CoChis: Deterministic and Coherent Implicits

ANONYMOUS AUTHOR(S)

Implicit Progamming (IP) mechanisms infer values by a type-directed resolution process, 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 the desirability of coherence, ensuring each implicit has a unique resolution; 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 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 coherence and avoiding ambiguity. CoChis supports local scoping, overlapping instances, first-class instances, and higher-order rules, while remaining type safe and coherent.

CoChis has a compact formulation. 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 called focussing.

CCS Concepts: •**Theory of computation →Type structures;** •**Software and its engineering →Functional languages;**

ACM Reference format:
Anonymous Author(s). 2016. CoChis: Deterministic and Coherent Implicits. 1, 1, Article 1 (January 2016), 50 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

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 to avoid programming pitfalls. Very 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 and Blott 1989), Scala's implicits (Odersky 2010), JavaGI's generalized interfaces (Wehr et al. 2007), C++'s concepts (Gregor et al. 2006), Agda's *instance arguments* (Devriese and Piessens 2011), Coq's type classes (Sozeau and Oury 2008) and Rust's *traits* (Mozilla Research 2017). IP can also be viewed as a form of (type-directed) program synthesis (Manna and Waldinger 1980). The programming is said to be *implicit* because expressions (e.g., those for function parameters) 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 and 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 be *coherent* (Jones 1992; Reynolds 1991). Coherence means that any valid program must have exactly one meaning (that is, the semantics is not ambiguous/non-deterministic). 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 the usual coherence property and 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 coherence. Advanced features of type classes, such as overlapping instances,[1] pose severe problems for coherence. In purely functional programming, "*substituting equals by equals*" is expected to hold. That is, when given two equivalent expressions replacing one by the other in *any context* always leads to two programs that yield the same result. Special care (via restrictions) is needed to preserve coherence and the ability of substituting equals for equals in the presence of overlapping instances.

Various past work has pointed out limitations of type classes (Camarão and Figueiredo 1999; Dijkstra and Swierstra 2005; Dreyer et al. 2007; Garcia et al. 2007; Kahl and Scheffczyk 2001; Morris and Jones 2010; Oliveira et al. 2010, 2012). In particular type classes allow at most one instance per type (or severely restrict overlapping instances) to exist in a program. This means that 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 second class interfaces and that the type-directed rules cannot be higher-order (Oliveira et al. 2012).

An alternative school of thought in the design of IP mechanisms favours *flexibility*. For instance, Scala implicits and Agda's instance arguments do not impose all of the type class restrictions. For example, Scala supports local scoping of instances, which can be used to allow distinct "instances" to exists for the same type in different scopes in 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). 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 be 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, where the rule requirements can themselves be other rules. The implicit calculus has been shown to be type-safe. Unfortunately, both the implicit calculus and the various existing language mechanisms that embody flexibility do not preserve coherence and the ability to substitute equals for equals.

The design of IP mechanisms has led to heated debate (Hulley 2009; Kmett 2015; Zhang 2014) about the pros and cons of each school of thought: ease of reasoning versus flexibility. Proponents of the Haskell school of thought argue that having coherence is extremely desirable, and flexibility should not come at the cost of that property. Proponents of flexible IP mechanisms argue that flexibility is more important and, in practice, problems due to incoherence are rare. As far as we are aware only two designs preserve coherence, while allowing some extra flexibility for local scoping (Dreyer et al. 2007; Siek and Lumsdaine 2005). However neither of those designs supports overlapping instances and various other features, such as first-class and higher-order rules.

[1] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#overlapping-instances

This paper presents Cochis[2]: the Calculus Of CoHerent ImplicitS. Cochis is an improved variant of the implicit calculus that preserves *coherence*. Cochis supports local scoping, overlapping instances, first-class instances and higher-order rules. Yet, in contrast to most previous work that supports such features, the calculus is not only type-safe, but also 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, it is possible to model source languages on top of Cochis that support global scoping only. Global scoping can be viewed as a particular case of local scoping where a single, global, implicit environment is assumed, and no local scoping constructs are allowed.

Ensuring coherence in Cochis is challenging. The overlapping and higher-order nature of rules poses significant challenges for the coherence and determinism of Cochis's resolution. 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 *focussing* (Liang and Miller 2009; Miller et al. 1991; Pfenning 2010). However, unlike focused proof search, which is still essentially non-deterministic, Cochis's resolution employs additional techniques to be entirely deterministic and coherent. In particular, unlike focused proof search, our resolution uses a stack discipline to prioritize rules, and removes any recursive resolutions from matching decisions.

In summary, our contributions are as follows:

- We present Cochis, a *coherent* (and type-safe) minimal formal model for implicit programming that supports local scoping, overlapping rules, first-class instances 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 focussing in proof search.
- We provide a semantics in the form of a translation from Cochis to System F. We prove our translation to be type-safe, and coherent. The full proofs are available in the appendix of this paper.

Organization. Section 2 presents an informal overview of our calculus. Section 3 describes a polymorphic type system that statically excludes ill-behaved programs. Section 4 provides the elaboration semantics of our calculus into System F and correctness results. Section 5 discusses related work and Section 6 concludes.

2 OVERVIEW

This section summarizes the relevant background on type classes, IP and coherence, and introduces Cochis's key features for ensuring coherence. 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 coherence approach taken in Cochis.

2.1 Type Classes and Implicit Programming

Type classes enable the declaration of overloaded functions like comparison, pretty printing, or parsing.

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

[2]Cochise, 1804–1874, was chief of the Chokonen band of the Chiricahua Apache.

class *Read* α **where**
  read :: *String* $\rightarrow \alpha$

A type class declaration consists of: a class name, such as *Ord*, *Show* or *Read*; a type parameter, such as α; and a set of method declarations, such as those for ($\leqslant$), *show*, and *read*. 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**
  $x \leqslant y = primIntLe\ x\ y$
instance *Ord Char* **where**
  $x \leqslant y = primCharLe\ x\ y$
instance (*Ord* α, *Ord* β) $\Rightarrow$ *Ord* (α, β) **where**
  $(x, x') \leqslant (y, y') = x < y \vee (x \equiv y \wedge x' \leqslant y')$

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 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

sort :: *Ord* $\alpha \Rightarrow [\alpha] \rightarrow [\alpha]$

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. The body of the function may refer to $\leqslant$ on type α.

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 instance, 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:

Prelude > *sort* [(3, 'a'), (2, 'c'), (3, 'b')]
[(2, 'c'), (3, 'a'), (3, 'b')]

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

instance (*Ord* α, *Ord* β) $\Rightarrow$ *Ord* (α, β) **where**
  $(xa, xb) \leqslant (ya, yb) = xa \leqslant ya \wedge xb \leqslant yb$

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, the expression:

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

is rejected in Haskell due to *ambiguity* of *type class resolution* (Jones 1992). Functions *show* and *read* print and parse values of any type α that implements the classes *Show* and *Read*. The program is rejected because there is more that one possible choice for α, for example it could be *Int*, *Float*, or *Char*. Choosing $\alpha = Float$ leads to *False*, since showing the float 3 would result in `"3.0"`, while choosing $\alpha = Int$ leads to *True*.

Overlapping and Incoherent Instances. Advanced features of type classes, such as overlapping instances, require additional restrictions to ensure coherence. The following program illustrates 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.

The overlapping declarations are clearly incoherent, since it is unclear whether *trans* 3 should return 3 using the first instance, or 4 using the second instance. Because the second instance is more specific, one might expect that it supersedes the first one; and that is indeed how Haskell assigns a meaning to overlapping instances.

But now consider the following declaration.

$bad :: \alpha \to \alpha$
$bad\ x = trans\ x$ -- incoherent definition!

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. Now *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 typecheck. But the use of incoherent instances is greatly discouraged.

Global Uniqueness of Instances. A consequence of having both coherence and 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. 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 datastructure 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.

Although global uniqueness is, in principle, a property that should hold in Haskell programs, Haskell implementations actually violate this property in various circumstances [3]. In fact it is acknowledged that providing a global uniqueness check is quite challenging for Haskell implementations [4].

2.3 Scala Implicits and Incoherence

Scala implicits (Oliveira et al. 2010) are an interesting alternative IP design. Unlike type classes, implicits have locally scoped rules. Consequently Scala does not have the global uniqueness property, since different "instances" may exist for the same type in different scopes. Another interesting difference between implicits and type classes is that values of any type can be used as implicit parameters; there are no special constructs analogous to type class or instance declarations. Instead, implicits are modeled with ordinary types. They can be abstracted over and do not suffer from the second-class nature of type classes. Such features mean that Scala implicits have a wider range of applications than type classes. For example, they can be used to solve the problem of *implicit configurations* (Kiselyov and Shan 2004) naturally. The following example, adapted from Kiselyov and Shan, illustrates this:

def *add* $(\alpha : Int, \beta : Int)$ (**implicit** *modulus* : *Int*) $= (\alpha + \beta)$ % *modulus*
def *mul* $(\alpha : Int, \beta : Int)$ (**implicit** *modulus* : *Int*) $= (\alpha * \beta)$ % *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 2 + 3 = 1 (mod 4) because 2 + 3 and 1 differ by a multiple of 4. The code shows the definition of addition and multiplication in modular arithmetic. In Scala % is modulo division. Both *add*ition and *mul*tiplication include a third (implicit) parameter, which is the modulus of the division. Although the modulus could be passed explicitly this would be extremelly cumbersume. 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 synthesizing 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 under different scopes.

Incoherence in Scala. Although Scala allows *nested* local scoping and overlapping rules, *coherence* 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 $\alpha \Rightarrow \alpha$. 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 synthesized 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 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

[3] http://stackoverflow.com/questions/12735274/breaking-data-set-integrity-without-generalizednewtypederiving
[4] https://mail.haskell.org/pipermail/haskell-cafe/2012-October/103887.html

```
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) incoherent 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

the expression in line (6), which is accepted in Haskell, is actually rejected in Scala.[5] Here the Scala compiler does detect two possible instances for $Int \Rightarrow Int$, but does not select the most specific one. Rejecting line (6) has another unfortunate consequence: not only is the semantics not preserved under unfolding, but typing is not preserved either! Clearly preserving desirable properties such as coherence and type preservation is a subtle matter in the presence of implicits and deserves careful study.

2.4 An Overview of Cochis

Like Haskell Cochis requires coherence and like Scala it permits nested declarations, and does not guarantee global uniqueness. Cochis improves upon the implicit calculus (Oliveira et al. 2012), which is an incoherent calculus designed to model the essence of Scala implicits. Like the implicit calculus it combines standard scoping mechanisms (abstractions and applications) and types à la System F, with a logic-programming-style query language. We now present the key features of Cochis and how these features are used for IP.

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.

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 $Int \Rightarrow Int$. 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

[5] We have observed this behavior for Scala 2.11; for lack of a specifiation, it is not clear to us whether this behavior is intended.

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.

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 $(3,4)$:

implicit 3 **in implicit** ($\lambda_? Int.(?Int, ?Int + 1)$) **in** $?(Int \times Int)$

Note that higher-order rules are a feature introduced by the implicit calculus and are neither supported in Haskell nor Scala.

Recursive Resolution. Note that resolving the query $?(Int \times Int)$ above involves applying multiple rules. The current environment does not contain the required integer pair. It does however contain the integer 3 and a rule $\lambda_? Int \Rightarrow Int \times Int.(?Int, ?Int + 1)$ 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 Rules and Queries. Cochis allows polymorphic rules. 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 rule. The following expression returns $((3,3),(True, True))$:

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

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 $((3,3),(3,3))$:

implicit 3 **in implicit** ($\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))$) **in** $?((Int \times Int) \times (Int \times Int))$

Locally and Lexically Scoped Rules. Rules can be nested and resolution respects the lexical scope of rules. Consider the following program:

