allow any types to be used in the substitution, rather than just monotypes. This could be problematic since the impredicative instantiations of the type variables could bring back the ambiguity issues discussed in Section 3. We expect that additional restrictions would be needed at type applications to prevent instantiations with problematic polymorphic types that would lead to ambiguity.

Allowing full impredicativity (both in type applications and resolution) seems more complicated. We expect that such a design is possible, but necessarily more complicated if ambiguity and undecidability are to be avoided. We expect that the work on impredicative type-inference (Le Botlan & Rémy, 2003; Leijen, 2008; Vytiniotis *et al.*, 2008) is relevant, and perhaps some of the design choices employed in those works would be helpful in the design of a system with full impredicativity.

6.2 Committed Choice

COCHIS commits to the first implicit whose head matches the query type. It has inherited this committed choice approach from Haskell. Consider for instance the following Haskell program with two overlapping instances:

```
class C α where
  m :: α → α
instance Eq α  ⇒ C [α] where ...
instance Ord α ⇒ C [α] where ...
f :: StablePtr Int → [StablePtr Int]
f sp = m [sp]
```

This code declares a type class $C\ \alpha$ and defines two instances. The first instance requires $Eq\ \alpha$, whereas the second instance requires $Ord\ \alpha$. The function f takes a stable pointer (*StablePtr*) and returns a list of stable pointers. Unlike many other types, *StablePtr* only supports equality, and not ordering. That is, there is an instance $Eq\ (StablePtr\ \alpha)$ but not one for $Ord\ (StablePtr\ \alpha)$.

Should the above code type-check or not? In GHC Haskell the answer is no. Even though there is no ambiguity in this program— resolution only succeeds with the first type class instance— the program is nevertheless rejected. The reason is that Haskell's resolution only checks whether the instance heads match. As there are two equality specific matching heads $C\ [\alpha]$, the program is rejected.

Although this Haskell design choice is not very well documented in the research literature, the reason for not allowing backtracking is folklore among Haskell programmers and can be found in various informal discussions (oleg, 2006; rHaskell, 2006; Bottu & simonpj, 2018). In essence there are two arguments for not allowing backtracking during resolution:

- **Reasoning:** When reasoning about Haskell code that involves type classes, programmers have to understand which type class instance is used. This involves performing the resolution algorithm mentally. The fact that only instance heads are needed to determine whether an instance is committed to, makes this much easier than performing a full backtracking process.
- **Performance:** If backtracking is allowed, type-checking times of programs could grow exponentially due to backtracking. Thus, by disallowing backtracking, GHC eliminates a potential source of significant performance degradation in type-checking.

No Backtracking in COCHIS Like in Haskell, our committed choice design for COCHIS also looks only at the heads of rule types, whereas a design with backtracking would need to look at their contexts as well. In other words we commit to a matching rule type, even if its recursive goals do not resolve. For instance, when resolving *Char* against the environment $\Gamma = ?Bool, ?(Bool \Rightarrow Char), ?(Int \Rightarrow Char)$, we commit to $?(Int \Rightarrow Char)$ even though its recursive goal *Int* cannot be resolved and thus the resolution of *Char* also fails. A more permissive approach would be to backtrack when a recursive resolution fails and try the next alternative matching implicit. That would allow *Char* to resolve.

In the design of COCHIS, we have followed Haskell's pragmatic reasons for committed choice. Considering that Haskell's 30 years of experience have shown that this works well in practice, we believe that it is a reasonable choice.

Alternative Designs While there are advantages to committed choice, backtracking also has its appeal and several systems have adopted it (White *et al.*, 2014; Sozeau & Oury, 2008). In particular, backtracking accepts more queries, and would allow us to have a sound and complete algorithm for COCHIS (instead of just a sound one) with respect to Figure 4. Yet, the specification in Figure 4 does not guarantee stability or coherence. So additional restrictions would be needed to have both properties.

OCaml's modular implicits (White *et al.*, 2014) suggest another alternative. When a query is resolved, it exhaustively searches the implicit context for all possible solutions. If more than one solution is found, then the program is rejected due to ambiguity. In this way it is possible to have highly overlapping implicits in the type environment, that could result in some queries being ambiguous. One advantage of this design is its flexibility, since contexts can be more liberal and all queries that would be resolved in unambiguous contexts with backtracking can, in principle, also be resolved with OCaml's modular implicits. However the modular implicits approach is not formalized yet, and the fact that contexts have to be searched exhaustively raises practical questions regarding performance and ease of reasoning that have dictated the committed choice approach taken by Haskell type classes.

Finally, in the context of theorem provers like Coq (Sozeau & Oury, 2008) where proof irrelevance typically holds, backtracking seems to be the better choice. If type classes are supplying proofs and it does not matter which proof is found, coherence is not relevant, and the objection about the difficulty of reasoning is also not relevant. Moreover, in theorem proving the expressiveness of search is often more important than having a very fast search method, and thus worse performance is more tolerable.

6.3 Superclasses

Like Scala's implicits design,[8] COCHIS does not directly support superclasses. While superclasses can be encoded in COCHIS to a certain extent, there are several problems.

Superclasses in Haskell Haskell has supported superclasses since the introduction of type classes. The *Eq* and *Ord* classes, with *Eq* a superclass of *Ord*, constitute a standard example. The following simplified code shows the two classes, together with instances for integers:

class $Eq\ \alpha$ **where**
  $(\equiv) :: \alpha \to \alpha \to Bool$
class $Eq\ \alpha \Rightarrow Ord\ \alpha$ **where**
  $(<) :: \alpha \to \alpha \to Bool$

[8] Note that, superclasses are often simulated in Scala with OO Subtyping and class hierarchies, although there is no one-to-one correspondence between both.

instance *Eq Int* **where** ...

instance *Ord Int* **where** ...

In the class context $Eq\ \alpha$ in the definition of the *Ord* class specifies that $Eq\ \alpha$ is a superclass of $Ord\ \alpha$. Superclasses allow the use of methods from the superclasses, even if only the subclass is part of the type class context. For example:

$p :: Ord\ \alpha \Rightarrow \alpha \to \alpha \to Bool$
$p = (\equiv)$ -- accepted because $Eq\ \alpha$ is a superclass of $Ord\ \alpha$

Here the signature of function p assumes that an instance for $Ord\ \alpha$ is available. In the body of p, the method $(\equiv)$ of the class $Eq\ \alpha$ is used. This code is accepted in Haskell because $Eq\ \alpha$ is a superclass of $Ord\ \alpha$.

Superclasses, Determinism and Coherence In the presence of superclasses, Haskell's resolution is not deterministic. Consider this variant of p, defined only for integers:

$p' :: Int \to Int \to Bool$
$p' = (\equiv)$

Haskell has two ways to resolve *Eq Int* to obtain the implementation of $\equiv$ in p'. The first is to get the implementation of *Eq Int* directly from the *Eq Int* instance. The second is to get an implementation of *Eq Int* as the superclass of *Ord Int*. The two elaborations are syntactically different, which makes elaboration with Haskell superclasses non-deterministic. Nonetheless the elaboration is still coherent, since both elaborations yield the same semantics (they both execute the code of the one *Eq Int* instance).

A First Attempt at Encoding Superclasses We can try to encode the previous Haskell definitions in the COCHIS environment $\Gamma = ?(\forall\alpha.Ord\ \alpha \Rightarrow Eq\ \alpha) \rightsquigarrow super, ?(Eq\ Int) \rightsquigarrow x, ?(Ord\ Int) \rightsquigarrow y$, whose implicit entries capture the superclass relation—*super* is a function that extracts the *Eq* dictionary out of an *Ord* dictionary— and the two instances[9]. With respect to COCHIS, the query $?(Eq\ Int)$ resolves deterministically by picking the second entry in Γ. Hence, COCHIS's explicit ordering of implicits avoids Haskell's non-determinism.

While the ordering is beneficial for determinism, fewer queries may succeed due to an unsuitable order of implicits. For example, suppose that we have instead the environment $\Gamma = ?(Eq\ Int) \rightsquigarrow x, ?(\forall\alpha.Ord\ \alpha \Rightarrow Eq\ \alpha) \rightsquigarrow super$, then resolving $?(Eq\ Int)$ fails because the first entry matches and its requirement *Ord Int* cannot be satisfied. In this case the committed choice semantics prevents reaching the second entry, which would resolve *Eq Int* directly.

Superclasses with Committed Choice The situation we just saw also arises in Haskell. Treating superclass relations in the same way as type class instances does not work well with a committed choice strategy. That is why GHC treats superclass relations differently.

[9] Note that the proposed encoding does not satisfy our termination conditions, but we ignore this aspect since there are other more pressing issues with the encoding.

In essence, whenever GHC adds a given type class constraint (e.g., from a programmer-supplied type signature) to the type environment, it uses the superclass relation to also add all of its ancestors. We believe that a similar strategy would be possible for COCHIS. For instance, in the last example, the type environment would be $\Gamma = ?\mathit{Eq\ Int} \rightsquigarrow x$, not contain the superclass relation and be able to resolve the query $?(\mathit{Eq\ Int})$ with x. When adding an *Ord Int* entry, we would also add its *Eq Int* superclass, yielding the modified environment $\Gamma = ?\mathit{Eq\ Int} \rightsquigarrow x, ?\mathit{Ord\ Int} \rightsquigarrow y, ?\mathit{Eq\ Int} \rightsquigarrow \mathit{super}\ y$. Now, the query $?(\mathit{Eq\ Int})$ is resolved with *super y*, which is the superclass value of the new *Ord Int* entry.

Superclasses with Global Scoping and COCHIS-style Resolution in GHC Haskell As we have explained above, standard Haskell gets away with committed choice in the presence of superclasses without unnecessarily rejecting queries. Yet, this is no longer the case for the recent extension with *quantified class constraints* (Bottu *et al.*, 2017). Like COCHIS, quantified class constraints allow nested implications in type class instance contexts and as superclasses. The committed choice approach does not work with the latter. GHC has decided to retain its committed choice approach and to reject queries that would require backtracking to explore the superclass relations.

6.4 Coherence

There are several ways to enforce coherence in a language design. For example, Haskell guarantees coherence by ensuring that there is a unique instance of a type class per type. In this way whenever code accesses a type class dictionary, it always returns the same (equal) dictionary value. This is a very strict way to enforce coherence.

COCHIS's way to achieve coherence is more relaxed than Haskell's. COCHIS enforces that the elaboration and resolution are deterministic but, under different scopes the same queries can resolve to different values (unlike Haskell).

While determinism is sufficient to ensure coherence, it is still a fairly strict way to ensure coherence. A more relaxed and general notion of coherence is to allow elaboration and resolution to have multiple different (but observationally equivalent) terms for the same expression. Our Corollary 5.1 provides a formal statement of coherence that is based on contextual equivalence of two expressions:

$$\forall \Gamma, \rho, E_1, E_2 : \qquad \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_1 \ \wedge\ \Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E_2 \quad \Rightarrow \quad |\Gamma| \vdash_{\mathrm{F}} E_1 \simeq_{\mathrm{ctx}} E_2 : |\rho|$$

This statement is close to the usual definition of coherence in the literature (Reynolds, 1991; Jones, 1992; Breazu-Tannen *et al.*, 1991). That is, E_1 and E_2 are not required to be syntactically equivalent, but they must be semantically equivalent. Many language designs that are coherent are often not necessarily deterministic (unlike COCHIS).

Alternative Designs We use determinism to establish coherence in COCHIS, but a more relaxed notion of coherence would also be possible. For example if, instead of committed choice, we decided to allow for a more powerful resolution strategy (for example with backtracking), then a more relaxed notion of coherence would be helpful. This could be useful to deal with some situations that appear in superclasses. For example, consider again the context $\Gamma = ?(\mathit{Eq\ Int}), ?(\forall \alpha.\mathit{Ord}\ \alpha \Rightarrow \mathit{Eq}\ \alpha), ?(\mathit{Ord\ Int})$. In this case the query $?(\mathit{Eq\ Int})$

can be resolved in two possible ways: either going via the superclass instance *Ord Int*; or by directly using the instance *Eq Int*. Even thought the two elaborations are syntactically different, their semantics is the same.

7 Related Work

The most closely related work can be divided into three strands: IP mechanisms that support local scoping with coherence and stability, but forbid overlapping implicits and lack other types of flexibility; IP mechanisms that have global scoping and preserve coherence and stability; and IP mechanisms that are unstable and sometimes incoherent but offer greater flexibility in terms of local scoping and/or overlapping implicits. COCHIS is unique in offering flexibility (local scoping with overlapping implicits, first-class implicits and higher-order rules), while preserving coherence and stability.

7.1 Implicit Programming with Local Scoping, but no Overlapping Rules

Our work allows a very flexible model of implicits with first-class implicits, higher-order rules and nested scoping with overlapping implicits while guaranteeing coherence and stability. Closest to our work in terms of combining additional flexibility with desirable properties (such as coherence and stability) are *modular type classes* (Dreyer *et al.*, 2007) and System F_G (Siek & Lumsdaine, 2005). Both works preserve coherence and stability in the presence of local scoping, but (unlike COCHIS) the local scopes *forbid overlapping implicits*. The restriction of no overlapping implicits is an essential part of guaranteeing coherence and stability. COCHIS also has several other fundamental differences to both modular type classes and System F_G. *Modular type classes* (Dreyer *et al.*, 2007) are a language design that uses ML-modules to model type classes. The main novelty of this design is that, in addition to explicit instantiation of modules, implicit instantiation is also supported. System F_G (Siek & Lumsdaine, 2005) also offers an implicit parameter passing mechanism with local scoping, which is used for concept-based generic programming (Siek, 2011). Both mechanisms are strongly influenced by type classes, and they preserve some of the characteristics of type classes such as only allowing modules or concepts to be implicitly passed. Moreover neither of those mechanisms support higher-order rules. In contrast COCHIS follows the Scala implicits philosophy and allows values of any type to be implicit, and additionally higher-order rules are supported.

Implicit parameters (Lewis *et al.*, 2000) are a proposal for a name-based implicit parameter passing mechanism with local scoping. Implicit parameters allow *named* arguments to be passed implicitly, and these arguments can be of any type. However, implicit parameters do not support recursive resolution, so for most use-cases of type-classes, including the *Ord* instance for pairs in Section 2.1, implicit parameters would be very cumbersome. They would require manual composition of rules instead of providing automatic recursive resolution. This is in stark contrast with most other IP mechanisms, including COCHIS, where recursive resolution and the ability to compose rules automatically is a key feature and source of convenience.