```
implicit 1 in
  implicit True in
    implicit (λ?Bool. if ?Bool then 2 else 0) in
      ?Int
```

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.

2.5 Overlapping Rules and Coherence in Cochis

As the previous example shows, the lexical scope imposes a natural precedence on rules. This precedence means that the lexically nearest rule is used to resolve a query, and not necessarily the most specific rule. For instance, the following Cochis variation on the running *trans* example from Section 2.2

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

yields the result 3 as the inner identity rule has precedence over the more specific incrementation rule in the outer scope. Yet, this lexical precedense alone is insufficient to guarantee coherence. Consider the program

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

While the query $?(\beta \to \beta)$ always matches $\forall \alpha.\alpha \to \alpha$, that is not always the lexically nearest match. Indeed, if β is instantiated to Int the rule $\mathit{Int} \to \mathit{Int}$ is a nearer match. However, if β is instantiated to any other type, $\mathit{Int} \to \mathit{Int}$ is not a valid match. In summary, we cannot always statically determine the lexically nearest match.

One might consider to resolve the incoherence by picking the lexically nearest rule 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 soundness, this form of incoherence is nevertheless undesirable for two reasons. Firstly, it makes the behavior of programs harder to predict, and, secondly, the behavior of programs is not stable under inlining. Indeed, if we inline the function definition of *bad* at the call site and substitute the arguments, we obtain the specialised program

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

This program yields the result 4 while the original incoherent version would yield 3. To avoid this unpredictable behavior, Cochis rejects incoherent programs.

3 THE COCHIS CALCULUS

This section formalizes the syntax and type system of Cochis, while Section 4 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:

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

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*.

Expressions e include three abstraction-eliminination pairs. Binder $\lambda(x : \rho).e$ abstracts expression e over values of type ρ, is eliminated by application $e_1\ e_2$, and refers to the bound value with variable x. Binder $\Lambda \alpha.e$

abstracts expression e over types, is eliminated by type application $e\ \rho$, and refers to the bound type with type variable α (but α itself is not a valid expression). Binder $\lambda_?\rho.e$ abstracts expression e over implicit values of type ρ, is eliminated by implicit application e_1 **with** e_2, and refers to the implicitly bound value with implicit query $?\rho$. For convenience we adopt the Barendregt convention (Barendregt 1981), that variables in binders are distinct, throughout this article.

Using rule abstractions and applications we can build the **implicit** sugar used in Section 2.

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

Here $\overline{\lambda_?\rho.}$ is a shortform for $\lambda_?\rho_1.\ \ldots\ \lambda_?\rho_n.$, and $\overline{\textbf{with}\ e}$ is a shortform for $\textbf{with}\ e_1 \ldots \textbf{with}\ e_n$.

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 System F, which is included in our system.

Well-Formed Types. As in 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 ρ.

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

Judgement $\Gamma \vdash \rho$ holds if type ρ is well-formed with respect to type environment Γ, that is, if all free type variables of ρ occur in Γ.

Well-Typed Expressions. 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 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 $\vdash_{unamb} \rho_1$ states that the type ρ_1 must be unambiguous; we explain this concept in Section 3.4. 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-condition states that ρ must be unambiguous. Resolution is defined in terms of the auxiliary judgement $\Gamma \vdash_r^a \rho$, which is explained next.

3.3 Resolution

Figure 3 provides a first (ambiguous) definition of the resolution judgement. Its underlying principle is resolution in logic. Intuitively, $\Gamma \vdash_r^a \rho$ holds if Γ entails ρ, where the types in Γ and ρ are read as propositions. 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 arrow, functions $\rho_1 \to \rho_2$ and rules $\rho_1 \Rightarrow \rho_2$, and the important twist is that we choose to treat only rules as implications, leaving functions as uninterpreted predicates.

Unfortunately, the definition in Figure 3 suffers from two problems. Firstly, the definition is *not syntax-directed*; several of the inference rules have overlapping conclusions. Hence, a deterministic resolution algorithm is non-obvious. Secondly and more importantly, the definition is *ambiguous*: a derivation can be shown by multiple different derivations. For instance, consider again the resolution in the last example of Section 2.4: in the environment

$$\Gamma_0 = (Int, Bool, (Bool \Rightarrow Int))$$

$\boxed{\Gamma \vdash \rho}$

$$\text{(WF-VarTy)}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \text{(WF-FunTy)}\ \frac{\Gamma \vdash \rho_1 \qquad \Gamma \vdash \rho_2}{\Gamma \vdash \rho_1 \to \rho_2}$$

$$\text{(WF-UnivTy)}\ \frac{\Gamma, \alpha \vdash \rho}{\Gamma \vdash \forall \alpha.\rho} \qquad \text{(WF-RulTy)}\ \frac{\Gamma \vdash \rho_1 \qquad \Gamma \vdash \rho_2}{\Gamma \vdash \rho_1 \Rightarrow \rho_2}$$

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

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

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

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

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

$$\text{(Ty-TApp)}\ \frac{\Gamma \vdash e : \forall \alpha.\rho_2 \leadsto E \qquad \Gamma \vdash \rho_1}{\Gamma \vdash e\, \rho_1 : \rho_2[\rho_1/\alpha] \leadsto E\, |\rho_1|}$$

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

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

$$\text{(Ty-Query)}\ \frac{\Gamma \vdash^a_r \rho \leadsto E \qquad \Gamma \vdash \rho \qquad \vdash_{unamb}\ \rho}{\Gamma \vdash ?\rho : \rho \leadsto E}$$

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

there are two different derivations for $\Gamma_0 \vdash^a_r Int$:

$$\text{(AR-IVar)}\ \frac{Int \in \Gamma_0}{\Gamma_0 \vdash^a_r Int} \qquad \text{and} \qquad \begin{array}{l}\text{(AR-IApp)}\ \dfrac{\text{(AR-IVar)}\ \dfrac{(Bool \Rightarrow Int) \in \Gamma_0}{\Gamma_0 \vdash^a_r (Bool \Rightarrow Int)} \qquad \text{(AR-IVar)}\ \dfrac{Bool \in \Gamma_0}{\Gamma_0 \vdash^a_r Bool}}{\Gamma_0 \vdash^a_r Int}\end{array}$$

$$\boxed{\Gamma \vdash_r^a \rho \rightsquigarrow E} \qquad \text{(AR-IVar)}\ \frac{\rho \rightsquigarrow x \in \Gamma}{\Gamma \vdash_r^a \rho \rightsquigarrow x}$$

$$\text{(AR-TAbs)}\ \frac{\Gamma, \alpha \vdash_r^a \rho \rightsquigarrow E}{\Gamma \vdash_r^a \forall \alpha.\rho \rightsquigarrow \Lambda\alpha.E} \qquad \text{(AR-TApp)}\ \frac{\Gamma \vdash_r^a \forall \alpha.\rho \rightsquigarrow E \qquad \Gamma \vdash \rho'}{\Gamma \vdash_r^a \rho[\rho'/\alpha] \rightsquigarrow E\ |\rho'|}$$

$$\text{(AR-IAbs)}\ \frac{\Gamma, \rho_1 \rightsquigarrow x \vdash_r^a \rho_2 \rightsquigarrow E \qquad x\ \mathit{fresh}}{\Gamma \vdash_r^a \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E} \qquad \text{(AR-IApp)}\ \frac{\Gamma \vdash_r^a \rho_1 \Rightarrow \rho_2 \rightsquigarrow E_2 \qquad \Gamma \vdash_r^a \rho_1 \rightsquigarrow E_1}{\Gamma \vdash_r^a \rho_2 \rightsquigarrow E_2\ E_1}$$

Fig. 3. Ambiguous Resolution

While this may seem harmless at the type-level, at the value-level each derivation corresponds to a (possibly) different value. Hence, ambiguous resolution renders the meaning of a program ambiguous.

3.4 Deterministic Resolution

Figure 4 defines judgement $\Gamma \vdash_r \rho$, which is a syntax-directed deterministic variant of $\Gamma \vdash_r^a \rho$. This determinstic variant is sound with respect to the ambiguous definition. In other words, $\Gamma \vdash_r^a \rho$ holds if $\Gamma \vdash_r \rho$ holds. Yet, the opposite is not true. The deterministic judgement sacrifices some expressive power in exchange for better behavedness.

Revised Syntax. To facilitate the definition of the deterministic resolution judgement we split the syntax of types into three different sorts: *context* types, *simple* types and *monotypes*.

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

Context types ρ correspond to the original types ρ. *Simple types* τ are a restricted form of context types without toplevel quantifiers and toplevel implicit arrows. Singling out this restricted form turns out to be convenient for the type-directed formulation of the judgement.

Monotypes σ are a further refinement of simple types without universal quantifiers and implicit arrows anywhere. They help us to address a form of ambiguity due to the *impredicativity* of Rule (AR-TApp). For instance, if we define $\Gamma_1 = \forall \alpha.\alpha \Rightarrow \alpha$, then there are two ways to resolve $\Gamma_1 \vdash \mathit{Int} \Rightarrow \mathit{Int}$:

$$\text{(AR-IVar)}\ \frac{(\forall \alpha.\alpha \Rightarrow \alpha) \in \Gamma_1}{\text{(AR-TApp)}\ \dfrac{\Gamma_1 \vdash_r^a \forall \alpha.\alpha \Rightarrow \alpha}{\Gamma \vdash_r^a \mathit{Int} \Rightarrow \mathit{Int}}} \qquad \dfrac{\text{(AR-IVar)}\ \dfrac{(\forall \alpha.\alpha \Rightarrow \alpha) \in \Gamma_1}{\Gamma_1 \vdash_r^a (\forall \alpha.\alpha \Rightarrow \alpha)}}{\text{(AR-TApp)}\ \dfrac{\text{(AR-IApp)}\ \dfrac{\Gamma_1 \vdash_r^a (\forall \beta.\beta \Rightarrow \beta) \Rightarrow (\forall \beta.\beta \Rightarrow \beta) \qquad \text{(AR-IVar)}\ \dfrac{(\forall \beta.\beta \Rightarrow \beta) \in \Gamma_1}{\Gamma_1 \vdash_r^a (\forall \beta.\beta \Rightarrow \beta)}}{\Gamma_1 \vdash_r^a \forall \beta.\beta \Rightarrow \beta}}{\Gamma \vdash_r^a \mathit{Int} \Rightarrow \mathit{Int}}}$$

The proof on the left only involves the predicative generalisation from *Int* to α. Yet, the second proof contains an impredicative generalisation from $\forall \beta.\beta \Rightarrow \beta$ to α. Impredicativity is a well-known source of such problems in other settings, such as in type inference for the polymorphic λ-calculus (Boehm 1985; Pfenning 1993). The

established solution also works here: restrict to predicativity. This is where the monotype sort σ comes in: we only allow generalisation over (or dually, instantiation with) monotypes σ.

Revised Resolution Rules. Figure 4 defines the main judgement $\Gamma \vdash_r \rho$ in terms of three interdependent auxiliary judgements. The first of these auxiliary judgements is $\bar{\alpha}; \Gamma \vdash_r \rho$, where the type variables $\bar{\alpha}$ are the free type variables in the original environment at the point of the query. Recall the *bad* example from Section 2.5 where there is only one such free type variable: β. Tracking these free variables plays a crucial role in guaranteeing coherence and ensuring that resolution is stable under all type substitutions that instantiate these variables, like $[\beta \mapsto Int]$; how we prevent these substitutions is explained below. The main judgement retains these free variables in rule (R-Main) with the function *tyvars*:

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

While the auxiliary judgement $\bar{\alpha}; \Gamma \vdash_r \rho$ extends the type environment Γ, it does not update the type variables $\bar{\alpha}$. This judgement is syntax-directed on the query type ρ. Its job is to strip ρ down to a simple type τ using literal copies of the ambiguous rules (AR-TAbs) and (AR-IAbs), and then to hand it off to the second auxiliary judgement in rule (R-Simp).