7.2 Implicit Programming with Local Scoping and Overlapping Implicits

The implicit calculus (Oliveira *et al.*, 2012) is the main inspiration for the design of COCHIS. There are two major differences between COCHIS and the implicit calculus. The first difference is that the implicit calculus, like Scala, does not enforce stability. Programs similar to that in Figure 1 can be written in the implicit calculus and there is no way to detect instability. The second difference is in the design of resolution. Rules in the implicit calculus have n-ary arguments, whereas in COCHIS rules have single arguments and n-ary arguments are simulated via multiple single argument rules. The resolution process with n-ary arguments in the implicit calculus is simple, but quite ad-hoc and forbids certain types of resolution that are allowed in COCHIS. For example, the query:

$$?(\mathit{Char} \Rightarrow \mathit{Bool}), ?(\mathit{Bool} \Rightarrow \mathit{Int}) \vdash^{\mathrm{a}}_{\mathrm{r}} \mathit{Char} \Rightarrow \mathit{Int}$$

does not resolve under the deterministic resolution rules of the implicit calculus, but it resolves in COCHIS. Essentially resolving such query requires adding the rule type's context to the implicit environment in the course of the resolution process. But in the implicit calculus the implicit environment never changes during resolution, which significantly weakens the power of resolution.

Rouvoet (2016) presents $\lambda^{S}_{\Rightarrow}$, which is a variation on the implicit calculus. The key feature of his calculus is the focusing resolution of Figure 4, although Rouvoet does not make the connection with focusing in proof search. As we have already explained in Section 3.5 this approach is both deterministic and unstable.

Scala implicits (Oliveira *et al.*, 2010; Odersky, 2010) were themselves the inspiration for the implicit calculus and, therefore, share various similarities with COCHIS. In Scala implicit arguments can be of any type, and local scoping (including overlapping implicits) is supported. However the original model of Scala implicits did not allow higher-order rules. Recently, following the implicit calculus and a preliminary version of COCHIS, Odersky et al. (2017) presented the SI calculus as a new basis for the Scala language's treatment of implicits. Prominently, SI features implicit function types $T_1 ?\!\!\rightarrow T_2$, which are akin to rule types $T_1 \Rightarrow T_2$ in COCHIS, and implicit queries $?$, which are akin to $?_T$ in COCHIS. There are two main differences with COCHIS. Firstly, like the Hindley-Milner calculus SI is aimed at type inference and, e.g., does not feature explicit abstraction over implicits $\lambda_? T.e$ or type application $e\,T$ at the term level. In contrast, COCHIS is more similar to System F in this sense, making all abstractions and applications explicit.

Secondly, while COCHIS aims to formalise and investigate the meta-theory of resolution, the priority of Odersky et al. is not so much the SI calculus itself as the derived implementation of the Scala compiler. As a consequence, SI features a simplified type system that is incoherent and unstable and a resolution algorithm that supports only monomorphic types, while the compiler's much more complex enforcement of coherence and support for polymorphism are only discussed informally.

An interesting design of implicits has also been created in OCaml (White *et al.*, 2014), where the implicit entities are OCaml modules. Like COCHIS, these implicits have local scope, but, unlike COCHIS, coherence is obtained by performing a backtracking search over all possible ways to resolve an implicit module signature, and fail if there is not exactly one way. Hence, while introducing overlapping implicits is allowed, they are only usable

if only one leads to a full resolution. As far as we know, a partial prototype exists but the approach has not been formalised yet.

7.3 Implicit Programming with Global Scoping

Several core calculi and refinements have been proposed in the context of type classes. As already discussed in detail in Section 1, there are a number of design choices that (Haskell-style) type classes take that are different from COCHIS. Most prominently, type classes make a strong differentiation between types and type classes, and they use global scoping instead of local scoping for instances/implicits. The design choice for global scoping can be traced back to Wadler and Blott's (1989) original paper on type classes. They wanted to extend Hindley-Milner type-inference (Hindley, 1969; Milner, 1978; Damas & Milner, 1982) and discovered that local instances resulted in the loss of principal types. For Haskell-like languages the preservation of principal types is very important, so local instances were discarded. However, there are many languages with IP mechanisms (including Scala, Coq, Agda, Idris or Isabelle) that have more modest goals in terms of type-inference. In these languages there are usually enough type annotations such that ambiguity introduced by local instances is avoided.

There have been some proposals for addressing the limitations that arise from global scoping (Kahl & Scheffczyk, 2001; Dijkstra & Swierstra, 2005) in the context of Haskell type classes. Both *named instances* (Kahl & Scheffczyk, 2001) and *Explicit Haskell* (Dijkstra & Swierstra, 2005) preserve most design choices taken in type classes (including global scoping), but allow instances that do not participate in the automatic resolution process to be named. This enables the possibility of overriding the compiler's default resolution result with a user-defined choice.

Jones's work on *qualified types* (Jones, 1995b) provides a particularly elegant framework that captures type classes and other forms of predicates on types. Like type classes, qualified types make a strong distinction between types and predicates over types, and scoping is global. Jones (1995a) discusses the coherence of qualified types. The formal statement of determinacy in COCHIS essentially guarantees a strong form of coherence similar to the one used in qualified types.

The GHC Haskell compiler supports overlapping instances (Peyton Jones *et al.*, 1997) that live in the same global scope. This allows some relief for the lack of local scoping. Yet, it still does not allow different instances for the same type to coexist in different scopes of a program. Moreover, GHC only commits to an instance if it is the only one whose head matches, and does not backtrack among multiple matching instances.

COCHIS has a different approach to overlapping compared to *instance chains* (Morris & Jones, 2010). With instance chains the programmer imposes an order on a set of overlapping type class instances. All instance chains for a type class have a global scope and are expected not to overlap. This makes the scope of overlapping closed within a chain. Instance chains can explicitly declare cases when resolution should fail, and these are dealt with by backtracking. In our calculus, we make our local scope closed, thus overlap only happens within one nested scope. More recently, there has been a proposal for *closed type families with overlapping equations* (Eisenberg *et al.*, 2014). This proposal allows the declaration of a type family and a (closed) set of instances. After this declaration no more

instances can be added. In contrast our notion of scoping is closed at a particular resolution point, but the scopes can still be extended at other resolution points.

Inspired by the focusing approach of COCHIS Bottu et al. (2017) have extended Haskell's type class inference with *quantified class constraints*. This generalizes the syntax of Haskell's type class constraints to feature arbitrarily nested uses of universal quantification and implication. Their work differs from COCHIS in that it does not support local instances. Moreover, they achieve coherence through requiring non-overlapping instances. Their algorithm performs a backtracking search among these instances as well as any local assumptions (which themselves can ultimately only be satisfied by combinations of global instances), rather than a linear committed-choice traversal of the environment.

IP Mechanisms in Dependently Typed Programming A number of dependently typed languages also have IP mechanisms inspired by type classes. Coq's type classes (Sozeau & Oury, 2008) and canonical structures (Gonthier *et al.*, 2011), Agda's instance arguments (Devriese & Piessens, 2011) and Idris type classes (Brady, 2015) all allow multiple and/or highly overlapping implicits/instances that can be incoherent. The Coq theorem prover has two mechanisms that allow modelling type-class like structures: *canonical structures* (Gonthier *et al.*, 2011) and *type classes* (Sozeau & Oury, 2008). The two mechanisms have quite a bit of overlap in terms of functionality. In both mechanisms the idea is to use dependent records to model type-class-like structures, and pass instances of such records implicitly, but they still follow Haskell's global scoping approach. Nevertheless highly overlapping instances, which can be incoherent, are allowed and resolution performs a backtracking search. Like implicits, the design of Idris type classes, known as *interfaces*, allows for any type of value to be implicit. Thus type classes in Idris are first-class and can be manipulated like any other values. The language distinguishes unnamed instances, which are used for resolution, and named instances which have to be applied explicitly. The former cannot be overlapping, while there can be multiple (incoherent) named instances of the same type. The implicit resolution follows the committed choice strategy of Haskell, and ignores the fact that named instances can distinguish between alternative derivations. *Instance arguments* (Devriese & Piessens, 2011) are an Agda extension that is closely related to implicits. Like COCHIS, instance arguments use a special arrow for introducing implicit arguments. However, unlike most other mechanisms, implicits are not declared explicitly. Instead they are drawn directly from the regular type environment, and any previously defined declaration can be used as an implicit. The original design of instance arguments severely restricted the power of resolution by forbidding recursive resolution. Since then, recursive resolution has been enabled in Agda. Like Coq's and Idris's type classes, instance arguments allow multiple incoherent implicits. Agda computes all possible resolutions and uses one of them only if all are equal.

7.4 Global Uniqueness and Same Instance Guarantee

Haskell type classes not only ensure coherence but also *global uniqueness* (Zhang, 2014) (due to global scoping), as discussed in Section 2.2. Unrestricted COCHIS programs ensure coherence only, as multiple implicits that match the same type can coexist in the same program. We agree that for programs such as the *Set* example, it is highly desirable to

ensure that the same ordering instance is used consistently. COCHIS is a core calculus, meant to enable the design of source languages that utilize its power. An example are Bottu et al.'s (2017) quantified class constraints for Haskell, which forbid local scoping constructs and, instead, make all declared implicits visible in a single global environment. This retains several of the benefits of COCHIS (such as first-class implicits, higher-order rules, and coherent overlapping implicits), while providing a form of global uniqueness. However this design is still essentially non-modular, which is a key motivation for many alternatives to type classes to provide local scoping instead.

Global uniqueness of instances is just a sufficient property to ensure consistent uses of the same instances for examples like *Set*. However, the important point is not to have global uniqueness, but to consistently use the same instance. COCHIS admittedly does not provide a solution to enforce such consistency, but there is existing work with an alternative solution to deal with the problem. Genus (Zhang *et al.*, 2015) tracks the types of instances to enforce their consistent use. For instance, in Genus two sets that use different orderings have different types that reflect which *Ord* instance they use. As a consequence, taking the union of those two sets is not possible. In contrast to COCHIS Genus is focused on providing a robust source language implementation for generic programming. Although the Genus authors have proved some meta-theoretic results, neither type-safety nor coherence have been proved for Genus. In dependently typed languages such as Agda and Idris, it is possible to parametrize types by the instances they use (Brady, 2015). This achieves a similar outcome to Genus's approach to consistent usage of instances. Investigating the applicability of a similar approach to COCHIS is an interesting line of future work.

7.5 Focused Proof Search

Part of the syntax-directedness of our deterministic resolution is very similar to that obtained by *focusing* in proof search (Andreoli, 1992; Miller *et al.*, 1991; Liang & Miller, 2009). Both approaches alternate a phase that is syntax directed on a "query" formula (our focusing judgment), with a phase that is syntax directed on a given formula (our matching judgment). This is as far as the correspondence goes though, as the choice of given formula to focus on is typically not deterministic in focused proof search.

8 Conclusion

This paper presented COCHIS, the Calculus Of CoHerent ImplicitS, a new calculus for implicit programming that improves upon the implicit calculus and strikes a good balance between flexibility and desirable properties such as coherence and stability. COCHIS supports local scoping, overlapping implicits, first-class implicits, and higher-order rules, while remaining type safe, coherent and stable. Interesting future work includes integrating Genus's solution for the instance coherence problem (Zhang *et al.*, 2015) in COCHIS; and adding more features that show up in various IP mechanisms, such as *associated types* (Chakravarty *et al.*, 2005b; Chakravarty *et al.*, 2005a) and *type families* (Schrijvers *et al.*, 2008).

Acknowledgments We are very grateful for the feedback of the anonymous reviewers. This work was partially supported by the the Hong Kong Research Grant Council projects number 27200514 and 17258816, and by the Flemish Fund for Scientific Research (FWO).

References

Andreoli, Jean-marc. (1992). Logic programming with focusing proofs in linear logic. *Journal of logic and computation*, **2**, 297–347.

Barendregt, H. (1981). *The lambda calculus: its syntax and semantics, volume 103 of studies in logic and the foundations of mathematics*. North-Holland.

Barendregt, Henk, Dekkers, Wil, & Statman, Richard. (2013). *Lambda calculus with types*. Cambridge University Press.

Biernacki, Dariusz, & Polesiuk, Piotr. (2018). Logical relations for coherence of effect subtyping. *Logical methods in computer science*, **14**(1).

Boehm, Hans-J. (1985). Partial polymorphic type inference is undecidable. *Pages 339–345 of: 26th annual symposium on foundations of computer science*. IEEE.

Bottu, Gert-Jan, & simonpj. (2018). *Quantified constraints - proposals*. https://github.com/Gertjan423/ghc-proposals/blob/quantified-constraints/proposals/0000-quantified-constraints.rst#overlap.

Bottu, Gert-Jan, Karachalias, Georgios, Schrijvers, Tom, Oliveira, Bruno, & Wadler, Philip. (2017). Quantified class constraints. *Acm sigplan haskell symposium*.

Brady, Edwin. (2015). *Type classes in Idris*. https://groups.google.com/forum/#!topic/idris-lang/OQQ3oc6zBaM.

Breazu-Tannen, Val, Coquand, Thierry, Gunter, Carl, & Scedrov, Andre. (1991). Inheritance as implicit coercion. *Information and computation*, **93**, 172–221. Also in .

Camarão, C., & Figueiredo, L. (1999). Type inference for overloading without restrictions, declarations or annotations. *Pages 37–52 of: Flops*. London, UK: Springer-Verlag.

Chakravarty, M., Keller, G., & Peyton Jones, S. L. (2005a). Associated type synonyms. *Pages 241–253 of: Icfp*. New York, NY, USA: ACM.

Chakravarty, M., Keller, G., Peyton Jones, S. L., & Marlow, S. (2005b). Associated types with class. *Pages 1–13 of: Popl*. New York, NY, USA: ACM.

Chong, Stephen. (2017). *Lecture 2: Logical relations part 2*. https://www.seas.harvard.edu/courses/cs252/2017fa/lectures/lec02-contextual-equiv.pdf.

Damas, Luís, & Milner, Robin. (1982). Principal type-schemes for functional programs. *Pages 207–212 of: Popl*. New York, NY, USA: ACM.

Devriese, D., & Piessens, F. (2011). On the bright side of type classes: Instance arguments in Agda. *Pages 143–155 of: Icfp*. New York, NY, USA: ACM.

Dijkstra, A., & Swierstra, S. D. (2005). *Making implicit parameters explicit*. Tech. rept. Utrecht University.

Dreyer, D., Harper, R., Chakravarty, M., & Keller, G. (2007). Modular type classes. *Pages 63–70 of: Popl*. New York, NY, USA: ACM.

Dunfield, Joshua, & Krishnaswami, Neelakantan R. (2013). Complete and easy bidirectional typechecking for higher-rank polymorphism. *Pages 429–442 of:* Morrisett, Greg, & Uustalu, Tarmo (eds), *ACM SIGPLAN international conference on functional programming, icfp'13, boston, ma, USA - september 25 - 27, 2013*. ACM.

Eisenberg, Richard A., Vytiniotis, Dimitrios, Peyton Jones, Simon, & Weirich, Stephanie. (2014). Closed type families with overlapping equations. *Pages 671–683 of: Popl*. New York, NY, USA: ACM.

Garcia, Ronald, Jarvi, Jaakko, Lumsdaine, Andrew, Siek, Jeremy, & Willcock, Jeremiah. (2007). An extended comparative study of language support for generic programming. *J. funct. program.*, **17**(2).

GHC. (2017). *Ghc users guide: Overlapping instances*. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#overlapping-instances.

GHC. (2017). *Quantified Constraints – GHC*. https://ghc.haskell.org/trac/ghc/wiki/QuantifiedConstraints.

Gonthier, Georges, Ziliani, Beta, Nanevski, Aleksandar, & Dreyer, Derek. (2011). How to make ad hoc proof automation less ad hoc. *Pages 163–175 of: Proceedings of the 16th acm sigplan international conference on functional programming*. ICFP '11. New York, NY, USA: ACM.

Gregor, D., Järvi, J., Siek, J. G., Stroustrup, B., Dos Reis, G., & Lumsdaine, A. (2006). Concepts: linguistic support for generic programming in c++. *Pages 291–310 of: Oopsla*. New York, NY, USA: ACM.

Hindley, J. Roger. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, **146**, 29–60.

Hulley, Brian. (2009). *A show-stopping problem for modular type classes?* http://lists.seas.upenn.edu/pipermail/types-list/2009/001405.html.

Jones, M. P. (1992). A theory of qualified types. *Esop*.

Jones, M. P. (1995a). *Qualified types: theory and practice*. Cambridge University Press.

Jones, M. P. (1995b). Simplifying and improving qualified types. *Fpca*.

Jones, Simon Peyton, Vytiniotis, Dimitrios, Weirich, Stephanie, & Shields, Mark. (2007). Practical type inference for arbitrary-rank types. *Journal of functional programming*, **17**(1), 1–82.

Kahl, W., & Scheffczyk, J. (2001). Named instances for Haskell type classes. *Haskell workshop*.

Kiselyov, Oleg, & Shan, Chung-chieh. (2004). Functional pearl: Implicit configurations–or, type classes reflect the values of types. *Proceedings of the 2004 acm sigplan workshop on haskell*. Haskell '04.

Kmett, Edward. (2015). *Type classes vs the world*. https://www.youtube.com/watch?v=hIZxTQP1ifo.

Le Botlan, Didier, & Rémy, Didier. (2003). MLF: Raising ML to the power of system F. *Proceedings of the eighth acm sigplan international conference on functional programming*. ICFP '03.

Leijen, Daan. (2008). HMF: Simple type inference for first-class polymorphism. *Proceedings of the 13th acm sigplan international conference on functional programming*. ICFP '08.

Lewis, J., Launchbury, J., Meijer, E., & Shields, M. (2000). Implicit parameters: dynamic scoping with static types. *Pages 108–118 of: Popl*. New York, NY, USA: ACM.

Liang, Chuck, & Miller, Dale. (2009). Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. comput. sci.*, **410**(46), 4747–4768.

Manna, Zohar, & Waldinger, Richard. (1980). A deductive approach to program synthesis. *Acm trans. program. lang. syst.*, **2**(1), 90–121.

Marlow, Simon. (2012). *Safe haskell and instance coherence*. https://mail.haskell.org/pipermail/haskell-cafe/2012-October/103887.html.

Martelli, Alberto, & Montanari, Ugo. (1982). An efficient unification algorithm. *Acm trans. program. lang. syst.*, **4**(2), 258–282.

Miller, Dale. (1992). Unification under a mixed prefix. *J. symb. comput.*, **14**(4), 321–358.

Miller, Dale, Nadathur, Gopalan, Pfenning, Frank, , & Scedrov, Andre. (1991). Uniform proofs as a foundation for logic programming. *Annals of pure and applied logic*, **51**(1–2), 125–157.

Milner, Robin. (1978). A theory of type polymorphism in programming. *J. comput. syst. sci.*, **17**(3), 348–375.

Morris, J. G., & Jones, M. P. (2010). Instance chains: type class programming without overlapping instances. *Pages 375–386 of: Icfp*. New York, NY, USA: ACM.

Morris, James Hiram, Jr. (1969). *Lambda-calculus models of programming languages*. Ph.D. thesis, Sloan School of Management.

Mozilla Research, Team. (2017). *The rust programming language*. https://www.rust-lang.org/en-US/.

Norell, Ulf. (2008). Dependently typed programming in agda. *Pages 230–266 of: Advanced functional programming*.

nponeccop. (2012). *Breaking data.set integrity without generalizednewtypederiving*. https://stackoverflow.com/questions/12735274/breaking-data-set-integrity-without-generalizednewtypederiving.

Odersky, M. (2010). *The Scala language specification, version 2.8*.

Odersky, Martin, & Läufer, Konstantin. (1996). Putting type annotations to work. *Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages*. POPL '96.

Odersky, Martin, Blanvillain, Olivier, Liu, Fengyun, Biboudis, Aggelos, Miller, Heather, & Stucki, Sandro. (2017). Simplicitly: Foundations and applications of implicit function types. *Proc. acm program. lang.*, **2**(POPL), 42:1–42:29.

oleg. (2006). *Typeclass vs. prolog programming*. https://mail.haskell.org/pipermail/haskell-cafe/2006-September/018500.html.

Oliveira, B. C. d. S., Moors, A., & Odersky, M. (2010). Type classes as objects and implicits. *Oopsla*. New York, NY, USA: ACM.

Oliveira, B. C. d. S., Schrijvers, T., Choi, W., Lee, W., & Yi, K. (2012). The implicit calculus: A new foundation for generic programming. *Pldi '12*. New York, NY, USA: ACM.

Peyton Jones, S. L., Jones, M. P., & Meijer, E. (1997). Type classes: exploring the design space. *Haskell workshop*.

Pfenning, Frank. (1993). On the undecidability of partial polymorphic type reconstruction. *Fundam. inform.*, **19**(1/2), 185–199.

Pierce, Benjamin C. (2002). *Types and programming languages*. Cambridge, MA, USA: MIT Press.

Reynolds, John C. (1991). The coherence of languages with intersection types. *Pages 675–700 of: Proceedings of the international conference on theoretical aspects of computer software*. TACS '91. London, UK, UK: Springer-Verlag.

rHaskell. (2006). *The constraint trick for instances*. https://www.reddit.com/r/haskell/comments/3afi3t/the_constraint_trick_for_instances/cscb33j/?st=jjcz7zeh&sh=1e5f3c8b.

Rouvoet, Arjen. (2016). *Programs for Free: Towards the Formalization of Implicit Resolution in Scala*. M.Phil. thesis, TU Delft, the Netherlands.

Scharli, Nathanael, Ducasse, Stéphane, Nierstrasz, Oscar, & Black, Andrew P. (2003). Traits: Composable units of behaviour. *Pages 248–274 of: European conference on object-oriented programming (ecoop 2003)*. Springer.

Schrijvers, T., Peyton Jones, S. L., Chakravarty, M., & Sulzmann, M. (2008). Type checking with open type functions. *Icfp*. New York, NY, USA: ACM.

Siek, J. G. (2011). *The C++0x Concepts Effort*. http://ecee.colorado.edu/~siek/concepts_effort.pdf.

Siek, J. G., & Lumsdaine, A. (2005). Essential language support for generic programming. *Pldi*. New York, NY, USA: ACM.

Sozeau, M., & Oury, N. (2008). First-class type classes. *Tphols*.

Sulzmann, M., Duck, G., Peyton Jones, S. L., & Stuckey, P. J. (2007). Understanding functional dependencies via Constraint Handling Rules. *Journal of functional programming*, **17**, 83–129.

Vytiniotis, Dimitrios, Weirich, Stephanie, & Peyton Jones, Simon. (2008). FPH: First-class polymorphism for Haskell. *Proceedings of the 13th acm sigplan international conference on functional programming*. ICFP '08.

Vytiniotis, Dimitrios, Peyton Jones, Simon, Schrijvers, Tom, & Suzmann, Martin. (2011). OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of functional programming*, **21**(4-5), 333–412.

Wadler, P. L., & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. *Popl*. New York, NY, USA: ACM.

Wadler, Philip. (2015). Propositions as types. *Commun. ACM*, **58**(12), 75–84.

Wehr, S., Lämmel, R., & Thiemann, P. (2007). JavaGI: Generalized interfaces for java. *Ecoop*.

White, Leo, Bour, Frédéric, & Yallop, Jeremy. (2014). Modular implicits. *Pages 22–63 of:* Kiselyov, Oleg, & Garrigue, Jacques (eds), *Proceedings ML family/ocaml users and developers workshops, ml/ocaml 2014, gothenburg, sweden, september 4-5, 2014*. EPTCS, vol. 198.

Zhang, Edward. (2014). *Type classes: confluence, coherence and global uniqueness*. http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/.

Zhang, Yizhou, Loring, Matthew C., Salvaneschi, Guido, Liskov, Barbara, & Myers, Andrew C. (2015). Lightweight, flexible object-oriented generics. *Proceedings of the 36th acm sigplan conference on programming language design and implementation*. PLDI '15.

$$\boxed{\Gamma \vdash_{\mathrm{F}} E : T}$$

$$\text{(F-VAR)}\ \frac{(x:T) \in \Gamma}{\Gamma \vdash_{\mathrm{F}} x : T}$$

$$\text{(F-ABS)}\ \frac{\Gamma, x : T_1 \vdash_{\mathrm{F}} E : T_2}{\Gamma \vdash_{\mathrm{F}} \lambda x : T_1 . E : T_1 \to T_2}$$

$$\text{(F-APP)}\ \frac{\Gamma \vdash_{\mathrm{F}} E_1 : T_2 \to T_1 \qquad \Gamma \vdash_{\mathrm{F}} E_2 : T_2}{\Gamma \vdash_{\mathrm{F}} E_1\, E_2 : T_1}$$

$$\text{(F-TAPP)}\ \frac{\Gamma \vdash_{\mathrm{F}} E : \forall \alpha . T_2}{\Gamma \vdash_{\mathrm{F}} E\, T_1 : T_2[T_1/\alpha]}$$

$$\text{(F-TABS)}\ \frac{\Gamma, \alpha \vdash_{\mathrm{F}} E : T}{\Gamma \vdash_{\mathrm{F}} \Lambda \alpha . E : \forall \alpha . T}$$

Fig. 11. System F Type System

A Proofs

This appendix summarizes the proofs for the theorems presented in the article as well as those of the auxiliary lemmas that feature in those proofs. Several of the proofs have been mechanized in Coq; they are available from https://bitbucket.org/KlaraMar/cochiscoq and this appendix refers to them where appropriate.

Throughout the proofs we refer to the type system rules of System F listed in Figure 11, also formalized in file systemF.v (inductive definition `Typing`).

A.1 Type Preservation

Lemma A.1 states that the translation of expressions to System F preserves types. Its proof relies on Lemma A.2, which states that the translation of resolution preserves types.

Lemma A.1. *If*

$$\Gamma \vdash e : \rho \rightsquigarrow E$$

then

$$|\Gamma| \vdash_{\mathrm{F}} E : |\rho|$$

Proof. See file Cochis.v, Lemma `Typing_TypePreservation`. □

Lemma A.2. *If*

$$\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow e$$

then

$$|\Gamma| \vdash_{\mathrm{F}} E : |\rho|$$

Proof. See file Cochis.v, Lemma `amb_res_TypePreservation`. □

A.2 Auxiliary Lemmas About Non-Determinstic Resolution

The non-deterministic resolution judgment enjoys a number of typical binder-related properties.

The first lemma is the weakening lemma: it states that an extended type environment preserves all the derivations of the original environment.

Lemma A.3 (Weakening)**.** *If*

$$\Gamma, \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$$

then

$$\Gamma, \Gamma'', \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$$

Proof. See file Cochis.v, Lemma `amb_res_weaken`. □

The second lemma is the substitution lemma which states that we can drop an axiom from the type environment if it is already implied by the remainder of the type environment.

Lemma A.4 (Substitution)**.** *If*

$$\Gamma, ?\rho \rightsquigarrow x, \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho' \rightsquigarrow E'$$

and

$$\Gamma \vdash^{\mathrm{a}}_{\mathrm{r}} \rho \rightsquigarrow E$$

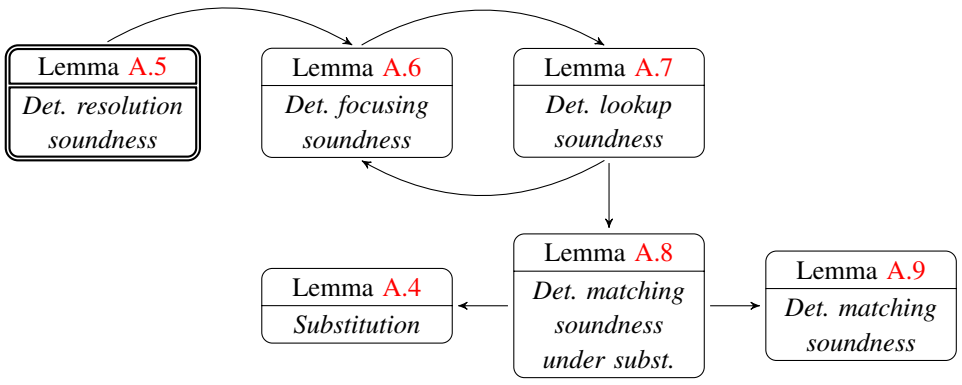
then

$$\Gamma, \Gamma' \vdash^{\mathrm{a}}_{\mathrm{r}} \rho' \rightsquigarrow E'[E/x]$$

Proof. See file Cochis.v, Lemma `sub_eimpl_amb_res`. □

A.3 Soundness of Deterministic Resolution

The proofs of this section are split into subproofs for each sub-judgment (focusing, lookup and matching) of the deterministic resolution judgment. They all proceed by induction on the corresponding deterministic resolution judgment. Therefore, their dependencies, depicted in the diagram below, follow those of the sub-judgments that constitute the main deterministic resolution judgment. Because the proofs of this section are mechanized, an auxiliary lemma is necessary to handle the multiple substitutions that rule (L-MATCH) uses.



Lemma A.5 states that deterministic resolution is sound with respect to non-deterministic resolution.

Lemma A.5. *If*

$$\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$$

then

$$\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow E$$

Proof. See file CochisDet.v, Lemma `DRes_Soundness`.

The lemma immediately follows from Lemma A.6. □

Lemma A.6. *If*

$$\bar{\alpha}; \Gamma \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$$

then

$$\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow E$$

Proof. See file CochisDet.v, Lemma `DRes_focus_Soundness`.