The second auxiliary judgement, $\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau$, is syntax-directed on Γ': it traverses Γ' from right to left until it finds a rule type ρ that matches the simple type τ. Rules (L-Var) and (L-TyVar) skip the irrelevant entries in the environment. Rule (L-RuleMatch) identifies a matching rule type ρ – where matching is determined by with the third auxiliary judgement – and takes care of recursively resolving its context types; details follow below. Finally, rule (L-RuleNoMatch) skips a rule type in the environment if it does not match. Its condition $stable(\bar{\alpha}, \Gamma, \rho, \tau)$ entails the opposite of rule (L-RuleMatch)'s first condition: $\nexists \Sigma : \ \Gamma; \rho \vdash_r \tau; \Sigma$. (We come back to the reason why the condition is stronger than this in Section 3.4.) As a consequence, rules (L-RuleMatch) and (L-RuleNoMatch) are mutually excluse and *the judgement effectively commits to the right-most matching rule in* Γ'. We maintain the invariant that Γ' is a prefix of Γ; rule (R-Simp) provides Γ as the initial value for Γ'. Hence, if a matching rule type ρ is found, we have that $\rho \in \Gamma$. Hence, the second auxiliary judgement plays much the same role as rule (AR-IVar) in Figure 3, which also selects a rule type $\rho \in \Gamma$. The difference is that rule (AR-IVar) makes a non-deterministic choice, while the second auxiliary judgement makes determinstic committed choice that prioritizes rule types more to the right in the environment. For instance, $Int, Int \vdash_r^a Int$ has two ways to resolve, while $Int, Int \vdash_r Int$ has only one because the second Int in the environment shadows the first.

Finally, the third auxiliary judgement, $\Gamma; \rho \vdash_r \tau; \Sigma$, determines whether the rule type ρ matches the simple type τ. The judgement is defined by structural induction on ρ, which is step by step instantiated to τ. Any recursive resolutions are deferred in this process – the postponed resolvents are captured in the Σ argument. This way they do not influence the matching decision and backtracking is avoided. Instead, the recursive resolutions are executed, as part of rule (L-RuleMatch), after the rule has been committed to. Rule (M-Simp) constitutes the base case where the rule type equals the target type. Rule (M-IApp) is the counterpart of the original rule (R-IApp) where the implication arrow $\rho_1 \Rightarrow \rho_2$ is instantiated to ρ_2; the resolution of ρ_1 is deferred. Lastly, rule (M-TApp) is the counterpart of the original rule (R-TApp). The main difference is that it only uses monotypes σ to substitute the type variable; this implements the predicativity restriction explained above.

The relation to the ambiguous definition of resolution can be summarized as follows: if $\Gamma; \rho \vdash_r \tau; \bar{\rho}$ with $\Gamma \vdash_r^a \rho$ and $\Gamma \vdash_r^a \bar{\rho}$, then $\Gamma \vdash_r^a \tau$.

Non-Ambiguity Constraints. The rule (M-TApp) does not explain how the substitution $[\sigma/\alpha]$ for the rule type $\forall \alpha.\rho$ should be obtained. At first sight it seems that the choice of σ is free and thus a source of non-determinism. 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

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

$$\boxed{\Gamma \vdash_r \rho \rightsquigarrow E} \qquad \text{(R-Main)} \quad \frac{tyvars(\Gamma); \Gamma \vdash_r \rho \rightsquigarrow E}{\Gamma \vdash_r \rho \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma \vdash_r \rho \rightsquigarrow E}$$

$$\text{(R-IAbs)} \quad \frac{\bar{\alpha}; \Gamma, \rho_1 \rightsquigarrow x \vdash_r \rho_2 \rightsquigarrow E \qquad x \text{ fresh}}{\bar{\alpha}; \Gamma \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E} \qquad \text{(R-TAbs)} \quad \frac{\bar{\alpha}; \Gamma, \alpha \vdash_r \rho \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_r \forall \alpha.\rho \rightsquigarrow \Lambda \alpha.E}$$

$$\text{(R-Simp)} \quad \frac{\bar{\alpha}; \Gamma; \Gamma \vdash_r \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_r \tau \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E}$$

$$\text{(L-RuleMatch)} \quad \frac{\Gamma; \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E; \overline{\rho \rightsquigarrow x} \qquad \bar{\alpha}; \Gamma \vdash_r \bar{\rho} \rightsquigarrow \bar{E}}{\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E[\bar{E}/\bar{x}]} \qquad \text{(L-RuleNoMatch)} \quad \frac{stable(\bar{\alpha}, \Gamma, \rho, \tau) \qquad \bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E'}$$

$$\text{(L-Var)} \quad \frac{\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; \Gamma', x : \rho \vdash_r \tau \rightsquigarrow E} \qquad \text{(L-TyVar)} \quad \frac{\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; \Gamma', \alpha \vdash_r \tau \rightsquigarrow E}$$

$$\boxed{\Gamma; \rho \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \Sigma}$$

$$\text{(M-Simp)} \quad \Gamma; \tau \rightsquigarrow E \vdash_r \tau \rightsquigarrow E; \epsilon$$

$$\text{(M-IApp)} \quad \frac{\Gamma, \rho_1 \rightsquigarrow x; \rho_2 \rightsquigarrow E\, x \vdash_r \tau \rightsquigarrow E'; \Sigma \qquad x \text{ fresh}}{\Gamma; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \Sigma, \rho_1 \rightsquigarrow x}$$

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

$$\boxed{stable(\bar{\alpha}, \Gamma, \rho, \tau)} \qquad \text{(Stable)} \quad \frac{\nexists \theta, E, \Sigma, dom(\theta) \subseteq \bar{\alpha} : \theta(\Gamma); \theta(\rho) \rightsquigarrow x \vdash_r \theta(\tau) \rightsquigarrow E; \Sigma}{stable(\bar{\alpha}, \Gamma, \rho, \tau)}$$

Fig. 4. Deterministic Resolution and Translation to System F

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

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

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

Fig. 5. Unambiguous context types

$\forall\alpha.(\alpha \to String) \Rightarrow (String \to \alpha) \Rightarrow (String \to String)$. This type encodes the well-known ambiguous Haskell type $\forall\alpha.(Show\ \alpha, Read\ \alpha) \Rightarrow String \to String$ of the expression $read \circ show$. 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 the types $(String \to \alpha)[\sigma/\alpha]$ 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 $\vdash_{unamb} \rho_1$ side-condition in rule (Ty-IAbs) of Figure 2.[6] This judgement is defined in Figure 5 in terms of the auxiliary judgement $\bar{\alpha} \vdash_{unamb} \rho$ which takes an additional sequence of type variables α that is initially empty.

The auxililiary judgement 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 also fixes the type variables $\bar{\alpha}$. 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.

Finally, the unambiguity condition is also imposed on the queried type ρ in rule (Ty-Query) because this type too may extend the implicit environment in rule (R-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 ambiguity is harmless, because it does not affect the semantics. Yet, for the sake of simplicity, we have opted to not differentiate between harmless and harmful ambiguity.

Coherence Enforcement. In order to enforce coherence, rule (L-RuleNoMatch) makes sure that the decision to not select a context type is stable under all possible substitutions θ. Consider for instance the *bad* example from Section 2.5: when looking up $\beta \to \beta$, the rule $Int \to Int$ does not match and is otherwise skipped. Yet, under the substitution $\theta = [\beta \mapsto Int]$ the rule would match after all. In order to avoid this unstable situation, rule (L-RuleNoMatch) only skips a context type in the implicit environment, if there is no substitution θ for which the type would match the context type.

This approach is similar to the treatment of overlapping type class instances or overlapping type family instances in Haskell. However, there is one important complicating factor here: the query type may contain universal quantifiers. Consider a query for $\forall\alpha.\alpha \to \alpha$. In this case we wish to rule out entirely the context type

[6] An alternative design to avoid such ambiguity would instantiate unused type variables to a dummy type, like GHC's `GHC.Prim.Any`, which is only used for this purpose.

Int → *Int* as a potential match. Even though it matches under the substitution $\theta = [\alpha \mapsto Int]$, that covers only one instantiation while the query clearly requires a resolvent that covers all possible instantiations.

We clearly identify which type variables $\bar{\alpha}$ are to be considered for substitution by rule (L-RuleNoMatch) by parametrising the judgements by this set. These are the type variables that occur in the environment Γ at the point of the query. The main resolution judgement $\vdash_r \rho$ grabs them and passes them on to all uses of rule (L-RuleNoMatch).

3.5 Algorithm

Figure 6 contains an algorithm that implements the non-algorithmic deterministic resolution rules of Figure 4. It differs from the latter in two important ways: firstly, it replaces explicit quantification over all substitutions θ in rule (L-RuleNoMatch) with a tractable approach to coherence checking; and, secondly, it computes rather than guesses type substitutions in rule (M-TApp).

The definition of the algorithm is structured in much the same way as the declarative specification: with one main judgement and three auxiliary ones that have similar roles. In fact, since the differences are not situated in the main and first auxiliary judgement, these are actually identical.

Algorithmic No-Match Check. The first difference is situated in rule (Alg-L-RuleNoMatch) of the second judgement. Instead of an explicit quantification over all possible substitutions, this rule uses the more algorithmic judgement $\bar{\alpha}; \Gamma; \rho \vdash_{coh} \tau$. This auxiliary judgement checks algorithmically whether there context type ρ matches τ under any possible instantiation of the type variables $\bar{\alpha}$.

The definition of $\bar{\alpha}; \Gamma; \rho \vdash_{coh} \tau$ is a variation on that of the declarative judgement $\Gamma; \rho \vdash_r \tau; \Sigma$. There are three differences. Firstly, since the judgement is only concerned with matchability, no recursive resolvents Σ are collected. Secondly, instead of guessing the type instantiation ahead of time in rule (M-TApp), rule (Coh-TApp) defers the instantiation to the base case, rule (Coh-Simp). This last rule performs the deferred instantiation of type variables $\bar{\alpha}$ by computing the *most general domain-restricted unifier* $\theta = mgu_{\Gamma;\bar{\alpha}}(\tau', \tau)$. A substitution θ is a unifier of two types ρ_1 and ρ_2 iff $\theta(\rho_1) = \theta(\rho_2)$. A unifier θ is restricted to domain $\bar{\alpha}$ if $dom(\theta) \subseteq \bar{\alpha}$. A most general domain-restricted unifier θ subsumes any other unifier restricted to the same domain $\bar{\alpha}$:

$$\forall \eta : \quad dom(\eta) \subseteq \bar{\alpha} \wedge \eta(\rho_1) = \eta(\rho_2) \quad \Rightarrow \quad \exists \iota : dom(\iota) \subseteq \bar{\alpha} \wedge \iota(\theta(\rho_1)) = \iota(\theta(\rho_2))$$

If this most-general unifier exists, a match has been established. If no unifier exists, then rule COH-Simp does not apply. Thirdly, since the coherence check considers the substitution of the type variables $\bar{\alpha}$ that occur in the environment at the point of the query, rule (Alg-L-RuleNoMatch) pre-populates the substitutable variables of the $\vdash_{coh}$ judgement with them.