This lemma is proved together with Lemma A.7 by mutual induction on the derivation of the first hypothesis of both lemmas. □

Lemma A.7. *If*

$$\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$$

then

$$\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \tau \rightsquigarrow E$$

Proof. See file CochisDet.v, Lemma `DRes_lookup_Soundness`.

The proof proceeds by induction on the derivation, mutually with the previous proof. □

The above proof relies on the following lemma.

Lemma A.8. *If*

$$\Gamma; [\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \bar{\rho} \rightsquigarrow \bar{x}; \tau \rightsquigarrow E'$$

and

$$\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \rho \rightsquigarrow E$$

and

$$\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \bar{\rho} \rightsquigarrow \bar{E}$$

then

$$\Gamma \vdash_{\mathrm{r}}^{\mathrm{a}} \tau \rightsquigarrow E'[\bar{E}/\bar{x}]$$

Proof. See file CochisDet.v, Lemma `DRes_match_Soundness`.

The proof proceeds by induction on the derivation of the first assumption and relies on Lemmas A.4 (Substitution) and A.9. □

$$\boxed{\vdash_{\text{unamb}} \Gamma}$$

$$\text{(UE-EMPTY)}\ \frac{}{\vdash_{\text{unamb}} \varepsilon} \qquad \text{(UE-VAR)}\ \frac{\vdash_{\text{unamb}} \Gamma}{\vdash_{\text{unamb}} \Gamma, x : \rho}$$

$$\text{(UE-TVAR)}\ \frac{\vdash_{\text{unamb}} \Gamma}{\vdash_{\text{unamb}} \Gamma, \alpha} \qquad \text{(UE-IMPL)}\ \frac{\vdash_{\text{unamb}} \Gamma \qquad \vdash_{\text{unamb}} \rho}{\vdash_{\text{unamb}} \Gamma, ?\rho \rightsquigarrow x}$$

Fig. A 1. Unambiguous Type Environment

Lemma A.9. *If*

$$\Gamma; [\rho] \rightsquigarrow E \vdash_{\text{r}} \bar{\rho} \rightsquigarrow \bar{x}; \tau \rightsquigarrow E'$$

and

$$\Gamma \vdash^{\text{a}}_{\text{r}} \rho \rightsquigarrow E$$

then

$$\Gamma, ?\bar{\rho} \rightsquigarrow \bar{x} \vdash^{\text{a}}_{\text{r}} \tau \rightsquigarrow E'$$

Proof. See file CochisDet.v, Lemma `DRes_match_impl_vars`. □

A.4 Deterministic Resolution is Deterministic

Similarly to Section A.3, the proofs of this section are split into subproofs for each sub-judgment (focusing, lookup and matching) of the deterministic resolution judgment. Their dependencies, depicted in the diagram below, follow those of the sub-judgments that constitute the main deterministic resolution judgment.

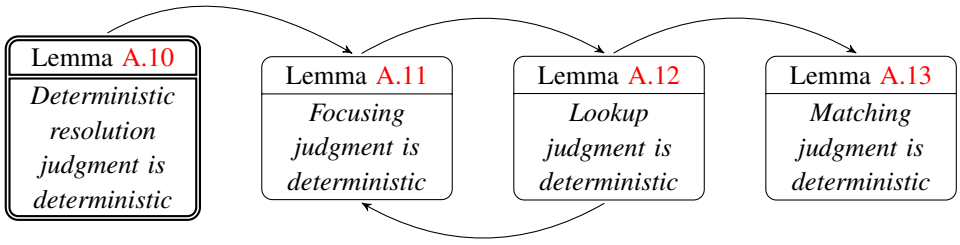


Figure A 1 defines the judgment $\vdash_{\text{unamb}} \Gamma$ which states that a typing environment is unambiguous. Essentially, the definition requires that all types ρ in Γ are unambiguous. This definition will be used throughout this section.

Lemma A.10. *If*

$$\vdash_{\text{unamb}} \Gamma$$

and

$$\vdash_{\text{unamb}} \rho$$

and

$$\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_1$$

and

$$\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E_2$$

then

$$E_1 = E_2$$

Proof. From the lemma's third and fourth hypotheses, the hypothesis of rule (R-MAIN) and Lemma A.11 the desired result follows

$$E_1 = E_2$$

□

Lemma A.11. *If*

$$\vdash_{\text{unamb}} \Gamma$$

and

$$\vdash_{\text{unamb}} \rho$$

and

$$\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E_1$$

and

$$\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E_2$$

then

$$E_1 = E_2$$

Proof. The proof proceeds by induction on the derivation of the third hypothesis.

(R-IABS) $\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho_1 \Rightarrow \rho_2] \rightsquigarrow \lambda x : |\rho_1| . E_1$

It follows that the lemma's fourth hypothesis is also derived by rule (R-IABS). It follows from the lemma's second hypothesis that

$$\vdash_{\text{unamb}} \rho_1 \quad \wedge \quad \vdash_{\text{unamb}} \rho_2$$

From this and lemma's first hypothesis, it follows that

$$\vdash_{\text{unamb}} \Gamma, ?\rho_1 \rightsquigarrow x$$

From the rule's hypothesis and the induction hypothesis, it follows that

$$E_1 = E_2$$

Hence, we may conclude

$$\lambda x : |\rho_1|.E_1 = \lambda x : |\rho_1|.E_2$$

(R-TABS) $\bar{\alpha};\Gamma \vdash_{\mathrm{r}} [\forall \alpha.\rho] \rightsquigarrow \Lambda\alpha.E_1$

It follows that the lemma's fourth hypothesis is also derived by rule (R-TABS). It follows from the lemma's second hypothesis that

$$\alpha \vdash_{\mathrm{unamb}} \rho$$

The unambiguity judgment enjoys a weakening property which we use here to obtain

$$\vdash_{\mathrm{unamb}} \rho$$

From the lemma's first hypothesis, it follows that

$$\vdash_{\mathrm{unamb}} \Gamma, \alpha$$

From the rule's hypothesis and the induction hypothesis, it follows that

$$E_1 = E_2$$

Hence, we may conclude

$$\Lambda\alpha.E_1 = \Lambda\alpha.E_2$$

(R-SIMP) $\bar{\alpha};\Gamma \vdash_{\mathrm{r}} [\tau] \rightsquigarrow E_1$

It follows that the lemma's fourth hypothesis is also derived by rule (R-SIMP). We obtain the desired result from Lemma A.12

$$E_1 = E_2$$

□

Lemma A.12. *If*

$$\vdash_{\mathrm{unamb}} \Gamma$$

and

$$\vdash_{\mathrm{unamb}} \Gamma'$$

and

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E_1$$

and

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E_2$$

then

$$E_1 = E_2$$

Proof. The proof proceeds by induction on the derivation of the third hypothesis.

(L-MATCH) $\bar{\alpha};\Gamma;[\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E_1[\bar{E}'_1/\bar{x}]$

The rule's two hypotheses are

$$\Gamma;[\rho] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma_1;\tau \rightsquigarrow E_1$$

where $\Sigma_1 = \overline{\rho_1' \rightsquigarrow x}$, and

$$\bar{\alpha};\Gamma \vdash_{\mathrm{r}} [\rho_1'] \rightsquigarrow E_1' \quad (\forall \rho_1' \in \overline{\rho}_1')$$

Then the fourth hypothesis was either derived from rule (L-MATCH), or from rule (L-NOMATCH). However, the hypothesis of the latter is not satisfied: $\varepsilon;E_1;\Sigma_1$ forms a counter-example. Hence, using rule (L-MATCH) the fourth hypothesis is also derived. Then it follows from the first hypothesis of the rule (given above), from the first hypothesis of the lemma, which entails $\vdash_{\text{unamb}} \rho$, and from Lemma A.13 (with $\bar{\sigma}_1 = \bar{\sigma}_2 = \bar{\sigma}_1' = \bar{\sigma}_2' = \varepsilon$) that

$$E_1 = E_2 \quad \wedge \quad \Sigma_1 = \Sigma_2$$

From the second hypothesis of the rule and Lemma A.11 it also follows that

$$\bar{E}_1' = \bar{E}_2'$$

Hence, we may conclude

$$E_1[\bar{E}_1'/\bar{x}] = E_2[\bar{E_2}'/\bar{x}]$$

(L-NOMATCH) $\quad \bar{\alpha};\Gamma;[\Gamma', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E_1'$

Then the fourth hypothesis was either derived from rule (L-MATCH), or from rule (L-NOMATCH). However, the hypothesis of the former is not satisfied, as it would be a counter-example for the first hypothesis of the assumed rule of the third hypothesis. Hence, the fourth hypothesis is also formed by rule (L-NOMATCH).
From the second hypothesis of the lemma we derive $\vdash_{\text{unamb}} \Gamma'$. Then from the second hypothesis of the rule and the induction hypothesis we conclude the desired result

$$E_1' = E_2'$$

(L-VAR) $\quad \bar{\alpha};\Gamma;[\Gamma', x:\rho] \vdash_{\mathrm{r}} \tau \rightsquigarrow E_1$

Clearly the fourth hypothesis is also derived by rule (L-VAR). Moreover, from the second hypothesis it follows that $\vdash_{\text{unamb}} \Gamma'$. Hence, from the induction hypothesis we conclude that

$$E_1 = E_2$$

(L-TYVAR) $\quad \bar{\alpha};\Gamma;[\Gamma', \alpha] \vdash_{\mathrm{r}} \tau \rightsquigarrow E_1$

Clearly the fourth hypothesis is also derived by rule (L-TYVAR). Moreover, from the second hypothesis it follows that $\vdash_{\text{unamb}} \Gamma'$. Hence, from the induction hypothesis we conclude that

$$E_1 = E_2$$

□

Before proceeding with the matching judgment, we present a version of it that is annotated with the sequence of substitution types $\bar{\sigma}$ used to instantiate the universal quantifiers.

$$\boxed{\bar{\sigma};\Gamma;[\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma;\tau \rightsquigarrow E'}$$

$$\text{(M-SIMP)}\ \varepsilon;\Gamma;[\tau] \rightsquigarrow E \vdash_{\mathrm{r}} \varepsilon;\tau \rightsquigarrow E$$

$$\text{(M-IAPP)}\ \frac{\bar{\sigma};\Gamma,?\rho_1 \rightsquigarrow x;[\rho_2] \rightsquigarrow E\,x \vdash_{\mathrm{r}} \Sigma;\tau \rightsquigarrow E' \qquad x\ \mathit{fresh}}{\bar{\sigma};\Gamma;[\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash_{\mathrm{r}} \rho_1 \rightsquigarrow x,\Sigma;\tau \rightsquigarrow E'}$$

$$\text{(M-TAPP)}\ \frac{\bar{\sigma};\Gamma;[\rho[\sigma/\alpha]] \rightsquigarrow E\,|\sigma| \vdash_{\mathrm{r}} \Sigma;\tau \rightsquigarrow E' \qquad \Gamma \vdash \sigma}{\bar{\sigma},\sigma;\Gamma;[\forall\alpha.\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma;\tau \rightsquigarrow E'}$$

It is not difficult to see that any derivation of the annotated judgment is in one to one correspondence with a derivation of the unannotated judgment.

Now we are ready to show that the judgment is deterministic.

Lemma A.13. *If*

$$\bar{\alpha} \vdash_{\mathrm{unamb}} \rho$$

and

$$\bar{\sigma}_1;\Gamma;[\rho[\bar{\sigma}_2/\bar{\alpha}]] \rightsquigarrow E[|\bar{\sigma}_2|/\bar{\alpha}] \vdash_{\mathrm{r}} \Sigma_1;\tau \rightsquigarrow E_1$$

and

$$\bar{\sigma}'_1;\Gamma;[\rho[\bar{\sigma}'_2/\bar{\alpha}]] \rightsquigarrow E[|\bar{\sigma}'_2|/\bar{\alpha}] \vdash_{\mathrm{r}} \Sigma_2;\tau \rightsquigarrow E_2$$

then

$$\bar{\sigma}_1 = \bar{\sigma}'_1 \quad\wedge\quad \bar{\sigma}_2 = \bar{\sigma}'_2 \quad\wedge\quad E_1 = E_2 \quad\wedge\quad \Sigma_1 = \Sigma_2 \quad\wedge\quad \vdash_{\mathrm{unamb}} \Sigma_1$$

Proof. The proof proceeds by induction on the derivation of the first hypothesis.

(UA-SIMP) $\bar{\alpha} \vdash_{\mathrm{unamb}} \tau'$

Then the second and third hypothesis of the lemma must have been formed by rule (M-SIMP) and hence

$$\bar{\sigma}_1 = \varepsilon = \bar{\sigma}'_1$$

For the same reason we have that $\tau'[\bar{\sigma}_2/\bar{\alpha}] = \tau = \tau'[\bar{\sigma}'_2/\bar{\alpha}]$. Since we know that $\bar{\alpha} \subseteq \mathit{ftv}(\tau)$, it must follow also that

$$\bar{\sigma}_2 = \bar{\sigma}'_2$$

As a consequence, we also have that

$$E_1 = E[|\bar{\sigma}_2|/\bar{\alpha}] = E[|\bar{\sigma}'_2|/\bar{\alpha}] = E_2$$

Finally, it also follows from rule (M-SIMP) that

$$\Sigma_1 = \varepsilon = \Sigma_2$$

and trivially

$$\vdash_{\text{unamb}} \varepsilon$$

(UA-IABS) $\quad \bar{\alpha} \vdash_{\text{unamb}} \rho_1 \Rightarrow \rho_2$

Then the second and third hypothesis of the lemma must have been formed by rule (M-IAPP). From their two hypotheses and from the hypothesis of the rule and the induction hypothesis, we obtain the desired results

$$\bar{\sigma}_1 = \bar{\sigma}_1' \quad \wedge \quad \bar{\sigma}_2 = \bar{\sigma}_2' \quad \wedge \quad E_1 = E_2 \quad \wedge \quad \Sigma_1, \rho_1[|\bar{\sigma}_2|/\bar{\alpha}] \rightsquigarrow x = \Sigma_2, \rho_1[|\bar{\sigma}_2'|/\bar{\alpha}] \rightsquigarrow x$$

We also derive from the induction hypothesis that $\vdash_{\text{unamb}} \Sigma_1$. Since $\bar{\alpha} \vdash_{\text{unamb}} \rho_1 \Rightarrow \rho_2$, we also have $\vdash_{\text{unamb}} \rho_1$. Hence we also conclude

$$\vdash_{\text{unamb}} \Sigma_1, \rho_1 \rightsquigarrow x$$

(UA-TABS) $\quad \bar{\alpha} \vdash_{\text{unamb}} \forall \alpha.\rho$

Then the second and third hypothesis of the lemma must have been formed by rule (M-TAPP), with $\bar{\sigma}_1 = \bar{\sigma}_{1,1}, \sigma_{1,2}$ and $\bar{\sigma}_1' = \bar{\sigma}_{1,1}', \sigma_{1,2}'$. From their two hypotheses and from the hypothesis of the rule and the induction hypothesis, we obtain

$$\bar{\sigma}_2, \sigma_{1,2} = \bar{\sigma}_2', \sigma_{1,2}' \quad \wedge \quad \bar{\sigma}_{1,1} = \bar{\sigma}_{1,1}' \quad \wedge \quad E_1 = E_2 \quad \wedge \quad \Sigma_1 = \Sigma_2 \quad \wedge \quad \vdash_{\text{unamb}} \Sigma_1$$

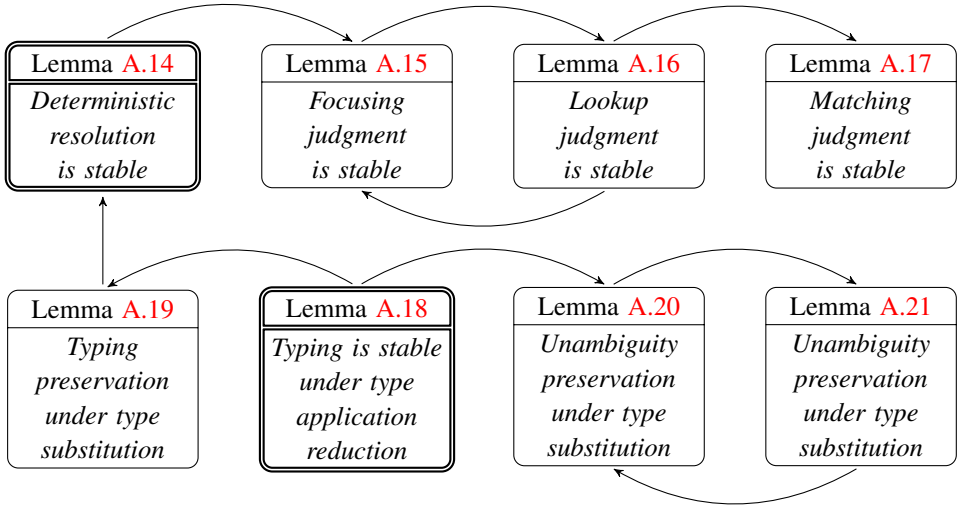
From this we conclude the desired result

$$\bar{\sigma}_1 = \bar{\sigma}_1' \quad \wedge \quad \bar{\sigma}_2 = \bar{\sigma}_2' \quad \wedge \quad E_1 = E_2 \quad \wedge \quad \Sigma_1 = \Sigma_2 \quad \wedge \quad \vdash_{\text{unamb}} \Sigma_1$$

□

A.5 Resolution and Typing Stability

The proofs of this section concern stability of the deterministic resolution judgment, as well as stability of typing. The first part, similarly to Section A.4, is split into subproofs for each auxiliary judgment (focusing, lookup and matching) of the deterministic resolution judgment and follows their dependencies. The second part uses the main result of the first part to establish stability of typing under type application reduction. In addition, it uses the preservation-under-type-substitution properties of typing and of the unambiguity judgment. The dependencies of the proofs of this section are depicted in the diagram below, where main results are indicated with double borders.



Deterministic resolution is stable under substitution.

Lemma A.14. *If*

$$\Gamma, \alpha, \Gamma' \vdash_{\mathrm{r}} \rho \rightsquigarrow E$$

then

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Proof. The proof proceeds by induction on the first derivation and uses Lemma A.15.

The hypothesis of rule (R-MAIN) then is

$$\mathit{ftv}(\Gamma), \alpha, \mathit{ftv}(\Gamma'); \Gamma, \alpha, \Gamma' \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$$

From Lemma A.15 it follows that

$$\mathit{ftv}(\Gamma), \mathit{ftv}(\Gamma'); \Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha]$$

As $\mathit{ftv}(\Gamma') = \mathit{ftv}(\Gamma'[\sigma/\alpha])$, the desired result follows from rule (R-MAIN)

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

□

The next two proofs make use of the auxiliary definition R, shown in Figure A 2. The relation $R(\Gamma; \alpha; \Gamma'; \Gamma''; \Gamma'''; \sigma)$ states that when performing a type substitution $\Gamma, \alpha, \Gamma' \vdash [\sigma/\alpha]$, then any $\Gamma'' \subseteq \Gamma, \alpha, \Gamma'$, results to Γ''' after the substitution has been performed. The two cases of the relation are necessary to separate between $\alpha \in \Gamma''$ and $\alpha \notin \Gamma''$.

Lemma A.15. *If*

$$\bar{\alpha}, \alpha, \bar{\alpha}; \Gamma, \alpha, \Gamma' \vdash_{\mathrm{r}} [\rho] \rightsquigarrow E$$

then

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha]$$

$$\boxed{R(\Gamma;\alpha;\Gamma';\Gamma'';\Gamma''';\sigma)}$$

(R-1) $R(\Gamma_1,\Gamma_2;\alpha;\Gamma';\Gamma_1;\Gamma_1;\sigma)$

(R-2) $R(\Gamma;\alpha;\Gamma'_1,\Gamma'_2;\Gamma,\alpha,\Gamma'_1;\Gamma,\Gamma'_1[\sigma/\alpha];\sigma)$

Fig. A 2. Auxiliary Definition R

Proof. The proof proceeds by induction on the derivation of the first hypothesis, mutually with Lemma A.16.

(R-IABS) $\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma' \vdash_{\mathrm{r}} [\rho_1 \Rightarrow \rho_2] \rightsquigarrow \lambda x : |\rho_1|.E$

From the rule's hypothesis and the induction hypothesis we have

$$\bar{\alpha},\bar{\alpha}';\Gamma,(\Gamma',?\rho_1 \rightsquigarrow x)[\sigma/\alpha] \vdash_{\mathrm{r}} [\rho_2[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha]$$

From the definition of substitution and rule (R-IABS) we then conclude

$$\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [(\rho_1 \Rightarrow \rho_2)[\sigma/\alpha]] \rightsquigarrow (\lambda x : |\rho_1|.E)[|\sigma|/\alpha]$$

(R-TABS) $\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma' \vdash_{\mathrm{r}} [\forall\beta.\rho] \rightsquigarrow \Lambda\beta.E$

From the rule's hypothesis and the induction hypothesis we have

$$\bar{\alpha},\bar{\alpha}';\Gamma,(\Gamma',\beta)[\sigma/\alpha] \vdash_{\mathrm{r}} [\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha]$$

From the definition of substitution and rule (R-TABS) we then conclude

$$\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [(\forall\beta.\rho)[\sigma/\alpha]] \rightsquigarrow (\Lambda\beta.E)[|\sigma|/\alpha]$$

(R-SIMP) $\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma' \vdash_{\mathrm{r}} [\tau] \rightsquigarrow E$

From the rule's hypothesis and Lemma A.16 we conclude

$$\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha];[\Gamma'''] \vdash_{\mathrm{r}} \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

and

$$R(\Gamma;\alpha;\Gamma';\Gamma,\alpha,\Gamma';\Gamma''';\sigma)$$

The latter could only have been obtained by rule (R-2). Hence, we know that $\Gamma''' = \Gamma,\Gamma'[\sigma/\alpha]$ and the former is equivalent to

$$\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha];[\Gamma,\Gamma'[\sigma/\alpha]] \vdash_{\mathrm{r}} \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

With this fact we can conclude by rule (R-SIMP)

$$\bar{\alpha},\bar{\alpha}';\Gamma,\Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [\tau[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha]$$

□

Lemma A.16. *If*

$$\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; [\Gamma''] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$$

and

$$\Gamma'' \subseteq \Gamma, \alpha, \Gamma'$$

then

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma'''] \vdash_{\mathrm{r}} \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

and

$$R(\Gamma; \alpha; \Gamma'; \Gamma''; \Gamma'''; \sigma)$$

Proof. The proof proceeds by induction on the derivation of the first hypothesis, mutually with Lemma A.15.

(L-MATCH) $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; [\Gamma'', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$

Then it follows from the first hypothesis of the rule and of Lemma A.17 that

$$\Gamma, \Gamma'[\sigma/\alpha]; [\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha] \vdash_{\mathrm{r}} \bar{\rho}[\sigma/\alpha] \rightsquigarrow \bar{x}; \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

Also it follows from the second hypothesis of the rule and of Lemma A.15 that

$$\alpha, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} [\bar{\rho}[\sigma/\alpha]] \rightsquigarrow \bar{E}[|\sigma|/\alpha]$$

By combining these two observations with rule (L-MATCH) we obtain the first desired result

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma''', ?\rho[\sigma/\alpha] \rightsquigarrow x] \vdash_{\mathrm{r}} \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

We obtain the second desired result by case analysis on $\Gamma'' \subseteq \Gamma, \alpha, \Gamma'$:

1. $\Gamma = \Gamma_1, ?\rho \rightsquigarrow x, \Gamma_2 \quad \wedge \quad \Gamma'' = \Gamma_1$:
   In this case we can use rule (R-1) to establish:
   $$R((\Gamma_1, ?\rho \rightsquigarrow x), \Gamma_2; \alpha; \Gamma'; \Gamma_1, ?\rho \rightsquigarrow x; \Gamma_1, ?\rho \rightsquigarrow x; \sigma)$$
   which is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', ?\rho \rightsquigarrow x; \Gamma'', ?\rho \rightsquigarrow x; \sigma)$$
2. $\Gamma' = \Gamma'_1, ?\rho \rightsquigarrow x, \Gamma'_2 \quad \wedge \quad \Gamma'' = \Gamma, \alpha, \Gamma'_1$:
   In this case we can use rule (R-2) to establish:
   $$R(\Gamma; \alpha; (\Gamma'_1, ?\rho \rightsquigarrow x), \Gamma'_2; \Gamma, \alpha, (\Gamma'_1, ?\rho \rightsquigarrow x); \Gamma, (\Gamma'_1, ?\rho \rightsquigarrow x)[\sigma/\alpha]; \sigma)$$
   which is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', ?\rho \rightsquigarrow x; \Gamma, (\Gamma'_1, ?\rho \rightsquigarrow x)[\sigma/\alpha]; \sigma)$$

(L-NOMATCH) $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; [\Gamma'', ?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'$

The rule's first hypothesis states that

$$\nexists \theta, E, \Sigma, dom(\theta) \subseteq (\bar{\alpha}, \alpha, \bar{\alpha}') : \theta(\Gamma, \alpha, \Gamma'); [\theta(\rho)] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma; \theta(\tau) \rightsquigarrow E$$

Hence, the above also holds when we restrict θ to be of the form $\theta' \cdot [\sigma/\alpha]$. In this case, the above simplifies to

$$\nexists \theta', E, \Sigma, dom(\theta) \subseteq (\bar{\alpha}, \bar{\alpha}') : \theta'(\Gamma, \Gamma'[\sigma/\alpha]); [\theta'(\rho[\sigma/\alpha])] \rightsquigarrow x \vdash_r \Sigma; \theta'(\tau[\sigma/\alpha]) \rightsquigarrow E$$

From the rule's second hypothesis and the induction hypothesis we have

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma'''] \vdash_r \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

With rule (L-NoMatch) we combine these two observations into the desired first result

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, (\Gamma', ?\rho \rightsquigarrow x)[\sigma/\alpha]; [\Gamma'''] \vdash_r \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

Similarly, following the rule's second hypothesis and the induction hypothesis we have:

$$R(\Gamma; \alpha; \Gamma'; \Gamma''; \Gamma'''; \sigma)$$

We do a case analysis on the derivation of this judgment.

1. (R-1):
   Then we have
   $$\Gamma = \Gamma_1, x : \rho, \Gamma_2 \quad \wedge \quad \Gamma'' = \Gamma_1 \quad \wedge \quad \Gamma''' = \Gamma_1$$
   By rule (R-2) we then have
   $$R((\Gamma_1, x : \rho), \Gamma_2; \alpha; \Gamma'; \Gamma_1, ?\rho \rightsquigarrow x; \Gamma_1, ?\rho \rightsquigarrow x; \sigma)$$
   which, given all the equations we have, is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', ?\rho \rightsquigarrow x; \Gamma'', ?\rho \rightsquigarrow x; \sigma)$$
2. (R-2):
   Then we have
   $$\Gamma' = \Gamma'_1, \Gamma'_2 \quad \wedge \quad \Gamma'' = \Gamma, \alpha, \Gamma'_1 \quad \wedge \quad \Gamma''' = \Gamma, \Gamma'_1[\sigma/\alpha]$$
   Since $\Gamma'', ?\rho \rightsquigarrow x \subseteq \Gamma, \alpha, \Gamma'$, it follows that $\Gamma'_2 = ?\rho \rightsquigarrow x, \Gamma'_{2,2}$. Hence, by rule (R-2) we can establish
   $$R(\Gamma; \alpha; (\Gamma'_1, ?\rho \rightsquigarrow x), \Gamma'_2; \Gamma, \alpha, (\Gamma'_1, ?\rho \rightsquigarrow x); \Gamma, (\Gamma'_1, ?\rho \rightsquigarrow x)[\sigma/\alpha]; \sigma)$$
   which, given all the equations we have, is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', ?\rho \rightsquigarrow x; \Gamma, (\Gamma'_1, ?\rho \rightsquigarrow x)[\sigma/\alpha]; \sigma)$$

(L-Var) $\quad \bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; [\Gamma'', x : \rho] \vdash_r \tau \rightsquigarrow E$

Then following the rule's hypothesis and the induction hypothesis we have:

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma'''] \vdash_r \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

By rule (L-Var) and the definition of substitution we then have

$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma''', x : \rho[\sigma/\alpha]] \vdash_r \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Similarly, following the rule's hypothesis and the induction hypothesis we have:

$$R(\Gamma; \alpha; \Gamma'; \Gamma''; \Gamma'''; \sigma)$$

We do a case analysis on the derivation of this judgment.