Deferred Variable Instantiation. The second main difference is situated in the third auxiliary judgement $\bar{\alpha}; \Gamma; \rho; \Sigma \vdash_{alg} \tau; \Sigma'$. This judgement is in fact an extended version of $\bar{\alpha}; \Gamma; \rho \vdash_{coh} \tau$ that does collect the recursive resolution obligations in Σ' just like the corresponding judgement in the declarative specification. The main difference with the latter is that it uses the deferred approach to instantiating type variables. In order to subject the resolution obligations to this substitution, which is computed in rule (Alg-M-Simp), the judgement makes use of an accumulating parameter Σ. This accumulator Σ represents all the obligations collected so far in which type variables have not been substituted yet. In contrast, Σ' denotes all obligations with type variables already substituted. Finally, note that rule (Alg-L-RuleMatch) does not pre-populate the type variables with those of the environment: we only want to instantiate the type variables that appear in the context type ρ itself for an actual match.

Domain-Restricted Unification. The algorithm for computing the most general domain-restricted unifier $\theta = mgu_{\Gamma;\bar{\alpha}}(\rho_1, \rho_2)$ is a key component of the two algorithmic changes explained above. Figure 7 provides its definition, which is an extension of standard first-order unification (Martelli and Montanari 1982). 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 differences with standard first-order unification arise because the algorithm has to account for type variable binders and the scope of type variables. For instance, using standard first-order unification for $mgu_{\Gamma;\beta}(\forall\alpha.\alpha \to \beta, \forall\alpha.\alpha \to \alpha)$ would yield the substitution $[\beta/\alpha]$. However, this solution is not acceptable because α is not in scope in Γ.

Rule (U-InstL) implements the base case for scope-safe unification. It only creates a substitution $[\sigma/\alpha]$ if α is one of the unification variables and if its instantiation does not refer to any type variables that have been introduced in the environment after α. The latter relation is captured in the auxiliary judgement $\beta >_\Gamma \alpha$. (We make an exception for unifiable type variables that have been introduced later: while the most general unifier itself may not yield a valid instantiation, it still signifies the existence of an infinite number of more specific valid instantiations.) Rule (U-InstR) is the symmetric form of (U-InstR).

Rule (U-Var) is the standard reflexivity rule. Rules (U-Fun) and (U-Rul) are standard congruence rules that combine the unifications derived for their subterms. Rule (U-Univ) is a congruence rule too, but additionally extends the environment Γ in the recursive call with the new type variable β that is in scope in the subterms.

3.6 Termination of Resolution

If we are not careful about which rules are added to the implicit 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

$$Char \Rightarrow Int, Int \Rightarrow Char \vdash_r^a Int$$

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

Termination Condition. 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 to avoid non-termination (Sulzmann et al. 2007). Adapting these restrictions to our setting, we obtain the termination judgement $\vdash_{term} \rho$ defined in Figure 8.

This judgement recursively constrains rule types $\rho_1 \Rightarrow \rho_2$ to guarantee that the recursive resolution process is well-founded. In particular, it defines a size measure $\|\rho\|$ for type terms ρ and makes sure that the size of the resolved head type decreases steadily with each recursive resolution step.

The size measure does not take universally quantified type variables into account. It assigns them size 1 but ignores the fact that the size may increase dramatically when the type variable is instantiated with a large type. The rule (T-Rule) makes up for this by requiring a size decrease for all possible instantiations of free type variables. However, rather than to specify this property non-constructively as

$$\forall\bar{\rho}: \quad \|[\bar{\alpha} \mapsto \bar{\rho}]\tau_1\| < \|[\bar{\alpha} \mapsto \bar{\rho}]\tau_2\|$$

it provides a more practical means to verify this condition by way of free variable occurrences. The number of occurrences $occ_\alpha(\tau_1)$ of free variable α in type τ_1 should be less than the number of occurrences $occ_\alpha(\tau_2)$ in τ_2. It is easy to see that the non-constructive property follows from this requirement.

Integration in the Type System. There are various ways to integrate the termination condition in the type system. The most generic approach is to require that all types satisfy the termination condition. This can be done by making the condition part of the well-formedness relation for types.

$$\boxed{\Gamma \vdash_{alg} \rho \rightsquigarrow E}$$

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

$$\boxed{\bar{\alpha}; \Gamma \vdash_{alg} \rho \rightsquigarrow E}$$

$$\text{(Alg-R-Simp)}\ \frac{\bar{\alpha}; \Gamma; \Gamma \vdash_{alg} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma \vdash_{alg} \tau \rightsquigarrow E}$$

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

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

$$\boxed{\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \tau \rightsquigarrow E}$$

$$\text{(Alg-L-RuleMatch)}\ \frac{\epsilon; \Gamma; \rho \rightsquigarrow x; \epsilon \vdash_{alg} \tau \rightsquigarrow E; \bar{\rho} \rightsquigarrow \bar{x} \qquad \bar{\alpha}; \Gamma \vdash_{alg} \bar{\rho} \rightsquigarrow \bar{E}}{\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_{alg} \tau \rightsquigarrow E[\bar{E}/\bar{x}]}$$

$$\text{(Alg-L-RuleNoMatch)}\ \frac{\bar{\alpha}; \Gamma; \rho \not\vdash_{coh} \tau \qquad \bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \tau \rightsquigarrow E'}{\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_{alg} \tau \rightsquigarrow E'}$$

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

$$\text{(Alg-L-TyVar)}\ \frac{\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \tau \rightsquigarrow E}{\bar{\alpha}; \Gamma; \Gamma', \alpha \vdash_{alg} \tau \rightsquigarrow E}$$

$$\boxed{\bar{\alpha}; \Gamma; \rho \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow E'; \Sigma'}$$

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

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

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

$$\boxed{\bar{\alpha}; \Gamma; \rho \vdash_{coh} \tau}$$

$$\text{(Coh-Simp)}\ \frac{\theta = mgu_{\Gamma;\bar{\alpha}}(\tau, \tau')}{\bar{\alpha}; \Gamma; \tau' \vdash_{coh} \tau}$$

$$\text{(Coh-TApp)}\ \frac{\bar{\alpha}, \alpha; \Gamma, \alpha; \rho \vdash_{coh} \tau}{\bar{\alpha}; \Gamma; \forall\alpha.\rho \vdash_{coh} \tau}$$

$$\text{(Coh-IApp)}\ \frac{\bar{\alpha}; \Gamma; \rho_2 \vdash_{coh} \tau}{\bar{\alpha}; \Gamma; \rho_1 \Rightarrow \rho_2 \vdash_{coh} \tau}$$

Fig. 6. Resolution Algorithm

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

$$\text{(U-InstL)}\ \frac{\alpha \in \bar{\alpha} \qquad \forall \beta \in ftv(\sigma) :\ \beta \in \bar{\alpha} \vee \beta >_\Gamma \alpha}{[\sigma/\alpha] = mgu_{\Gamma;\bar{\alpha}}(\alpha, \sigma)}$$

$$\text{(U-Var)}\ \frac{}{\epsilon = mgu_{\Gamma;\bar{\alpha}}(\beta, \beta)} \qquad \text{(U-InstR)}\ \frac{\alpha \in \bar{\alpha} \qquad \forall \beta \in ftv(\sigma) :\ \beta \in \bar{\alpha} \vee \beta >_\Gamma \alpha}{[\sigma/\alpha] = mgu_{\Gamma;\bar{\alpha}}(\sigma, \alpha)}$$

$$\text{(U-Fun)}\ \frac{\theta_1 = mgu_{\Gamma;\bar{\alpha}}(\rho_{1,1}, \rho_{2,1}) \qquad \theta_2 = mgu_{\Gamma;\bar{\alpha}}(\theta_1(\rho_{1,2}), \theta_1(\rho_{2,2}))}{\theta_2 \cdot \theta_1 = mgu_{\Gamma;\bar{\alpha}}(\rho_{1,1} \to \rho_{1,2}, \rho_{2,1} \to \rho_{2,2})}$$

$$\text{(U-Rul)}\ \frac{\theta_1 = mgu_{\Gamma;\bar{\alpha}}(\rho_{1,1}, \rho_{2,1}) \qquad \theta_2 = mgu_{\Gamma;\bar{\alpha}}(\theta_1(\rho_{1,2}), \theta_1(\rho_{2,2}))}{\theta_2 \cdot \theta_1 = mgu_{\Gamma;\bar{\alpha}}(\rho_{1,1} \Rightarrow \rho_{1,2}, \rho_{2,1} \Rightarrow \rho_{2,2})}$$

$$\text{(U-Univ)}\ \frac{\theta = mgu_{\Gamma,\beta;\bar{\alpha}}(\rho_1, \rho_2)}{\theta = mgu_{\Gamma;\bar{\alpha}}(\forall \beta.\rho_1, \forall \beta.\rho_2)}$$

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

Fig. 7. Most General Unifier

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

$$\text{(T-Rule)}\ \frac{\begin{array}{c}\vdash_{term} \rho_1 \qquad \vdash_{term} \rho_2 \\ \tau_1 = \mathrm{hd}(\rho_1) \qquad \tau_2 = \mathrm{hd}(\rho_2) \qquad \|\tau_1\| < \|\tau_2\| \\ \forall \alpha \in ftv(\rho_1) \cup ftv(\rho_2) :\ occ_\alpha(\tau_1) \leqslant occ_\alpha(\tau_2)\end{array}}{\vdash_{term} \rho_1 \Rightarrow \rho_2}$$

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

$$occ_\alpha(\beta) = \begin{cases} 1 & (\alpha = \beta) \\ 0 & (\alpha \neq \beta) \end{cases} \qquad occ_\alpha(\forall \beta.\rho) = occ_\alpha(\rho) \quad (\alpha \neq \beta)$$

$$occ_\alpha(\rho_1 \to \rho_2) = occ_\alpha(\rho_1) + occ_\alpha(\rho_2) \qquad occ_\alpha(\rho_1 \Rightarrow \rho_2) = occ_\alpha(\rho_1) + occ_\alpha(\rho_2)$$

$$\|\alpha\| = 1 \qquad \|\forall \alpha.\rho\| = \|\rho\|$$

$$\|\rho_1 \to \rho_2\| = 1 + \|\rho_1\| + \|\rho_2\| \qquad \|\rho_1 \Rightarrow \rho_2\| = 1 + \|\rho_1\| + \|\rho_2\|$$

Fig. 8. Termination Condition

4 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 contexts into explicit parameters and statically resolves all queries, much like Wadler and Blott's dictionary passing translation for type classes (Wadler and 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 4.

4.1 Type-Directed Translation

Figure 2 presents the translation rules that convert Cochis expressions into ones of System F. 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 rule 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:

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

Its 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 4. 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 rule ρ type is found in the environment, to rule (L-RuleMatch). 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-IAbs).

It falls to rule (L-RuleMatch) 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 4.1 (Type-preserving translation). *Let e be an Cochis expression, ρ be a type, Γ a type environment and E be a System F expression. If $\Gamma \vdash e : \rho \rightsquigarrow E$, then $|\Gamma| \vdash E : |\rho|$.*

Here we define the translation of the type environment form Cochis to System F as:

$$\begin{array}{rclcrcl} |\epsilon| & = & \epsilon & & |\Gamma,\alpha| & = & |\Gamma|,\alpha \\ |\Gamma,x:\rho| & = & |\Gamma|,x:|\rho| & & |\Gamma,\rho \rightsquigarrow x| & = & |\Gamma|,x:|\rho| \end{array}$$

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

Lemma 4.2 (Type-Preserving Resolution). *Let Γ be a type environment, ρ be a type and E be a System F expression. If $\Gamma \vdash^a_r \rho \rightsquigarrow E$, then $|\Gamma| \vdash E : |\rho|$.*

Moreover, we can express three key properties of Figure 4'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.

Lemma 4.3 (Soundness). *Figure 4's definition of resolution is sound (but incomplete) with respect to Figure 3's definition.*

$$\forall \Gamma,\rho,E: \qquad \Gamma \vdash_r \rho \rightsquigarrow E \quad \Rightarrow \quad \Gamma \vdash^a_r \rho \rightsquigarrow E$$

Secondly, the deterministic resolution guarantees a strong form of coherence:

Lemma 4.4 (Determinacy). *The generated evidence of resolution is uniquely determined.*

$$\forall \Gamma,\rho,E_1,E_2: \qquad \Gamma \vdash_r \rho \rightsquigarrow E_1 \ \wedge \ \Gamma \vdash_r \rho \rightsquigarrow E_2 \quad \Rightarrow \quad E_1 = E_2$$

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

Lemma 4.5 (Stability). *Resolution is stable under substitution.*

$$\forall \Gamma,\alpha,\Gamma',\sigma,\rho,E: \qquad \Gamma,\alpha,\Gamma' \vdash_r \rho \rightsquigarrow E \quad \wedge \quad \Gamma \vdash \sigma \quad \Rightarrow \quad \Gamma,\Gamma'[\sigma/\alpha] \vdash_r \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