1. (R-1):
   Then we have
   $$\Gamma = \Gamma_1, x : \rho, \Gamma_2 \quad \wedge \quad \Gamma'' = \Gamma_1 \quad \wedge \quad \Gamma''' = \Gamma_1$$
   By rule (R-2) we then have
   $$R((\Gamma_1, x : \rho), \Gamma_2; \alpha; \Gamma'; \Gamma_1, x : \rho; \Gamma_1, x : \rho; \sigma)$$
   which, given all the equations we have, is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', x : \rho; \Gamma'', x : \rho; \sigma)$$
2. (R-2):
   Then we have
   $$\Gamma' = \Gamma'_1, \Gamma'_2 \quad \wedge \quad \Gamma'' = \Gamma, \alpha, \Gamma'_1 \quad \wedge \quad \Gamma''' = \Gamma, \Gamma'_1[\sigma/\alpha]$$
   Since $\Gamma'', x : \rho \subseteq \Gamma, \alpha, \Gamma'$, it follows that $\Gamma'_2 = x : \rho, \Gamma'_{2,2}$. Hence, by rule (R-2) we can establish
   $$R(\Gamma; \alpha; (\Gamma'_1, x : \rho), \Gamma'_2; \Gamma, \alpha, (\Gamma'_1, x : \rho); \Gamma, (\Gamma'_1, x : \rho)[\sigma/\alpha]; \sigma)$$
   which, given all the equations we have, is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', x : \rho; \Gamma, (\Gamma'_1, x : \rho)[\sigma/\alpha]; \sigma)$$

(L-TyVar) $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; [\Gamma'', \beta] \vdash_r \tau \rightsquigarrow E$

Then following the rule's hypothesis and the induction hypothesis we have:
$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma'''] \vdash_r \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$
By rule (L-TyVar) and the definition of substitution we then have
$$\bar{\alpha}, \bar{\alpha}'; \Gamma, \Gamma'[\sigma/\alpha]; [\Gamma''', \beta] \vdash_r \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$
Similarly, following the rule's hypothesis and the induction hypothesis we have:
$$R(\Gamma; \alpha; \Gamma'; \Gamma''; \Gamma'''; \sigma)$$
We do a case analysis on the derivation of this judgment.

1. (R-1):
   Then we have
   $$\Gamma = \Gamma_1, \Gamma_2 \quad \wedge \quad \Gamma'' = \Gamma_1 \quad \wedge \quad \Gamma''' = \Gamma_1$$
   We further distinguish between two mutually exclusive cases:
   (a) $\Gamma_2 = \varepsilon$
   It follows that $\alpha = \beta$ and we can establish by means of (R-2) that
   $$R(\Gamma_1, \Gamma_2; \alpha; \varepsilon, \Gamma'; \Gamma_1, \Gamma_2, \alpha; \Gamma_1, \Gamma_2, \varepsilon[\sigma/\alpha]; \sigma)$$
   which, given all the equations we have, is equivalent to
   $$R(\Gamma; \alpha; \Gamma'; \Gamma'', \beta; \Gamma; \sigma)$$

(b) $\Gamma_2 \neq \varepsilon$

Then it follows that $\Gamma_2 = \beta, \Gamma_{2,2}$ and by rule (R-2) we have

$$R((\Gamma_1, \beta), \Gamma_{2,2}; \alpha; \Gamma'; \Gamma_1, \beta; \Gamma_1, \beta; \sigma)$$

which, given all the equations we have, is equivalent to

$$R(\Gamma; \alpha; \Gamma'; \Gamma'', \beta; \Gamma_1, \beta; \sigma)$$

2. (R-2):

Then we have

$$\Gamma' = \Gamma'_1, \Gamma'_2 \quad \wedge \quad \Gamma'' = \Gamma, \alpha, \Gamma'_1 \quad \wedge \quad \Gamma''' = \Gamma, \Gamma'_1[\sigma/\alpha]$$

Since $\Gamma'', \beta \subseteq \Gamma, \alpha, \Gamma'$, it follows that $\Gamma'_2 = \beta, \Gamma'_{2,2}$. Hence, by rule (R-2) we can establish

$$R(\Gamma; \alpha; (\Gamma'_1, \beta), \Gamma'_2; \Gamma, \alpha, (\Gamma'_1, \beta); \Gamma, (\Gamma'_1, \beta)[\sigma/\alpha]; \sigma)$$

which, given all the equations we have, is equivalent to

$$R(\Gamma; \alpha; \Gamma'; \Gamma'', \beta; \Gamma, (\Gamma'_1, \beta)[\sigma/\alpha]; \sigma)$$

□

Lemma A.17. *If*

$$\Gamma, \alpha, \Gamma'; [\rho] \rightsquigarrow E \vdash_r \bar{\rho} \rightsquigarrow \bar{x}; \tau \rightsquigarrow E'$$

then

$$\Gamma, \Gamma'[\sigma/\alpha]; [\rho[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha] \vdash_r \bar{\rho}[\sigma/\alpha] \rightsquigarrow \bar{x}; \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

Proof. The proof proceeds by induction on the derivation of the hypothesis.

(M-SIMP) $\Gamma, \alpha, \Gamma'; [\tau] \rightsquigarrow E \vdash_r \varepsilon; \tau \rightsquigarrow E$

The desired conclusion follows directly from rule (M-SIMP)

$$\Gamma, \Gamma'[\sigma/\alpha]; [\tau[\sigma/\alpha]] \rightsquigarrow E[|\sigma/\alpha|] \vdash_r \varepsilon; \tau[\sigma/\alpha] \rightsquigarrow E[|\sigma/\alpha|]$$

(M-IAPP) $\Gamma, \alpha, \Gamma'; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E \vdash_r \Sigma, \rho_1 \rightsquigarrow x; \tau \rightsquigarrow E'$

From the rule's hypothesis and the induction hypothesis we have

$$\Gamma, (\Gamma', ?\rho_1 \rightsquigarrow x)[\sigma/\alpha]; [\rho_2[\sigma/\alpha]] \rightsquigarrow (E\ x)[|\sigma|/\alpha] \vdash_r (\rho_1 \rightsquigarrow x, \Sigma)[\sigma/\alpha]; \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

Then from the definition of substutition and rule (M-IAPP) we conclude

$$\Gamma, \Gamma'[\sigma/\alpha]; [(\rho_1 \Rightarrow \rho_2)[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha] \vdash_r \Sigma[\sigma/\alpha]; \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

(M-TAPP) $\Gamma, \alpha, \Gamma'; [\forall \beta.\rho] \rightsquigarrow E \vdash_r \Sigma; \tau \rightsquigarrow E'$

From the rule's hypothesis and the induction hypothesis we have

$$\Gamma, \Gamma'[\sigma/\alpha]; [\rho[\sigma'/\beta][\sigma/\alpha]] \rightsquigarrow (E|\sigma'|)[|\sigma|/\alpha] \vdash_r \Sigma[\sigma/\alpha]; \tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

We conclude by rule (M-TAPP), reasoning modulo the definition of substitution

$$\Gamma,\Gamma'[\sigma/\alpha];[(\forall\beta.\rho)[\sigma/\alpha]] \rightsquigarrow E[|\sigma|/\alpha] \vdash_r \Sigma[\sigma/\alpha];\tau[\sigma/\alpha] \rightsquigarrow E'[|\sigma|/\alpha]$$

□

Lemma A.18. *If*

$$\Gamma \vdash (\Lambda\alpha.e)\,\sigma : \rho[\sigma/\alpha] \rightsquigarrow (\Lambda\alpha.E)\,|\sigma|$$

then

$$\Gamma \vdash e[\sigma/\alpha] : \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Proof. By case analysis, the two last rules used in the derivation of the theorem's hypothesis must be instances of (TY-TABS) and (TY-TAPP), as shown below.

$$\text{TY-TAPP}\ \frac{\text{TY-TABS}\ \dfrac{\Gamma,\alpha \vdash e : \rho \rightsquigarrow E}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E}}{\Gamma \vdash (\Lambda\alpha.e)\,\sigma : \rho[\sigma/\alpha] \rightsquigarrow (\Lambda\alpha.E)\,|\sigma|}$$

Therefore, it suffices to show that

$$\begin{array}{rl} \text{If} & \Gamma,\alpha \vdash e : \rho \rightsquigarrow E \\ \text{then} & \Gamma \vdash e[\sigma/\alpha] : \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha] \end{array}$$

This can be proven as a special case ($\Gamma' = \varepsilon$) of the following theorem. □

Lemma A.19. *If*

$$\Gamma,\alpha,\Gamma' \vdash e : \rho \rightsquigarrow E$$

then

$$\Gamma,\Gamma'[\sigma/\alpha] \vdash e[\sigma/\alpha] : \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Proof. The proof proceeds by induction on the first hypothesis.

(TY-VAR) $\quad \Gamma,\alpha,\Gamma' \vdash x : \rho \rightsquigarrow x$

From the rule's premise, we know that $(x : \rho) \in \Gamma,\alpha,\Gamma'$. Hence, it also holds that $(x : \rho[\sigma/\alpha]) \in \Gamma,\Gamma'[\sigma/\alpha]$. Rule (TY-VAR) can then be instantiated with the latter, to conclude $\Gamma,\Gamma'[\sigma/\alpha] \vdash x : \rho[\sigma/\alpha] \rightsquigarrow x$.

(TY-ABS) $\quad \Gamma,\alpha,\Gamma' \vdash \lambda x : \rho_1.e : \rho_1 \to \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

From the rule's premise, we get $\Gamma,\alpha,\Gamma',x : \rho_1 \vdash e : \rho_2 \rightsquigarrow E$. By passing this to the induction hypothesis, we obtain

$$\Gamma,\Gamma'[\sigma/\alpha],x : \rho_1[\sigma/\alpha] \vdash e : \rho_2[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

From the definition of substitution and by supplying the latter to rule (TY-ABS), we reach the goal.

(TY-APP) $\Gamma, \alpha, \Gamma' \vdash e_1\, e_2 : \rho_2 \rightsquigarrow E_1\, E_2$

From the assumptions of rule (TY-APP), the induction hypothesis and the definition of substitution, we obtain

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash e_2 : \rho_1[\sigma/\alpha] \to \rho_2[\sigma/\alpha] \rightsquigarrow E_1[|\sigma|/\alpha]$$

and

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash e_1 : \rho_1[\sigma/\alpha] \rightsquigarrow E_2[|\sigma|/\alpha]$$

By instantiating rule (TY-APP) with the two judgments above, we obtain our goal.

(TY-TABS) $\Gamma, \alpha, \Gamma' \vdash \Lambda\alpha'.e : \forall\alpha'.\rho \rightsquigarrow \Lambda\alpha'.E$

From the assumptions of rule (TY-TABS) and the induction hypothesis, we obtain

$$\Gamma, (\Gamma', \alpha')[\sigma/\alpha] \vdash e[\sigma/\alpha] : \rho \rightsquigarrow E|\sigma|/\alpha]$$

where, from the definition of substitution, we have $(\Gamma', \alpha')[\sigma/\alpha] = \Gamma'[\sigma/\alpha], \alpha'$. Using this equation in the judgment above, we reach the goal by applying rule (TY-TABS) on it.

(TY-TAPP) $\Gamma, \alpha, \Gamma' \vdash e\,\rho_1 : \rho_2[\rho_1/\alpha'] \rightsquigarrow E\,|\rho_1|$

The first assumption of rule (TY-TAPP) is

$$\Gamma, \alpha, \Gamma' \vdash e : \forall\alpha'.\rho_2 \rightsquigarrow E$$

Then, from the induction hypothesis and the definition of substitution, we get

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash e[\sigma/\alpha] : \forall\alpha'.(\rho_2[\sigma/\alpha]) \rightsquigarrow E[|\sigma|/\alpha]$$

Using this in rule (TY-TAPP), we get

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash (e[\sigma/\alpha])\,(\rho_1[\sigma/\alpha]) : (\rho_2[\sigma/\alpha])[\rho_1[\sigma/\alpha]/\alpha'] \rightsquigarrow E[|\sigma|/\alpha]\,|\rho_1[\sigma/\alpha]|$$

which is syntactically equal to our goal, since by the definition of substitution and type translation and from the commutativity-like property of substitution composition, we get the following equations.

$$(e[\sigma/\alpha])\,(\rho_1[\sigma/\alpha]) = (e\,\rho_1)[\sigma/\alpha]$$

$$(\rho_2[\sigma/\alpha])[\rho_1[\sigma/\alpha]/\alpha'] = \rho_2[\rho_1/\alpha'][\sigma/\alpha]$$

$$E[|\sigma|/\alpha]\,|\rho_1[\sigma/\alpha]| = E[|\sigma|/\alpha]\,|\rho_1|[|\sigma|/\alpha] = (E\,\rho_1)[|\sigma|/\alpha]$$

(TY-IABS) $\Gamma, \alpha, \Gamma' \vdash \lambda_?\rho_1.e : \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

From the first assumption of rule (TY-IABS) and the induction hypothesis, we get

$$\Gamma, \Gamma'[\sigma/\alpha], ?\rho_1[\sigma/\alpha] \rightsquigarrow x \vdash e[\sigma/\alpha] : \rho_2[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Passing the second assumption of rule (TY-IABS) to Lemma A.20, we get

$$\vdash_{\text{unamb}} \rho_1[\sigma/\alpha]$$

By using the two obtained results and the freshness assumption of rule (TY-IABS) to instantiate a new (TY-IABS) rule, we reach the goal.

(TY-IAPP) $\Gamma, \alpha, \Gamma' \vdash e_1$ **with** $e_2 : \rho_1 \rightsquigarrow E_1\, E_2$

From the two assumptions of rule (TY-IAPP), the induction hypothesis and the definition of substitution, we obtain

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash e_1[\sigma/\alpha] : \rho_2[\sigma/\alpha] \Rightarrow \rho_1[\sigma/\alpha] \rightsquigarrow E_1[|\sigma|/\alpha]$$
$$\text{and} \quad \Gamma, \Gamma'[\sigma/\alpha] \vdash e_2[\sigma/\alpha] : \rho_2[\sigma/\alpha] \rightsquigarrow E_2[|\sigma|/\alpha]$$

By passing the two judgments above to rule (TY-IABS), we obtain the desired result.

(TY-QUERY) $\Gamma, \alpha, \Gamma' \vdash ?\rho : \rho \rightsquigarrow E$

From the first assumption of rule (TY-QUERY), where we consider deterministic resolution, and Lemma A.14 we get

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash_{\mathrm{r}} \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Passing the second assumption of rule (TY-QUERY) to Lemma A.20 results in

$$\vdash_{\mathrm{unamb}} \rho[\sigma/\alpha]$$

Then, we can instantiate rule (TY-QUERY) with the two obtained results above, to reach the goal of this case. □

Lemma A.20. *If*

$$\vdash_{\mathrm{unamb}} \rho$$

then

$$\vdash_{\mathrm{unamb}} \rho[\sigma/\alpha]$$

Proof. The proof proceeds by induction on the theorem's third hypothesis, mutually with the next proof. The only induction case of this proof, (UA-MAIN), is proved as a special case ($\bar{\alpha} = \varepsilon$) of Lemma A.21, since naturally $\alpha \notin \varepsilon$. □

Lemma A.21. *If*

$$\bar{\alpha} \vdash_{\mathrm{unamb}} \rho$$

and

$$\alpha \notin \bar{\alpha}$$

then

$$\bar{\alpha} \vdash_{\mathrm{unamb}} \rho[\sigma/\alpha]$$

Proof. The proof proceeds by induction on the theorem's first hypothesis, mutually with the previous proof.