4.2 Evidence Generation in the Algorithm

The evidence generation in Figure 6 is largely similar to that in the deterministic specification of resolution in Figure 4. With the evidence we can state the correctness of the algorithm.

Theorem 4.6 (Partial Correctness). *Let Γ be a type environment, ρ be a type and E be a System F expression. Assume that $\epsilon \vdash_{unamb} \rho$ and also $\forall \rho_i \in \Gamma : \epsilon \vdash_{unamb} \rho_i$. Then $\Gamma \vdash_r \rho \rightsquigarrow E$ if and only if $\Gamma \vdash_{alg} \rho \rightsquigarrow E$, provided that the algorithm terminates.*

4.3 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 and Lumsdaine 2005), this dynamic semantics is:

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

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

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

Theorem 4.7 (Type Safety). *If $\epsilon \vdash e : \rho$, then $eval(e) = V$ for some System F value V.*

The proof follows trivially from Theorem 4.1.

5 RELATED WORK

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

5.1 Implicit Programming with Local Scoping, Coherence but no Overlapping Rules

Our work allows a very flexible model of implicits with first-class rules, higher-order rules and nested scoping with overlapping rules while guaranteeing coherence. Closest to our work in terms of combining additional flexibility with coherence are *modular type classes* (Dreyer et al. 2007) and System F_G (Siek and Lumsdaine 2005). Both works preserve coherence in the presence of local scoping, but (unlike Cochis) the local scopes *forbid overlapping rules*. The restriction of no overlapping rules is an essential part of guaranteeing coherence. 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 and 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.

5.2 Implicit Programming with Coherence and 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/rules. 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 (Damas and Milner 1982; Hindley 1969; Milner 1978) 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 (Dijkstra and Swierstra 2005; Kahl and Scheffczyk 2001) in the context of Haskell type classes. Both *named instances* (Kahl and Scheffczyk 2001) and *Explicit Haskell* (Dijkstra and Swierstra 2005) preserve most design choices taken in type clases (including global scoping), but allow instances that 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, but it still does not allow different instances for the same type to coexist in different scopes of a program. CochiS has a different approach to overlapping compared to *instance chains* (Morris and 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. 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.

5.3 Implicit Programming without Coherence

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 coherence. Programs similar to that in Figure 1 can be written in the implicit calculus and there is no way to detect incoherence. 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:

$$Char \Rightarrow Bool, Bool \Rightarrow Int \vdash_r^a Char \Rightarrow Int$$

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. *Scala implicits* (Odersky 2010; Oliveira et al. 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 rules) is supported. However Scala implicits are incoherent and they do not allow higher-order rules either.

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 and Oury 2008) and canonical structures (Gonthier et al. 2011), Agda's instance arguments (Devriese and Piessens 2011) and Idris type classes (Brady 2015) all allow multiple and/or highly overlapping rules/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 and 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. Like implicits, the design of Idris type classes allows for any

type of value to be implicit. Thus type classes in Idris are first-class, can be manipulated as any other value, an also allow multiple (incoherent) instances of the same type. *Instance arguments* (Devriese and 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, implicit rules are not declared explicitly. Instead rules are drawn directly from the regular type environment, and any previously defined declaration can be used as a rule. 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 rules.

5.4 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 rules for 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. It should be easy enough to design source languages on top of Cochis that forbid local scoping constructs and, instead, make all declared rules visible in a single global environment. This would retain several of benefits of Cochis (such as first-class, higher-order rules, and coherent overlapping rules), while providing a form of global uniqueness. However this design would still be 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.

5.5 Focused Proof Search

Part of the syntax-directedness of our deterministic resolution is very similar to that obtained by *focusing* in proof search (Liang and Miller 2009; Miller et al. 1991; Pfenning 2010). Both approaches alternate a phase that is syntax directed on a "query" formula (our first auxiliary judgement), with a phase that is syntax directed on a given formula (our third auxiliary judgement). 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.

6 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 coherence. In particular, Cochis supports local scoping, overlapping rules, first-class rules, and higher-order rules, while remaining type safe, coherent and unambiguous. 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. 2005a,b) and *type families* (Schrijvers et al. 2008).

REFERENCES

H. Barendregt. 1981. *The Lambda Calculus: its Syntax and Semantics, volume 103 of Studies in Logic and the Foundations of Mathematics*. North-Holland.

Hans-J. Boehm. 1985. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*. IEEE, 339–345.

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

C. Camarão and L. Figueiredo. 1999. Type Inference for Overloading without Restrictions, Declarations or Annotations. In *FLOPS*. Springer-Verlag, London, UK, 37–52.

M. Chakravarty, G. Keller, and S. L. Peyton Jones. 2005a. Associated type synonyms. In *ICFP*. ACM, New York, NY, USA, 241–253.

M. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. 2005b. Associated types with class. In *POPL*. ACM, New York, NY, USA, 1–13.

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM, New York, NY, USA, 207–212.

D. Devriese and F. Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *ICFP*. ACM, New York, NY, USA, 143–155.

A. Dijkstra and S. D. Swierstra. 2005. *Making Implicit Parameters Explicit*. Technical Report. Utrecht University.

D. Dreyer, R. Harper, M. Chakravarty, and G. Keller. 2007. Modular type classes. In *POPL*. ACM, New York, NY, USA, 63–70.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *POPL*. ACM, New York, NY, USA, 671–683.

Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. 2007. An Extended Comparative Study of Language Support for Generic Programming. *J. Funct. Program.* 17, 2 (March 2007).

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to Make Ad Hoc Proof Automation Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 163–175.

D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. 2006. Concepts: linguistic support for generic programming in C++. In *OOPSLA*. ACM, New York, NY, USA, 291–310.

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.

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

M. P. Jones. 1992. A Theory of Qualified Types. In *ESOP*.

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

M. P. Jones. 1995b. Simplifying and improving qualified types. In *FPCA*.

W. Kahl and J. Scheffczyk. 2001. Named Instances for Haskell Type Classes. In *Haskell Workshop*.

Oleg Kiselyov and Chung-chieh Shan. 2004. Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*.

Edward Kmett. 2015. Type Classes vs the World. https://www.youtube.com/watch?v=hIZxTQP1ifo. (2015).

J. Lewis, J. Launchbury, E. Meijer, and M. Shields. 2000. Implicit parameters: dynamic scoping with static types. In *POPL*. ACM, New York, NY, USA, 108–118.

Chuck Liang and Dale Miller. 2009. Focusing and Polarization in Linear, Intuitionistic, and Classical Logics. *Theor. Comput. Sci.* 410, 46 (2009), 4747–4768.

Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121.

Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282.

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

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375.

J. G. Morris and M. P. Jones. 2010. Instance chains: type class programming without overlapping instances. In *ICFP*. ACM, New York, NY, USA, 375–386.

Team Mozilla Research. 2017. *The Rust Programming Language*. https://www.rust-lang.org/en-US/.

M. Odersky. 2010. The Scala Language Specification, Version 2.8. (2010). http://www.scala-lang.org/docu/files/ScalaReference.pdf

B. C. d. S. Oliveira, A. Moors, and M. Odersky. 2010. Type classes as objects and implicits. In *OOPSLA*. ACM, New York, NY, USA.

B. C. d. S. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming. In *PLDI '12*. ACM, New York, NY, USA.

S. L. Peyton Jones, M. P. Jones, and E. Meijer. 1997. Type classes: exploring the design space. In *Haskell Workshop*.

Frank Pfenning. 1993. On the Undecidability of Partial Polymorphic Type Reconstruction. *Fundam. Inform.* 19, 1/2 (1993), 185–199.

Frank Pfenning. 2010. Lecture Notes on Focusing, Oregon Summer School 2010, Proof Theory Foundations. (2010). https://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press, Cambridge, MA, USA.
John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '91)*. Springer-Verlag, London, UK, UK, 675–700. http://dl.acm.org/citation.cfm?id=645867.670915
T. Schrijvers, S. L. Peyton Jones, M. Chakravarty, and M. Sulzmann. 2008. Type checking with open type functions. In *ICFP*. ACM, New York, NY, USA.
J. G. Siek. 2011. The C++0x fiConceptsfi Effort. http://ecee.colorado.edu/~siek/concepts_effort.pdf. (2011).
J. G. Siek and A. Lumsdaine. 2005. Essential language support for generic programming. In *PLDI*. ACM, New York, NY, USA.
M. Sozeau and N. Oury. 2008. First-Class Type Classes. In *TPHOLs*.
M. Sulzmann, G. Duck, S. L. Peyton Jones, and P. J. Stuckey. 2007. Understanding functional dependencies via Constraint Handling Rules. *Journal of Functional Programming* 17 (2007), 83–129.
Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.
P. L. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *POPL*. ACM, New York, NY, USA.
S. Wehr, R. Lämmel, and P. Thiemann. 2007. JavaGI: Generalized interfaces for java. In *ECOOP*.
Edward Zhang. 2014. Type classes: confluence, coherence and global uniqueness. http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/. (2014).
Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, Flexible Object-oriented Generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*.

$\boxed{\Gamma \vdash T}$

(F-WF-VarTy) $$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}$$

(F-WF-FunTy) $$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 \to T_2}$$

(F-WF-UnivTy) $$\frac{\Gamma, \alpha \vdash T}{\Gamma \vdash \forall \alpha . T}$$

$\boxed{\Gamma \vdash E : T}$

(F-Var) $$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

(F-Abs) $$\frac{\Gamma, x : T_1 \vdash E : T_2 \qquad \Gamma \vdash T_1}{\Gamma \vdash \lambda x : T_1 . E : T_1 \to T_2}$$

(F-App) $$\frac{\Gamma \vdash E_1 : T_2 \to T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1\, E_2 : T_1}$$

(F-TApp) $$\frac{\Gamma \vdash E : \forall \alpha . T_2 \qquad \Gamma \vdash T_1}{\Gamma \vdash E\, T_1 : T_2[T_1/\alpha]}$$

(F-TAbs) $$\frac{\Gamma, \alpha \vdash E : T}{\Gamma \vdash \Lambda \alpha . E : \forall \alpha . T}$$

Fig. 9. System F Type System

APPENDIX

B PROOFS

Throughout the proofs we refer to the type system rules of System F listed in Figure 9.

B.1 Type Preservation

Lemma B.1 states that the translation preserves the well-formedness of types.

LEMMA B.1. *If*

$$\Gamma \vdash \rho$$

then

$$|\Gamma| \vdash |\rho|$$

PROOF. By structural induction on the expression and corresponding inference rule.

(WF-VarTy) $\Gamma \vdash \alpha$

It follows from the rule that $\alpha \in \Gamma$. Hence, obviously $\alpha \in |\Gamma|$. Finally, by rule (F-WF-VARTY), and taking into account that $|\alpha| = \alpha$, we conclude

$$|\Gamma| \vdash \alpha$$

(WF-FunTy) $\Gamma \vdash \rho_1 \to \rho_2$

It follows from the induction hypotheses and the hypotheses of the rule that

$$|\Gamma| \vdash |\rho_1| \quad \wedge \quad |\Gamma| \vdash |\rho_2|$$

Hence, by rule (F-WF-FUNTY), and taking into account that $|\rho_1| \to |\rho_2| = |\rho_1 \to \rho_2|$, we conclude that

$$|\Gamma| \vdash |\rho_1 \to \rho_2|$$

(WF-RulTy) $\Gamma \vdash \rho_1 \Rightarrow \rho_2$

It follows from the induction hypotheses and the hypotheses of the rule that

$$|\Gamma| \vdash |\rho_1| \quad \wedge \quad |\Gamma| \vdash |\rho_2|$$

Hence, by rule (F-WF-FUNTY), and taking into account that $|\rho_1| \to |\rho_2| = |\rho_1 \Rightarrow \rho_2|$, we conclude that

$$|\Gamma| \vdash |\rho_1 \Rightarrow \rho_2|$$

(WF-UnivTy) $\Gamma \vdash \forall\alpha.\rho$

It follows from the induction hypothesis and the hypothesis of the rule that

$$|\Gamma, \alpha| \vdash |\rho|$$

As $|\Gamma, \alpha| = |\Gamma|, \alpha$, we can simplify this to

$$|\Gamma|, \alpha \vdash |\rho|$$

Hence, by rule (F-WF-UNIVTY), and taking into account that $\forall\alpha.|\rho| = |\forall\alpha.\rho|$, we conclude that

$$|\Gamma| \vdash |\forall\alpha.\rho|$$

□

Lemma B.2 states that the translation of expressions to System F preserves types. Its proof relies on Lemma B.3, which states that the translation of resolution preserves types.

Lemma B.2. *If*

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

then

$$|\Gamma| \vdash E : |\rho|$$

Proof. By structural induction on the expression and corresponding inference rule.

(Ty-Var) $\Gamma \vdash x : \rho \rightsquigarrow x$

It follows from (Ty-Var) that

$$(x : \rho) \in \Gamma$$

Based on the definition of $|\cdot|$ it follows

$$(x : |\rho|) \in |\Gamma|$$

Thus we have by (F-Var) that

$$|\Gamma| \vdash x : |\rho|$$

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

The first hypothesis of (Ty-Abs) is that

$$\Gamma, x : \rho_1 \vdash e : \rho_2 \rightsquigarrow E$$

and thus by the induction hypothesis we have that

$$|\Gamma|, x : |\rho_1| \vdash E : |\rho_2|$$

The second hypothesis of (Ty-Abs) is that

$$\Gamma \vdash |\rho_1|$$

and thus by Lemma B.1 we have that

$$|\Gamma| \vdash |\rho_1|$$

Hence, by (F-Abs) we conclude

$$|\Gamma| \vdash \lambda x : |\rho_1| . E : |\rho_1 \to \rho_2|$$

(Ty-App) $\Gamma \vdash e_1\ e_2 : \rho_1 \rightsquigarrow E_1\ E_2$

By the induction hypothesis, we have:

$$|\Gamma| \vdash E_1 : |\rho_2 \to \rho_1| \quad \wedge \quad |\Gamma| \vdash E_2 : |\rho_2|$$

and, because $|\rho_2 \to \rho_1| = |\rho_2| \to |\rho_1|$, we can write the former as

$$|\Gamma| \vdash E_1 : |\rho_2| \to |\rho_1|$$

Then it follows by (F-App) that

$$|\Gamma| \vdash E_1\ E_2 : |\rho_1|$$

(Ty-TAbs) $\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E$

Based on (Ty-TAbs) and the induction hypothesis, we have

$$|\Gamma, \alpha| \vdash E : |\rho|$$

Thus, based on (F-TAbs) and because $|\Gamma, \alpha| = |\Gamma|, \alpha$, we have

$$|\Gamma| \vdash \Lambda\alpha.E : \forall\alpha.|\rho|$$

or, because $|\forall\alpha.\rho| = \forall\alpha.|\rho|$, we conclude

$$|\Gamma| \vdash \Lambda\alpha.E : |\forall\alpha.\rho|$$

(Ty-TApp) $\Gamma \vdash e\ \rho_1 : \rho_2[\rho_1/\alpha] \rightsquigarrow E\ |\rho_1|$

By the first hypothesis of the rule and the induction hypothesis of the lemma, it follows that

$$|\Gamma| \vdash E : |\forall\alpha.\rho_2|$$

From this we have by definition of $|\cdot|$

$$|\Gamma| \vdash E : \forall\alpha.|\rho_2|$$

By the second hypothesis of the rule and Lemma B.1 we also have

$$|\Gamma| \vdash |\rho_1|$$

It then follows from (F-TApp) that

$$|\Gamma| \vdash E\ |\rho_1| : |\rho_2|[|\rho_1|/\alpha]$$

This is easily seen to be equivalent to

$$|\Gamma| \vdash E\ |\rho_1| : |\rho_2[\rho_1/\alpha]|$$

(Ty-IAbs) $\Gamma \vdash \lambda_?\rho_1.e : \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

Based on the first hypothesis of the rule and the induction hypothesis, we have

$$|\Gamma, \rho_1 \rightsquigarrow x| \vdash E : |\rho_2|$$

or, using the definition of $|\cdot|$,

$$|\Gamma|, x : |\rho_1| \vdash E : |\rho_2|$$

Based on the second hypothesis of the rule and Lemma B.1 we have

$$|\Gamma| \vdash |\rho_1|$$

Thus, based on (F-Abs) we have

$$|\Gamma| \vdash \lambda x : |\rho_1|.E : |\rho_1| \to |\rho_2|$$

or, using the definition of $|\cdot|$ again,

$$|\Gamma| \vdash \lambda x : |\rho_1|.E : |\rho_1 \Rightarrow \rho_2|$$

(Ty-IApp) $\Gamma \vdash e_1 \textbf{ with } e_2 : \rho_1 \rightsquigarrow E_1\, E_2$

From the hypotheses of the rule and the induction hypothesis we have:

$$|\Gamma| \vdash E_1 : |\rho_2 \Rightarrow \rho_1| \quad \wedge \quad |\Gamma| \vdash E_2 : |\rho_2|$$

Based on the definition of $|\cdot|$, the first of these means

$$|\Gamma| \vdash E_1 : |\rho_2| \to |\rho_1|$$

Finally, based on (F-App), we know

$$|\Gamma| \vdash E_1\, E_2 : |\rho_1|$$

(Ty-Query) $\Gamma \vdash ?\rho : \rho \rightsquigarrow E$

Based on the first hypothesis of the rule and Lemma B.3 we know

$$|\Gamma| \vdash E : |\rho|$$

□

Lemma B.3. *If*

$$\Gamma \vdash_r^a \rho \rightsquigarrow E$$

and

$$\Gamma \vdash \rho$$

then

$$|\Gamma| \vdash E : |\rho|$$

Proof. By induction on the derivation.

(AR-TAbs) $\Gamma \vdash_r^a \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E$

From the hypothesis of the rule and the induction hypothesis, we have

$$|\Gamma, \alpha| \vdash E : |\rho|$$

or alternatively, based on the definition of $|\cdot|$,

$$|\Gamma|, \alpha \vdash E : |\rho|$$

Then, rule (F-TAbs) allows us to conclude

$$|\Gamma| \vdash \Lambda\alpha.E : \forall\alpha.|\rho|$$

or, again based on the definition of $|\cdot|$,

$$|\Gamma| \vdash \Lambda\alpha.E : |\forall\alpha.\rho|$$

(AR-TApp) $\Gamma \vdash_r^a \rho[\sigma/\alpha] \rightsquigarrow E\,|\sigma|$

From the first hypothesis of the rule and the induction hypothesis, we have

$$|\Gamma| \vdash E : |\forall\alpha.\rho|$$

or alternatively, based on the definition of $|\cdot|$,

$$|\Gamma| \vdash E : \forall\alpha.|\rho|$$

From the second hypothesis of the rule and Lemma B.1, we have

$$|\Gamma| \vdash |\sigma|$$

Then, rule (F-TApp) allows us to conclude

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

or, again based on the definition of $|\cdot|$,

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

(AR-IVar) $\quad \Gamma \vdash^a_r \rho \rightsquigarrow x$

From the hypothesis of the rule and the definition of $|\cdot|$, we have

$$(x : |\rho|) \in |\Gamma|$$

Thus, using rule (F-Var), we can conclude

$$|\Gamma| \vdash x : |\rho|$$

(AR-IAbs) $\quad \Gamma \vdash^a_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

From the first hypothesis of the rule and the induction hypothesis, we have

$$|\Gamma, \rho_1 \rightsquigarrow x| \vdash E : |\rho_2|$$

or alternatively, based on the definition of $|\cdot|$,

$$|\Gamma|, x : |\rho_1| \vdash E : |\rho_2|$$

Then, rule (F-Abs) allows us to conclude

$$|\Gamma| \vdash \lambda x : |\rho_1|.E : |\rho_1| \to |\rho_2|$$

or, again based on the definition of $|\cdot|$,

$$|\Gamma| \vdash \lambda x : |\rho_1|.E : |\rho_1 \Rightarrow \rho_2|$$

(AR-IApp) $\quad \Gamma \vdash^a_r \rho_2 \rightsquigarrow E_2\,E_1$

From the hypotheses of the rule and the induction hypothesis, we have

$$|\Gamma| \vdash E_1 : |\rho_1| \quad \wedge \quad |\Gamma| \vdash E_2 : |\rho_1 \Rightarrow \rho_2|$$

The second conjunct can be reformulated, based on the definition of $|\cdot|$, to

$$|\Gamma| \vdash E_2 : |\rho_1| \to |\rho_2|$$

Then, rule (F-App) allows us to conclude

$$|\Gamma| \vdash E_2\,E_1 : |\rho_2|$$

□

B.2 Auxiliary Lemmas About Non-Determistic Resolution

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

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

Lemma B.4 (Weakening). *If*

$$\Gamma, \Gamma' \vdash^a_r \rho \rightsquigarrow E$$

then

$$\Gamma, \Gamma'', \Gamma' \vdash^a_r \rho \rightsquigarrow E$$

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

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

Lemma B.5 (Substitution). *If*

$$\Gamma, \rho \rightsquigarrow x, \Gamma' \vdash^a_r \rho' \rightsquigarrow E'$$

and

$$\Gamma \vdash^a_r \rho \rightsquigarrow E$$

then

$$\Gamma, \Gamma' \vdash^a_r \rho' \rightsquigarrow E'[E/x]$$

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

The key case is the one for rule (AR-IVar) where $\rho' = \rho$ and $E' = x$. In this case the second hypothesis gives us

$$\Gamma \vdash^a_r \rho \rightsquigarrow E$$

As $E = x[E/x]$, this also means

$$\Gamma \vdash^a_r \rho \rightsquigarrow x[E/x]$$

Finally, we can apply the Weakening Lemma B.4 to obtain the desired result.

$$\Gamma, \Gamma' \vdash^a_r \rho \rightsquigarrow x[E/x]$$

All other cases are straightforward. □

B.3 Soundness of Deterministic Resolution

Lemma B.6 states that deterministic resolution is sound with respect to non-deterministic resolution.

Lemma B.6. *If*

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

then

$$\Gamma \vdash^a_r \rho \rightsquigarrow E$$

Proof. The lemma immediately follows from Lemma B.7. □

Lemma B.7. *If*

$$\bar{\alpha}; \Gamma \vdash_r \rho \rightsquigarrow E$$

then

$$\Gamma \vdash^a_r \rho \rightsquigarrow E$$

Proof. The proof proceeds by induction on the derivation.

(R-IAbs) $\bar{\alpha}; \Gamma \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

From the first assumption of rule (R-IAbs) and the induction hypothesis, we have

$$\Gamma, x : \rho_1 \vdash_r^a \rho_2 \rightsquigarrow E$$

Hence, from rule (AR-IAbs) and the freshness condition on x in rule (R-IAbs) it follows that

$$\Gamma \vdash_r^a \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$$

(R-TAbs) $\bar{\alpha}; \Gamma \vdash_r \forall \alpha.\rho \rightsquigarrow \Lambda \alpha.E$

From the precondition of rule (R-TAbs) and the induction hypothesis, we have

$$\Gamma, \alpha \vdash_r^a \rho \rightsquigarrow E$$

Hence, from rule (AR-TAbs) it follows that

$$\Gamma \vdash_r^a \forall \alpha.\rho \rightsquigarrow \Lambda \alpha.E$$

(R-Simp) $\bar{\alpha}; \Gamma \vdash_r \tau \rightsquigarrow E$

From the precondition of the rule, Lemma B.8 and the simple fact that $\Gamma \subseteq \Gamma$, it follows that

$$\Gamma \vdash_r^a \tau \rightsquigarrow E$$

□

The above proof relies on the following auxiliary lemma for the resolution of simple types. The proof of this auxiliary lemma proceeds by mutual induction with the proof of the main lemma.

Lemma B.8. *If*

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E$$

and

$$\Gamma' \subseteq \Gamma$$

then

$$\Gamma \vdash_r^a \tau \rightsquigarrow E$$

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

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

The first assumption of the rule is

$$\Gamma; \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E; \bar{E} \rightsquigarrow \bar{x}$$

From the lemma's assumption $(\Gamma', \rho \rightsquigarrow x) \subseteq \Gamma$ we conclude $(\rho \rightsquigarrow x) \in \Gamma$. Hence, by rule (AR-Simp) we have

$$\Gamma \vdash_r^a \rho \rightsquigarrow x$$

From the second precondition of the rule and the (mutual) induction hypothesis, we also have

$$\Gamma \vdash_r^a \bar{\rho} \rightsquigarrow \bar{E}$$

The above three observations allow us to invoke the auxiliary Lemma B.9 and conclude

$$\Gamma \vdash_r^a \tau \rightsquigarrow E[\bar{E}/\bar{x}]$$

(L-Var), (L-TyVar), (L-RuleNoMatch)