(UA-SIMP) $\bar{\alpha} \vdash_{\text{unamb}} \tau$

We need to show that $\bar{\alpha} \vdash_{\text{unamb}} \tau[\sigma/\alpha]$, which can be satisfied only by (an appropriate instance of) rule (UA-SIMP). Therefore, it suffices to show that $\bar{\alpha} \subseteq \mathit{ftv}(\tau[\sigma/\alpha])$, where $\mathit{ftv}(\tau[\sigma/\alpha]) = (\mathit{ftv}(\tau) \setminus \{\alpha\}) \cup \mathit{ftv}(\sigma)$.

From the assumption of rule (UA-MAIN) and the second hypothesis of the theorem, we have that $\bar{\alpha} \subseteq \mathit{ftv}(\tau) \setminus \{\alpha\}$. Then trivially it also holds that $\bar{\alpha} \subseteq (\mathit{ftv}(\tau) \setminus \{\alpha\}) \cup \mathit{ftv}(\sigma)$.

(UA-TABS) $\bar{\alpha} \vdash_{\text{unamb}} \forall\beta.\rho$

From the second hypothesis of the theorem and the adopted Barendregt convention, it follows that $\alpha \notin \bar{\alpha}, \beta$.

From the assumption of the rule and the induction hypothesis, we get

$$\bar{\alpha}, \beta \vdash_{\text{unamb}} \rho[\sigma/\alpha]$$

By using this in rule (UA-TABS) and from the definition of substitution, we reach the goal:

$$\bar{\alpha} \vdash_{\text{unamb}} (\forall\beta.\rho)[\sigma/\alpha]$$

(UA-IABS) $\bar{\alpha}, \alpha, \bar{\alpha}' \vdash_{\text{unamb}} \rho_1 \Rightarrow \rho_2$

Passing the first assumption of rule (UA-IABS) to Lemma A.20 results in

$$\vdash_{\text{unamb}} \rho_1[\sigma/\alpha]$$

From the second assumption of rule (UA-IABS) and the induction hypothesis, we get

$$\bar{\alpha} \vdash_{\text{unamb}} \rho_2[\sigma/\alpha]$$

By the definition of substitution and the last two obtained results used with rule (UA-IABS), we reach the goal:

$$\bar{\alpha} \vdash_{\text{unamb}} (\rho_1 \Rightarrow \rho_2)[\sigma/\alpha]$$

□

A.6 Auxiliary Lemmas About the Unification Algorithm

This section concerns properties of the unification algorithm that are required by the proofs of the next sections. For this purpose, a notion of a partial order over substitutions is also used.

Definition A.1 (Partial Order over Substitutions)**.** *A substitution θ_1 is more general than a substitution θ_2, denoted with $\theta_2 \sqsubseteq \theta_1$, iff there is a θ' such that $\theta_2 = \theta' \cdot \theta_1$.*

Lemma A.22 states that the unification algorithm produces a substitution that indeed unifies its two input types.

Lemma A.22. *If*

$$\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$$

then

$$\theta(\tau) = \theta(\tau')$$

and

$$\bar{\alpha};\Gamma \vdash \theta$$

Proof. Straightforward induction on the derivation. □

We assume the following two properties of unification as given.

Assumption A.1 states that $\text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$ is the most general unifier for τ and τ'.

Assumption A.1. *If*

$$\theta(\tau) = \theta(\tau')$$

and

$$\bar{\alpha};\Gamma \vdash \theta$$

then

$$\theta \sqsubseteq \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$$

Assumption A.2. *If*

$$\theta(\tau) = \tau'$$

and

$$\bar{\alpha};\Gamma \vdash \theta$$

and

$$\bar{\alpha} \subseteq \mathit{ftv}(\tau)$$

and

$$\forall \alpha \in \bar{\alpha}, \forall \beta \in \mathit{ftv}(\tau'), \quad \beta >_\Gamma \alpha$$

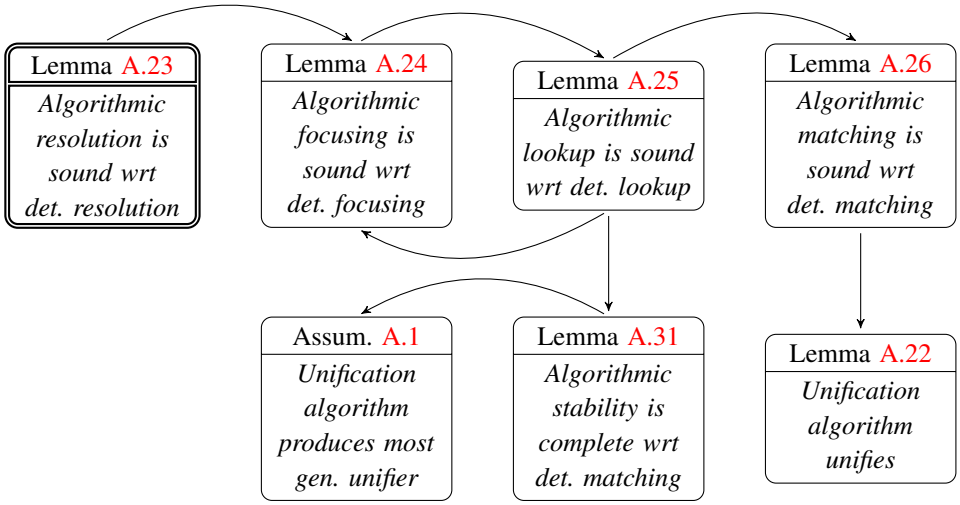
then

$$\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$$

A.7 Soundness of the Algorithmic wrt Deterministic Resolution

This section proves soundness of the resolution algorithm with respect to its declarative specification. As summarized in the diagram below, the main lemma depends on three auxiliary soundness lemmas, one for each of the algorithmic focusing, lookup and matching judgments. The dependencies of these proofs follow those of the definitions of the corresponding judgments.

Interestingly, the proof for algorithmic lookup soundness also uses completeness of the algorithmic stability judgment. The latter requires a correctness property of the unification algorithm. One more auxiliary correctness property is used by the algorithmic matching soundness proof.



Lemma A.23. *If*

$$\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$$

then

$$\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E$$

Proof. From the hypothesis it follows that

$$\mathsf{tyvars}(\Gamma); \Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$$

Hence, by Lemma A.24 and rule (R-MAIN) the desired conclusion follows. □

Lemma A.24. *If*

$$\bar{\alpha}; \Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$$

then

$$\bar{\alpha}; \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E$$

Proof. The proof proceeds by induction on the derivation of the hypothesis.

Cases (ALG-R-IABS) and (ALG-R-TABS) follow from the isomorphism between the rule sets of the two judgments. Case (ALG-R-SIMP) follows from Lemma A.25. □

Lemma A.25. *If*

$$\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{alg}} \rho \rightsquigarrow E$$

then

$$\bar{\alpha}; \Gamma; [\Gamma'] \vdash_{\text{r}} \rho \rightsquigarrow E$$

Proof. The proof proceeds by induction on the derivation of the hypothesis.

(ALG-L-MATCH) $\bar{\alpha}; \Gamma; [\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{alg}} \tau \rightsquigarrow E[\bar{E}/\bar{x}]$

From the rule's first hypothesis and Lemma A.26 we have

$$\Gamma;[\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \bar{\rho} \rightsquigarrow \bar{x};\tau \rightsquigarrow E'$$

where $\theta = \varepsilon$, as the only possibility such that $\varepsilon;\Gamma \vdash \theta$. Then, using Lemma A.24 and rule (L-MATCH) we conclude

$$\bar{\alpha};\Gamma;[\Gamma',?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E[\bar{E}/\bar{x}]$$

(ALG-L-NOMATCH) $\quad \bar{\alpha};\Gamma;[\Gamma',?\rho \rightsquigarrow x] \vdash_{\mathrm{alg}} \tau \rightsquigarrow E'$

From the rule's second premise and the induction hypothesis we have

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'$$

From the rule's first premise and the negation of Lemma A.31, there are no substitution, θ, expression, E, and set of goals, Σ, such that both judgments below hold.

$$\bar{\alpha};\Gamma \vdash \theta \quad \text{and} \quad \theta(\Gamma);[\theta(\rho)] \rightsquigarrow x \vdash_{\mathrm{r}} \Sigma;\theta(\tau) \rightsquigarrow E$$

Hence, with rule (L-NOMATCH) we conclude

$$\bar{\alpha};\Gamma;[\Gamma',?\rho \rightsquigarrow x] \vdash_{\mathrm{r}} \tau \rightsquigarrow E'$$

(ALG-L-VAR) $\quad \bar{\alpha};\Gamma;[\Gamma',x:\rho] \vdash_{\mathrm{alg}} \tau \rightsquigarrow E$

From the rule's hypothesis and the induction hypothesis we obtain

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$$

By rule (L-VAR) we conclude

$$\bar{\alpha};\Gamma;[\Gamma',x:\rho] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$$

(ALG-L-TYVAR) $\quad \bar{\alpha};\Gamma;[\Gamma',\alpha] \vdash_{\mathrm{alg}} \tau \rightsquigarrow E$

From the rule's hypothesis and the induction hypothesis we obtain

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$$

By rule (L-TYVAR) we conclude

$$\bar{\alpha};\Gamma;[\Gamma',\alpha] \vdash_{\mathrm{r}} \tau \rightsquigarrow E$$

□

Lemma A.26. *If*

$$\bar{\alpha};\Gamma;[\rho] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \Sigma';\tau \rightsquigarrow E'$$

then, there is a substitution θ *such that*

$$\bar{\alpha};\Gamma \vdash \theta$$

and

$$\theta(\Gamma);[\theta(\rho)] \rightsquigarrow |\theta|(E) \vdash_{\text{r}} \Sigma'';\theta(\tau) \rightsquigarrow E'$$

for Σ'' *such that*

$$\Sigma' = \theta(\Sigma),\Sigma''$$

Proof. The proof proceeds by induction on the derivation of the first hypothesis.

(ALG-M-SIMP) $\bar{\alpha};\Gamma;[\tau'] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \theta(\Sigma);\tau \rightsquigarrow |\theta|(E)$ with $\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau,\tau')$

From the hypothesis of the rule and Lemma A.22, it follows that $\bar{\alpha};\Gamma \vdash \theta$ and $\theta(\tau') = \theta(\tau)$. The latter allows us to use rule (M-SIMP), with which we obtain the goal judgment.

$$\theta(\Gamma);[\theta(\tau')] \rightsquigarrow |\theta|E \vdash_{\text{r}} \varepsilon;\theta(\tau') \rightsquigarrow |\theta|E$$

(ALG-M-IAPP) $\bar{\alpha};\Gamma;[\rho_1 \Rightarrow \rho_2] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \Sigma';\tau \rightsquigarrow E'$

From the premise of the rule, the induction hypothesis and the definition of substitution, there is a substitution θ such that

$$\bar{\alpha};\Gamma,?\rho \rightsquigarrow x \vdash \theta$$

and

$$\theta(\Gamma),?\theta(\rho_1) \rightsquigarrow x;[\theta(\rho_2)] \rightsquigarrow |\theta|(E\,x) \vdash_{\text{r}} \Sigma'';\theta(\tau) \rightsquigarrow E'$$

where $\Sigma' = \theta(\Sigma,\rho_1 \rightsquigarrow x),\Sigma''$. From the first result, because θ regards only type variable substitutions, it is easy to deduce

$$\bar{\alpha};\Gamma \vdash \theta$$

By rule (M-IAPP), applied on the second result of the induction hypothesis, we may conclude

$$\theta(\Gamma);[\theta(\rho_1 \Rightarrow \rho_2)] \rightsquigarrow |\theta|(E) \vdash_{\text{r}} \theta(\rho_1 \rightsquigarrow x),\Sigma'';\theta(\tau) \rightsquigarrow E'$$

Trivially, the last condition of the theorem holds, since $\Sigma' = \theta(\Sigma),((\theta(\rho_1 \rightsquigarrow x),\Sigma'')$.

(ALG-M-TAPP) $\bar{\alpha};\Gamma;[\forall\alpha.\rho] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \Sigma';\tau \rightsquigarrow E'$

From the premise of the rule and the induction hypothesis, there is a substitution, θ, such that

$$\bar{\alpha},\alpha;\Gamma,\alpha \vdash \theta$$

and

$$\theta(\Gamma,\alpha);[\theta(\rho)] \rightsquigarrow |\theta|(E\,\alpha) \vdash_{\mathrm{r}} \Sigma'';\theta(\tau) \rightsquigarrow E'$$

where $\Sigma' = \theta(\Sigma),\Sigma''$.

By case analysis on the first result, it follows that $\theta = [\sigma/\alpha]\cdot\theta'$, where $\Gamma \vdash \sigma$ and $\bar{\alpha};\Gamma \vdash \theta'$. Note that $\alpha \notin ftv(\tau)$. Therefore, the deterministic matching judgment above can be refined to

$$\theta'(\Gamma);[\theta'(\rho[[\sigma/\alpha]])] \rightsquigarrow |\theta|'(E\,|\sigma|) \vdash_{\mathrm{r}} \Sigma'';\theta'(\tau) \rightsquigarrow E'$$

Hence, it follows from rule (M-TAPP) and the type well-formedness judgment for σ that

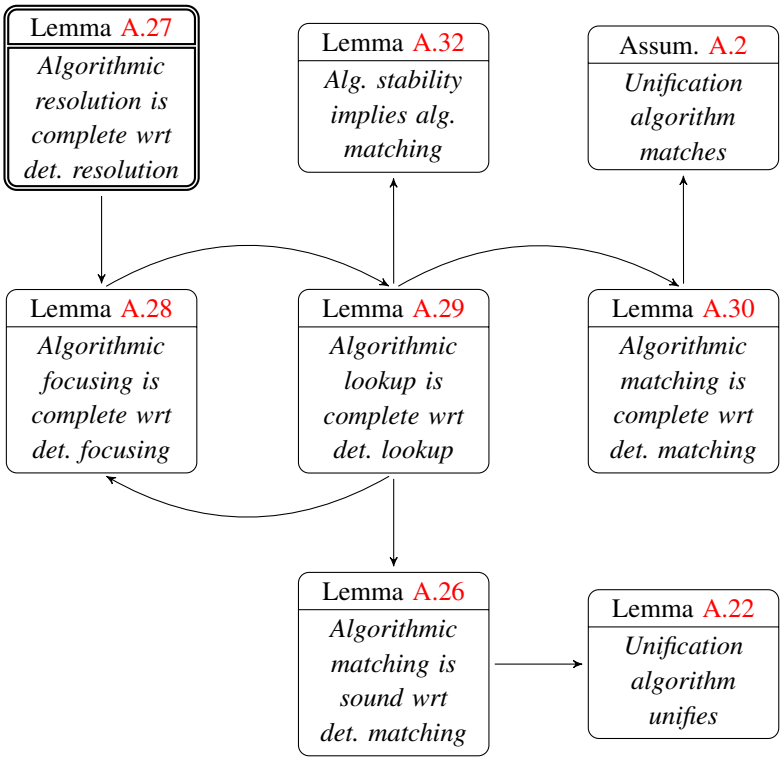
$$\theta'(\Gamma);[\theta'(\forall\alpha.\rho)] \rightsquigarrow |\theta|'(E) \vdash_{\mathrm{r}} \Sigma'';\theta'(\tau) \rightsquigarrow E'$$

□

A.8 Completeness of the Algorithm wrt Deterministic Resolution

This section proves completeness of the resolution algorithm with respect to its declarative counterpart. Like in Section A.7, the main lemma depends on three auxiliary completeness lemmas, one for each of the focusing, lookup and matching judgments. The dependencies of these proofs follow those of the definitions of the corresponding declarative judgments.