Trivially by applying the induction hypothesis on the precondition of the rule.

□

The above proof relies on the following auxiliary lemma.

Lemma B.9. *If*

$$\Gamma; \rho \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \bar{\rho} \rightsquigarrow \bar{x}$$

and

$$\Gamma \vdash_r^a \rho \rightsquigarrow E$$

and

$$\Gamma \vdash_r^a \bar{\rho} \rightsquigarrow \bar{E}$$

then

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

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

(M-Simp) $\Gamma; \tau \rightsquigarrow E \vdash_r \tau \rightsquigarrow E; \epsilon$

The first assumption of the lemma is the desired conclusion

$$\Gamma \vdash_r^a \tau \rightsquigarrow E$$

(M-IApp) $\Gamma; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \bar{\rho} \rightsquigarrow \bar{x}, \rho_1 \rightsquigarrow x$

The third hypothesis of the lemma then is

$$\Gamma \vdash_r^a \bar{\rho} \rightsquigarrow \bar{E} \quad \wedge \quad \Gamma \vdash_r^a \rho_1 \rightsquigarrow E_1$$

The Weakening Lemma B.4 turns the first conjunct into

$$\Gamma, \rho_1 \rightsquigarrow x \vdash_r^a \bar{\rho} \rightsquigarrow \bar{E}$$

The second hypothesis of the lemma then is

$$\Gamma \vdash_r^a \rho_1 \Rightarrow \rho_2 \rightsquigarrow E$$

By applying the Weakening Lemma B.4 we get

$$\Gamma, \rho_1 \rightsquigarrow x \vdash_r^a \rho_1 \Rightarrow \rho_2 \rightsquigarrow E$$

From rule (AR-IVar) we can also conclude

$$\Gamma, \rho_1 \rightsquigarrow x \vdash_r^a \rho_1 \rightsquigarrow x$$

These two facts allow us to derive from rule (AR-IApp)

$$\Gamma, \rho_1 \rightsquigarrow x \vdash_r^a \rho_2 \rightsquigarrow E\, x$$

We now have the necessary ingredients to invoke the induction hypothesis on the hypothesis of the rule and obtain

$$\Gamma, \rho_1 \rightsquigarrow x \vdash_r^a \tau \rightsquigarrow E'[\bar{E}/\bar{x}]$$

Finally, we use the second conjunct of the third hypothesis to invoke the Substitution Lemma B.5 on the above and reach our desired conclusion

$$\Gamma \vdash_r^a \tau \rightsquigarrow E'[\bar{E}/\bar{x}][E_1/x]$$

(M-TApp) $\Gamma; \forall\alpha.\rho \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \bar{\rho} \rightsquigarrow \bar{x}$

Then the second hypothesis of the lemma is

$$\Gamma \vdash_r^a \forall\alpha.\rho \rightsquigarrow E$$

This allows us to conclude by rule (AR-TApp) that

$$\Gamma \vdash_r^a \rho[\sigma/\alpha] \rightsquigarrow E\,|\sigma|$$

The third hypothesis of the lemma is

$$\Gamma \vdash_r^a \bar{\rho} \rightsquigarrow \bar{E}$$

We now have the necessary ingredients to invoke the induction hypothesis on the hypothesis of the rule and obtain the desired conclusion

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

□

B.4 Deterministic Resolution is Deterministic

Lemma B.10. *If*

$$\vdash_{unamb} \Gamma$$

and

$$\vdash_{unamb} \rho$$

and

$$\Gamma \vdash_r \rho \rightsquigarrow E_1$$

and

$$\Gamma \vdash_r \rho \rightsquigarrow E_2$$

then

$$E_1 = E_2$$

Proof. From the third and fourth hypotheses of the lemma, the hypothesis of rule (R-Main) and Lemma B.11 the desired result follows

$$E_1 = E_2$$

□

Lemma B.11. *If*

$$\vdash_{unamb} \Gamma$$

and

$$\vdash_{unamb} \rho$$

and

$$\bar{\alpha}; \Gamma \vdash_r \rho \rightsquigarrow E_1$$

and

$$\bar{\alpha}; \Gamma \vdash_r \rho \rightsquigarrow E_2$$

then

$$E_1 = E_2$$

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

(R-IAbs) $\bar{\alpha}; \Gamma \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : ||\rho_1||.E_1$

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

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

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

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

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) $\bar{\alpha}; \Gamma \vdash_r \forall \alpha.\rho \rightsquigarrow \Lambda \alpha.E_1$

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

$$\vdash_{unamb} \rho$$

From the lemma's first hypothesis, it follows that

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

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) $\bar{\alpha}; \Gamma \vdash_r \tau \rightsquigarrow E_1$

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

$$E_1 = E_2$$

□

Lemma B.12. *If*

$$\vdash_{unamb} \Gamma$$

and

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

and

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E_1$$

and

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_r \tau \rightsquigarrow E_2$$

then

$$E_1 = E_2$$

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

(L-RuleMatch) $\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E_1[\bar{E}_1/\bar{x}]$

Then the fourth hypothesis was either derived from rule (L-RuleMatch), or from rule (L-RuleNoMatch). However, the hypothesis of the latter is not satisfied: $\epsilon; E_1; \Sigma_1$ forms a counter-example. Hence, the fourth hypothesis is also formed by rule (L-RuleMatch).

Then it follows from the first hypothesis of the rule and Lemma B.13 that

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

From the second hypothesis of the rule and Lemma B.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-RuleNoMatch) $\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E_1'$

Then the fourth hypothesis was either derived from rule (L-RuleMatch), or from rule (L-RuleNoMatch). 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-RuleNoMatch).

From the second hypothesis of the lemma we derive $\vdash_{unamb} \Gamma'$. Then from the second hypothesis of the rule and the induction hypothesis we conclude the desired result

$$E_1' = E_2'$$

(L-Var) $\bar{\alpha}; \Gamma; \Gamma', x : \rho \vdash_r \tau \rightsquigarrow E_1$

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

$$E_1 = E_2$$

(L-TyVar) $\bar{\alpha}; \Gamma; \Gamma', \alpha \vdash_r \tau \rightsquigarrow E_1$

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

$$E_1 = E_2$$

□

We annotated the judgement with the sequence of substitution types $\bar{\sigma}$ used to instantiate the universal quantifiers.

$$\boxed{\bar{\sigma}; \Gamma; \rho \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \Sigma}$$

$$\text{(M-Simp)} \quad \epsilon; \Gamma; \tau \rightsquigarrow E \vdash_r \tau \rightsquigarrow E; \epsilon$$

$$\text{(M-IApp)} \quad \frac{\bar{\sigma}; \Gamma, \rho_1 \rightsquigarrow x; \rho_2 \rightsquigarrow E\, x \vdash_r \tau \rightsquigarrow E'; \Sigma \qquad x \text{ fresh}}{\bar{\sigma}; \Gamma; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E \vdash_r \tau \rightsquigarrow E'; \Sigma, \rho_1 \rightsquigarrow x}$$

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

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

The judgement is deterministic.

Lemma B.13. *If*

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

and

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

and

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

then

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

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

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

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

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

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

$$\Sigma_1 = \epsilon = \Sigma_2$$

and trivially

$$\vdash_{unamb} \epsilon$$

(UA-IAbs) $\bar{\alpha} \vdash_{unamb} \rho_1 \Rightarrow \rho_2$

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 $\vdash_{unamb} \Sigma_1$. Since $\bar{\alpha} \vdash_{unamb} \rho_1 \Rightarrow \rho_2$, we also have $\vdash_{unamb} \rho_1$. Hence we also conclude

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

(UA-TAbs) $\bar{\alpha} \vdash_{unamb} \forall \alpha.\rho$

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 $\bar{\sigma}_1' = \bar{\sigma}_{1,1}', \sigma_{2,2}'$. From their two hypotheses and from the hypothesis of the rule and the induction hypothesis, we obtain

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

From this we conclude the desired result

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

□

B.5 Resolution Coherence

Deterministic resolution is stable under substitution.

Lemma B.14. *If*

$$\Gamma, \alpha, \Gamma' \vdash_r \rho \rightsquigarrow E$$

and

$$\Gamma \vdash \sigma$$

then

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash_r \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

Proof. The hypothesis of rule then is

$$ftv(\Gamma), \alpha, ftv(\Gamma'); \Gamma, \alpha, \Gamma' \vdash_r \rho \rightsquigarrow E$$

From Lemma B.15 it follows that

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

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

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash_r \rho[\sigma/\alpha] \rightsquigarrow E[|\sigma|/\alpha]$$

□

Lemma B.15. *If*

$$\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma' \vdash_r \rho \rightsquigarrow E$$

and

$$\Gamma \vdash \sigma$$

then

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

Proof. **(R-IAbs)** $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma' \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

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

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

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

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

(R-TAbs) $\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma' \vdash_r \forall\beta.\rho \rightsquigarrow \Lambda\beta.E$

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

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

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

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

(R-Simp) $\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma' \vdash_r \tau \rightsquigarrow E$

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

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

and

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

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

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

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

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

□

Lemma B.16. *If*

$$\bar{\alpha},\alpha,\bar{\alpha}';\Gamma,\alpha,\Gamma';\Gamma'' \vdash_r \tau \rightsquigarrow E$$

and

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

and

$$\Gamma \vdash \sigma$$

then

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

and

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

where

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

$$\text{(R-1)}\ \frac{}{R(\Gamma_1,\Gamma_2;\alpha;\Gamma';\Gamma_1;\Gamma_1;\sigma)}$$

$$\text{(R-2)}\ \frac{}{R(\Gamma;\alpha;\Gamma_1',\Gamma_2';\Gamma,\alpha,\Gamma_1';\Gamma,\Gamma_1'[\sigma/\alpha];\sigma)}$$

Proof. **(L-RuleMatch)** $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; \Gamma'', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E$

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

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

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

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

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

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

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-RuleNoMatch) $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; \Gamma'', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E'$

The rule's first hypothesis states that

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

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

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

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-RuleNoMatch) 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 judgement.

(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) $\bar{\alpha}, \alpha, \bar{\alpha}'; \Gamma, \alpha, \Gamma'; \Gamma'', x : \rho \vdash_r \tau \rightsquigarrow E$
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 judgement.

(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 judgement.

(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) $\Gamma_2 = \epsilon$
It follows that $\alpha = \beta$ and we can establish by means of (R-2) that

$$R(\Gamma_1, \Gamma_2; \alpha; \epsilon, \Gamma'; \Gamma_1, \Gamma_2, \alpha; \Gamma_1, \Gamma_2, \epsilon[\sigma/\alpha]; \sigma)$$

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

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

(b) $\Gamma_2 \neq \epsilon$
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 B.17. *If*

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

and

$$\Gamma \vdash \sigma$$

then

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

Proof. **(M-Simp)** $\Gamma, \alpha, \Gamma'; \tau \rightsquigarrow E \vdash_r \tau \rightsquigarrow E; \epsilon$

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

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

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

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

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

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

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

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

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

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

From the rule's second hypothesis (and the preservation of well-typing under type-susbstitution) we have

$$\Gamma, \Gamma'[\sigma/\alpha] \vdash \sigma'[\sigma'/\alpha]$$

From these two facts we conclude by rule (M-TApp), reasoning modulo the definition of substitution

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

□

B.6 Soundness of the Algorithm wrt Deterministic Resolution

Lemma B.18. *If*

$$\Gamma \vdash_{alg} \rho \rightsquigarrow E$$

then

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

Proof. From the hypothesis it follows that

$$\mathit{tyvars}(\Gamma); \Gamma \vdash_{alg} \rho \rightsquigarrow E$$

Hence, by Lemma B.19 and rule (R-Main) the desired conclusion follows

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

□

Lemma B.19. *If*

$$\bar{\alpha}; \Gamma \vdash_{alg} \rho \rightsquigarrow E$$

then

$$\bar{\alpha}; \Gamma \vdash_r \rho \rightsquigarrow E$$

Proof. The lemma follows from the isomorphism between the rule sets of the two judgements and from Lemma B.20. □

LEMMA B.20. *If*

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \rho \rightsquigarrow E$$

then

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_{r} \rho \rightsquigarrow E$$

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

(Alg-L-RuleMatch) $\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_{alg} \tau \rightsquigarrow E[\bar{E}/\bar{x}]$

From the rule's first hypothesis and Lemma B.21 we have

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

Then, using Lemma B.19 and rule (L-RULEMATCH) we conclude

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

(Alg-L-RuleNoMatch) $\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_{alg} \tau \rightsquigarrow E'$

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

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

Then from the rule's first hypothesis and the negation of Lemma B.22, we have:

$$\nexists E, \Sigma : \bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \epsilon \vdash_{alg} \tau \rightsquigarrow E; \Sigma$$

By Lemma B.21 we thus have

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

Hence with rule (L-RULENOMATCH) we conclude

$$\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_{r} \tau \rightsquigarrow E'$$

(Alg-L-Var) $\bar{\alpha}; \Gamma; \Gamma', x : \rho \vdash_{alg} \tau \rightsquigarrow E$

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

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

By rule (L-VAR) we conclude

$$\bar{\alpha}; \Gamma; \Gamma', x : \rho \vdash_{r} \tau \rightsquigarrow E$$

(Alg-L-TyVar) $\bar{\alpha}; \Gamma; \Gamma', \alpha \vdash_{alg} \tau \rightsquigarrow E$

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

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

By rule (L-TYVAR) we conclude

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

□

We assume that the judgement is decorated with an additional argument, the substitution for the $\bar{\alpha}$ type variables.

Lemma B.21. *If*

$$\bar{\alpha}; \Gamma; \rho \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow E'; \Sigma', \theta(\Sigma); \theta$$

and

$$dom(\theta) \subseteq \bar{\alpha}$$

then

$$\theta(\Gamma); \theta(\rho) \rightsquigarrow |\theta|(E) \vdash_r \theta(\tau) \rightsquigarrow E'; \Sigma'$$

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

(Alg-M-Simp) $\bar{\alpha}; \Gamma; \tau' \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow |\theta|(E); \epsilon, \theta(\Sigma); \theta$

From the hypothesis of the rule and Lemma B.23 it follows that $\theta(\tau') = \theta(\tau)$. Hence, the target judgement can be rewritten as

$$\theta(\Gamma); \theta(\tau') \rightsquigarrow |\theta|(E) \vdash_r \theta(\tau') \rightsquigarrow |\theta|(E); \epsilon$$

This follows from rule (M-Simp).

(Alg-M-IApp) $\bar{\alpha}; \Gamma; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow E'; \Sigma', \theta(\rho_1) \rightsquigarrow x, \theta(\Sigma); \theta$

From the hypothesis of the rule and the induction hypothesis, we have that

$$\theta(\Gamma, \rho_1 \rightsquigarrow x); \theta(\rho_2) \rightsquigarrow |\theta|(E\, x) \vdash_r \theta(\tau) \rightsquigarrow E'; \Sigma'$$

By rule (M-IApp) we may then conclude

$$\theta(\Gamma); \theta(\rho_1 \Rightarrow \rho_2) \rightsquigarrow |\theta|(E) \vdash_r \theta(\tau) \rightsquigarrow E'; \Sigma', \theta(\rho_1) \rightsquigarrow x$$

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

Then it follows from the rule's hypothesis and from the induction hypothesis that

$$\theta(\Gamma); \theta(\rho) \rightsquigarrow |\theta|(E\, \alpha) \vdash_r \theta(\tau) \rightsquigarrow E'; \Sigma'$$

Hence, it follows from rule (M-TApp) that

$$\theta(\Gamma); \theta(\forall \alpha.\rho) \rightsquigarrow |\theta|(E) \vdash_r \theta(\tau) \rightsquigarrow E'; \Sigma'$$

□

Lemma B.22. *If*

$$\bar{\alpha}; \Gamma; \rho \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow E'; \Sigma'$$

then

$$\bar{\alpha}; \rho \vdash_{coh} \tau$$

Proof. The derivation of the conclusion is obtained by erasing the irrelevant arguments from the derivation of the hypothesis. □

Lemma B.23. *If*

$$\theta = mgu_{\bar{\alpha}}(\tau, \tau')$$

then

$$\theta(\tau) = \theta(\tau')$$

and

$$dom(\theta) \subseteq \bar{\alpha}$$

Proof. Straightforward induction on the derivation. □

B.7 Completeness of the Algorithm wrt Deterministic Resolution

Lemma B.24. *If*

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

then

$$\Gamma \vdash_{alg} \rho \rightsquigarrow E$$

Proof. From the hypothesis it follows that

$$tyvars(\Gamma); \Gamma \vdash_r \rho \rightsquigarrow E$$

Hence, by Lemma B.25 and rule (Alg-R-Main) the desired conclusion follows

$$\Gamma \vdash_{alg} \rho \rightsquigarrow E$$

□

Lemma B.25. *If*

$$\bar{\alpha}; \Gamma \vdash_r \rho \rightsquigarrow E$$

then

$$\bar{\alpha}; \Gamma \vdash_{alg} \rho \rightsquigarrow E$$

Proof. The lemma follows from the isomorphism between the rule sets of the two judgements and from Lemma B.26. □

Lemma B.26. *If*

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_r \rho \rightsquigarrow E$$

then

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \rho \rightsquigarrow E$$

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

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

From the rule's first hypothesis and Lemma B.27 we have

$$\epsilon; \Gamma; \rho \rightsquigarrow x; \epsilon \vdash_{alg} \tau \rightsquigarrow E'; \bar{\rho} \rightsquigarrow \bar{x}$$

Then, using Lemma B.25 and rule (Alg-L-RuleMatch) we conclude

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

(L-RuleNoMatch) $\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_r \tau \rightsquigarrow E'$

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

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \tau \rightsquigarrow E'$$

From the rule's first hypothesis and the negation of Lemma B.28, we have:

$$\bar{\alpha}; \rho \not\forall_{coh} \tau$$

Hence with rule (Alg-L-RuleNoMatch) we conclude

$$\bar{\alpha}; \Gamma; \Gamma', \rho \rightsquigarrow x \vdash_{alg} \tau \rightsquigarrow E'$$

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

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

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \tau \rightsquigarrow E$$

By rule (Alg-L-Var) we conclude

$$\bar{\alpha}; \Gamma; \Gamma', x : \rho \vdash_{alg} \tau \rightsquigarrow E$$

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

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

$$\bar{\alpha}; \Gamma; \Gamma' \vdash_{alg} \tau \rightsquigarrow E$$

By rule (Alg-L-TyVar) we conclude

$$\bar{\alpha}; \Gamma; \Gamma', \alpha \vdash_{alg} \tau \rightsquigarrow E$$

□

Lemma B.27. *If*

$$\theta_1(\Gamma); \theta_1(\rho) \rightsquigarrow |\theta_1|(E) \vdash_r \theta_1(\tau) \rightsquigarrow |\theta_1|(E'); \theta_1(\Sigma')$$

and

$$dom(\theta_1) \subseteq \bar{\alpha}$$

then

$$\bar{\alpha}; \Gamma; \rho \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow |\theta_2|(E'); \theta_2(\Sigma', \Sigma)$$

and

$$dom(\theta_2) \subseteq \bar{\alpha}$$

and

$$\theta_1 \sqsubseteq \theta_2$$

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

(M-Simp) $\theta_1(\Gamma); \theta_1(\tau') \rightsquigarrow |\theta_1|(E) \vdash_r \theta_1(\tau) \rightsquigarrow |\theta_1|(E); \theta_1(\epsilon)$ where $\theta_1(\Gamma) = \theta_1(\tau')$.

From Lemma B.29 and rule (Alg-M-Simp) we then have

$$\bar{\alpha}; \Gamma; \tau' \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow \theta_2(E); \theta_2(\Sigma)$$

(M-IApp) $\theta_1(\Gamma); \theta_1(\rho_1 \Rightarrow \rho_2) \rightsquigarrow |\theta_1|(E) \vdash_r \theta_1(\tau) \rightsquigarrow |\theta_1|(E'); \theta_1(\Sigma', \rho_1 \rightsquigarrow x)$

From the hypothesis of the rule and the induction hypothesis, we have that

$$\bar{\alpha}; \Gamma, \rho_1 \rightsquigarrow x; \rho_2 \rightsquigarrow E\, x; \rho_1 \rightsquigarrow x, \Sigma \vdash_{alg} \tau \rightsquigarrow |\theta_2|(E'); \theta_2(\Sigma', \rho_1 \rightsquigarrow x, \Sigma)$$

By rule (Alg-M-IApp) we may then conclude

$$\bar{\alpha}; \Gamma; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow |\theta_2|(E'); \theta_2(\Sigma', \rho_1 \rightsquigarrow x, \Sigma)$$

(M-TApp) $\theta_1(\Gamma); \theta_1(\forall \alpha . \rho) \rightsquigarrow |\theta_1|(E) \vdash_r \theta_1(\tau) \rightsquigarrow |\theta_1|(E'); \theta_1(\Sigma')$

From the hypothesis of the rule and the induction hypothesis, we have that

$$\bar{\alpha}, \alpha; \Gamma; \rho \rightsquigarrow E\, \alpha; \Sigma \vdash_{alg} \tau \rightsquigarrow |\theta_2|(E'); \theta_2(\Sigma', \Sigma)$$

Hence, it follows from rule (Alg-M-TApp) that

$$\bar{\alpha}; \Gamma; \forall \alpha . \rho \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow |\theta_2|(E'); \theta_2(\Sigma', \Sigma)$$

□

Lemma B.28. *If*

$$\bar{\alpha}; \rho \vdash_{coh} \tau$$

then for all E, Γ, Σ *there exist* E', Σ' *such that*

$$\bar{\alpha}; \Gamma; \rho \rightsquigarrow E; \Sigma \vdash_{alg} \tau \rightsquigarrow E'; \Sigma'$$

Proof. The proof is straightforward induction on the derivation. The conclusion's judgement is an annotated version of the hypothesis' judgement. □

Lemma B.29. *If*

$$\theta(\tau) = \theta(\tau')$$

and

$$dom(\theta) \subseteq \bar{\alpha}$$

then

$$\theta' = mgu_{\bar{\alpha}}(\tau, \tau')$$

and

$$dom(\theta') \subseteq \bar{\alpha}$$

and

$$\theta \sqsubseteq \theta'$$