In symmetry with the proof of the auxiliary lookup lemma for algorithmic soundness, the proof for algorithmic lookup completeness also uses soundness of the algorithmic matching judgment, resulting in interdependent soundness and completeness proofs. Additionally, an auxiliary correctness property of the unification algorithm is necessary for proving completeness of algorithmic matching.



Lemma A.27. *If*

$$\vdash_{\text{unamb}} \Gamma$$

and

$$\vdash_{\text{unamb}} \rho$$

and

$$\Gamma \vdash_{\text{r}} \rho \rightsquigarrow E$$

then

$$\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$$

Proof. From the third hypothesis it follows that

$$\mathsf{tyvars}(\Gamma); \Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E$$

Hence, by Lemma A.28 and rule (ALG-R-MAIN) the desired conclusion follows

$$\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$$

□

Lemma A.28. *If*

$$\vdash_{\text{unamb}} \Gamma$$

and

$$\vdash_{\text{unamb}} \rho$$

and

$$\bar{\alpha};\Gamma \vdash_{\text{r}} [\rho] \rightsquigarrow E$$

then

$$\bar{\alpha};\Gamma \vdash_{\text{alg}} \rho \rightsquigarrow E$$

Proof. The proof proceeds by induction on the derivation of the third hypothesis.

Cases (R-IABS) and (R-TABS) follow from the isomorphism between the rule sets of the two judgments. Case (R-SIMP) follows from Lemma A.29. □

Lemma A.29. *If*

$$\vdash_{\text{unamb}} \Gamma'$$

and

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\text{r}} \rho \rightsquigarrow E$$

then

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\text{alg}} \rho \rightsquigarrow E$$

Proof. The proof proceeds by induction on the derivation of the second hypothesis.

(L-MATCH) $\bar{\alpha};\Gamma;[\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{r}} \tau \rightsquigarrow E[\bar{E}/\bar{x}]$

From the first hypothesis of the theorem, we derive that $\vdash_{\text{unamb}} \rho$. From the rule's second hypothesis and Lemma A.30, with $\theta = \varepsilon$ and $\alpha = \varepsilon$, we have

$$\varepsilon;\Gamma;[\rho] \rightsquigarrow x;\varepsilon \vdash_{\text{alg}} \bar{\rho} \rightsquigarrow \bar{x};\tau \rightsquigarrow E'$$

Then, using Lemma A.28 and rule (ALG-L-MATCH) we conclude

$$\bar{\alpha};\Gamma;[\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{alg}} \tau \rightsquigarrow E[\bar{E}/\bar{x}]$$

(L-NOMATCH) $\bar{\alpha};\Gamma;[\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{r}} \tau \rightsquigarrow E'$

From the rule's second hypothesis and the induction hypothesis we have

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E'$$

From the rule's first hypothesis, there are no θ, Σ_1 and E_1 such that $\bar{\alpha};\Gamma \vdash \theta$ and

$$\theta(\Gamma);[\theta(\rho)] \rightsquigarrow x \vdash_{\text{r}} E_1;\Sigma_1 \rightsquigarrow \tau$$

From the negation of Lemma A.26 and then the negation of Lemma A.32, we have:

$$\bar{\alpha};\Gamma;\rho \not\vdash_{\text{sta}} \tau$$

Hence with rule (ALG-L-NOMATCH) we conclude

$$\bar{\alpha};\Gamma;[\Gamma', ?\rho \rightsquigarrow x] \vdash_{\text{alg}} \tau \rightsquigarrow E'$$

(L-VAR) $\bar{\alpha};\Gamma;[\Gamma', x:\rho] \vdash_{\text{r}} \tau \rightsquigarrow E$

From the rule's hypothesis and the induction hypothesis we obtain

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E$$

By rule (ALG-L-VAR) we conclude

$$\bar{\alpha};\Gamma;[\Gamma', x:\rho] \vdash_{\text{alg}} \tau \rightsquigarrow E$$

(L-TYVAR) $\bar{\alpha};\Gamma;[\Gamma', \alpha] \vdash_{\text{r}} \tau \rightsquigarrow E$

From the rule's hypothesis and the induction hypothesis we obtain

$$\bar{\alpha};\Gamma;[\Gamma'] \vdash_{\text{alg}} \tau \rightsquigarrow E$$

By rule (ALG-L-TYVAR) we conclude

$$\bar{\alpha};\Gamma;[\Gamma', \alpha] \vdash_{\text{alg}} \tau \rightsquigarrow E$$

□

Lemma A.30. *If*

$$\theta(\Gamma);[\theta(\rho)] \rightsquigarrow |\theta|(E) \vdash_{\text{r}} \Sigma';\tau \rightsquigarrow E'$$

and

$$\bar{\alpha} \vdash_{\text{unamb}} \rho$$

and

$$dom(\theta) \subseteq \bar{\alpha}$$

and

$$\forall \alpha \in \bar{\alpha}, \forall \beta \in ftv(\tau), \quad \beta >_{\Gamma} \alpha$$

then, for all Σ*,*

$$\bar{\alpha};\Gamma;[\rho] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \theta(\Sigma), \Sigma';\tau \rightsquigarrow E'$$

Proof. The proof proceeds by induction on the derivation of the first hypothesis.

(M-SIMP) $\theta(\Gamma);[\theta(\tau')] \rightsquigarrow |\theta|(E) \vdash_{\text{r}} \varepsilon;\tau \rightsquigarrow E'$ with $\tau = \theta(\tau')$ and $E' = |\theta|(E)$

From the second hypothesis of the theorem, we have that $\bar{\alpha} \subseteq ftv(\tau')$. By using this, together with the third and fourth hypotheses of the theorem and the side condition that $\tau = \theta(\tau')$, in Assumption A.2, we get

$$\theta = \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$$

Trivially, from rule (ALG-M-SIMP) we have

$$\bar{\alpha};\Gamma;[\tau'] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \theta(\Sigma);\tau \rightsquigarrow |\theta|(E)$$

(M-IAPP) $\theta(\Gamma);[\theta(\rho_1 \Rightarrow \rho_2)] \rightsquigarrow |\theta|(E) \vdash_r \theta(\rho_1) \rightsquigarrow x, \Sigma'; \tau \rightsquigarrow E'$

From the hypothesis of the rule and the induction hypothesis, we have that

$$\forall \Sigma, \quad \bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\,x; \Sigma \vdash_{alg} \theta(\Sigma), \Sigma'; \tau \rightsquigarrow E'$$

Then, by choosing $\Sigma = \Sigma'', \rho_1 \rightsquigarrow x$ with any Σ'', the above can be refined to

$$\forall \Sigma'', \quad \bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x; [\rho_2] \rightsquigarrow E\,x; \Sigma'', \rho_1 \rightsquigarrow x \vdash_{alg} \theta(\Sigma'', \rho_1 \rightsquigarrow x), \Sigma'; \tau \rightsquigarrow E'$$

By rule (ALG-M-IAPP) we may then conclude

$$\forall \Sigma'', \quad \bar{\alpha}; \Gamma; [\rho_1 \Rightarrow \rho_2] \rightsquigarrow E; \Sigma'' \vdash_{alg} \theta(\Sigma''), \theta(\rho_1) \rightsquigarrow x, \Sigma'; \tau \rightsquigarrow E'$$

(M-TAPP) $\theta(\Gamma);[\theta(\forall \alpha.\rho)] \rightsquigarrow |\theta|(E) \vdash_r \Sigma'; \tau \rightsquigarrow E'$

From the definition of substitution and the premise of the rule, we have

$$\theta(\Gamma);[\theta(\rho)[\sigma/\alpha]] \rightsquigarrow |\theta|(E\,|\sigma|) \vdash_r \Sigma'; \tau \rightsquigarrow E'$$

The Barendregt convention allows us to asses that $\alpha \notin \theta(\Gamma)$. Thus, we can consider $\theta' = [\sigma/\alpha] \cdot \theta$, for which it follows that $dom(\theta') \subseteq \bar{\alpha}, \alpha$. From the definition of substitution,

$$\begin{aligned} \theta'(\rho) &= \theta(\rho)[\sigma/\alpha] \\ \theta'(\Gamma, \alpha) &= \theta(\Gamma) \\ |\theta'|(E\,\alpha) &= |\theta|(E\,|\sigma|) \end{aligned}$$

Also, since α does not appear free in $|\theta|(E)$, the substitution $[\sigma/\alpha]$ does not affect it. The premise of the rule can, then, be rewritten as

$$\theta'(\Gamma, \alpha);[\theta'(\rho)] \rightsquigarrow |\theta'|(E\,\alpha) \vdash_r \Sigma'; \tau \rightsquigarrow E'$$

From the second hypothesis of the theorem, we have that

$$\bar{\alpha}, \alpha \vdash_{unamb} \rho$$

and since α is fresh, it holds that for any $\beta \in ftv(\tau)$, $\beta >_\Gamma \alpha$. We can now use the induction hypothesis, to obtain

$$\bar{\alpha}, \alpha; \Gamma; [\rho] \rightsquigarrow E\,\alpha; \Sigma \vdash_{alg} \theta'(\Sigma), \Sigma'; \tau \rightsquigarrow E',$$

Hence, it follows from rule (ALG-M-TAPP) that

$$\bar{\alpha}; \Gamma; [\forall \alpha.\rho] \rightsquigarrow E; \Sigma \vdash_{alg} \theta'(\Sigma), \Sigma'; \tau \rightsquigarrow E'$$

□

Lemma A.31. *If*

$$\theta_1(\Gamma);[\theta_1(\rho)] \rightsquigarrow E \vdash_r \Sigma'; \theta_1(\tau) \rightsquigarrow E'$$

and

$$\bar{\alpha}; \Gamma \vdash \theta_1$$

then

$$\bar{\alpha}; \Gamma; \rho \vdash_{sta} \tau$$

Proof. The proof proceeds by induction on the derivation of the first hypothesis.

(M-SIMP) $\theta_1(\Gamma); [\theta_1(\tau')] \rightsquigarrow E \vdash_r \varepsilon; \theta_1(\tau) \rightsquigarrow E$ where $\theta_1(\tau) = \theta_1(\tau')$

By Assumption A.1 applied on the second hypothesis of the theorem and the side condition that $\theta_1(\tau) = \theta_1(\tau')$, we get

$$\theta_1 \sqsubseteq \text{unify}_{\Gamma;\bar{\alpha}}(\tau, \tau')$$

Trivially, from rule (STA-SIMP) we have

$$\bar{\alpha}; \Gamma; \tau' \vdash_{\text{sta}} \tau$$

(M-IAPP) $\theta_1(\Gamma); [\theta_1(\rho_1 \Rightarrow \rho_2)] \rightsquigarrow E \vdash_r \theta_1(\rho_1) \rightsquigarrow x, \Sigma'; \theta_1(\tau) \rightsquigarrow E'$

From the hypothesis of the rule and the induction hypothesis, we have that

$$\bar{\alpha}; \Gamma, ?\rho_1 \rightsquigarrow x; \rho_2 \vdash_{\text{sta}} \tau$$

We can strengthen this—which we use without proof—to obtain:

$$\bar{\alpha}; \Gamma; \rho_2 \vdash_{\text{sta}} \tau$$

Then, by rule (STA-IAPP) we conclude

$$\bar{\alpha}; \Gamma; \rho_1 \Rightarrow \rho_2 \vdash_{\text{sta}} \tau$$

(M-TAPP) $\theta_1(\Gamma); [\theta_1(\forall \alpha.\rho)] \rightsquigarrow E \vdash_r \Sigma'; \theta_1(\tau) \rightsquigarrow E'$

From the definition of substitution and the first premise of the rule, we have

$$\theta_1(\Gamma); [\theta_1(\rho)[\sigma/\alpha]] \rightsquigarrow E\,|\sigma| \vdash_r \Sigma'; \theta_1(\tau) \rightsquigarrow E'$$

The Barendregt convention allows us to assess that $\alpha \notin dom(\theta_1)$. Thus, we can consider $\theta' = [\sigma/\alpha] \cdot \theta_1$. By instantiating rule (S-CONS) with the theorem's second hypothesis and the second premise of rule (M-TAPP), we have

$$\bar{\alpha}, \alpha; \Gamma, \alpha \vdash \theta'$$

where, from the definition of substitution,

$$\theta'(\rho) = \theta_1(\rho)[\sigma/\alpha] \quad \text{and} \quad \theta'(\Gamma, \alpha) = \theta_1(\Gamma)$$

In addition, since α does not appear free in the type $\theta_1(\tau)$, it follows that the substitution $[\sigma/\alpha]$ does not affect it. The premise of the rule can, then, be rewritten as

$$\theta'(\Gamma, \alpha); [\theta'(\rho)] \rightsquigarrow E\,\sigma \vdash_r \Sigma'; \theta'(\tau) \rightsquigarrow E'$$

From the induction hypothesis, we have that

$$\bar{\alpha}, \alpha; \Gamma, \alpha; \rho \vdash_{\text{sta}} \tau,$$

Hence, it follows from rule (STA-TAPP) that

$$\bar{\alpha}; \Gamma; \forall \alpha.\rho \vdash_{\text{sta}} \tau$$

□

Lemma A.32. *If*

$$\bar{\alpha};\Gamma;\rho \vdash_{\text{sta}} \tau$$

then for all E,Σ *there exist* E',Σ' *such that*

$$\bar{\alpha};\Gamma;[\rho] \rightsquigarrow E;\Sigma \vdash_{\text{alg}} \Sigma';\tau \rightsquigarrow E'$$

Proof. The proof is straightforward induction on the derivation. The conclusion's judgement is an annotated version of the hypothesis' judgement. □