Programs for Free
Towards the Formalization of Implicit Resolution in Scala

A thesis submitted for the degree of Master of Computer Science at the Technical University Delft

Arjen Rouvoet
{a.rouvoet@student.tudelft.nl}

Supervisors

Sandro Stucki, PhD Student, EPFL
Erik Meijer, Professor, TU Delft
Abstract

Implicit resolution has been part of the Scala language for a while, but the formal system it constitutes and its properties have not been formalized. Oliveira et al. were the first to formalize a calculus of implicits $\lambda_{\Rightarrow}$ based on the ideas of implicits in Scala. We propose a stronger calculus $\lambda_{S\Rightarrow}$, extending their results and closing the gap with the maximum achievable expressiveness of resolution on top of System F. We believe that the strengthened results provide new insights into implicit resolution and are a big step towards a formalization that is entirely faithful to Scala.

We give a partial algorithm for $\lambda_{S\Rightarrow}$ and use a proof-technique proposed by Abel & Altenkirch to establish its soundness and partial completeness independent of termination. To ensure that our results are correct and that it can be extended in the future, we made the language and proofs precise in the Agda dependently typed language.
# Contents

1 The Informal Language of Implicits  
  1.1 Implicit Values & Rules ............................................. 9  
  1.2 Resolution ............................................................. 11  
    1.2.1 Decidability .................................................... 12  
  1.3 Implicits in Existing Languages .................................... 15  
    1.3.1 Haskell .......................................................... 15  
    1.3.2 Scala ............................................................ 16  
    1.3.3 Agda ............................................................. 17  
    1.3.4 Coq ............................................................... 18  

2 The Formal Language of Implicits .................................... 20  
  2.1 Syntax & Welltypedness ............................................... 20  
  2.2 Design Considerations ............................................... 24  
    2.2.1 Implicit Application .............................................. 24  
    2.2.2 Rule Types ....................................................... 25  
    2.2.3 The Underlying Typesystem ...................................... 26  
  2.3 $\lambda_{\text{imp}}$ in Agda ............................................. 26
3 Ambiguous and Deterministic Resolution

3.1 Ambiguous Resolution $\lambda^A$ .................................................. 27
   3.1.1 Undecidability of $\lambda^A$ ............................................... 28
   3.1.2 Finiteness & Termination .................................................. 33
3.2 Deterministic Resolution $\lambda^D$ .................................................. 34
   3.2.1 Deciding $\lambda^D$ .......................................................... 36
3.3 $1\frac{1}{2}$ Order Resolution .......................................................... 37
3.4 Naively Strengthening Matching .................................................. 41
   3.4.1 The Rules of $\lambda^D^+$ .................................................... 41
   3.4.2 Deciding $\lambda^D^+$ ......................................................... 43

4 $\lambda^S$: Syntax Directed Resolution .............................................. 44
   4.1 The Rules of $\lambda^S$ .......................................................... 44
   4.2 Expressiveness of $\lambda^S$ .................................................... 45
      4.2.1 Closing the Gap .......................................................... 48
   4.3 A Partial Algorithm for $\lambda^S$ ............................................. 53
      4.3.1 A Partial Matching Algorithm ......................................... 54
      4.3.2 A Partial Resolution Algorithm ....................................... 59
      4.3.3 The Source of Non-Termination ...................................... 60

5 Termination Of Resolution ............................................................ 62
   5.1 Finite Fragments of $\lambda^S$ .................................................. 62
      5.1.1 $\lambda^S^N$: Fixed Depth Recursion Stack ............................ 65
      5.1.2 Oliveira’s Termination Condition .................................... 65

3
5.1.3 Scala’s Termination Condition ........................................... 68

6 The Semantics of Resolution .................................................. 70

6.1 Translating Types and Contexts ......................................... 70
6.2 Translating Resolution Derivations ................................... 71
6.3 Translating Welltyped $\lambda \Rightarrow$ Terms ........................... 73
List of Figures

2.1 Syntax of $\lambda_{\Rightarrow}$ .................................................... 21
2.2 The head of a context type $\alpha \Rightarrow$ ........................................ 22
2.3 Typesystem of $\lambda_{\Rightarrow}$ .................................................... 23
2.4 Unambiguous types ............................................................... 23
3.1 Oliveira’s ambiguous resolution rules ........................................ 28
3.2 Translation of types and contexts between $\lambda_{\Rightarrow}$ and System F ........................................ 30
3.3 Translation of $\lambda_A$ derivations to System-F expressions ............ 31
3.4 Oliveira’s deterministic resolution rules ...................................... 35
3.5 Incompleteness of the deterministic w.r.t. the ambiguous rules ........ 39
3.6 The design space between $\lambda_D$ and $\lambda_A$ ............................... 40
3.7 $\lambda_{\Rightarrow}^{D+}$ with strengthened matching ............................... 42
4.1 The rules of $\lambda_S$ ............................................................... 45
4.2 $\eta$-long-$\beta$-normal System-F expressions [Abe08] ...................... 50
4.3 Translation of $\lambda_S$ into $\eta$-long-$\beta$-normal System-F expressions ........ 51
4.4 The function $\text{match}$ .......................................................... 55
4.5 Recursion scheme of $\text{resolve}$ ............................................... 61
5.1 Recursion scheme of $\text{resolve}^{\Phi,N}$ ........................................... 64
5.2 Oliveira’s termination condition ........................................... 66
6.2 Semantics of $\lambda^S$ resolution derivations .............................. 76
6.3 The semantics of welltyped $\lambda^S$ terms ................................. 77
Introduction

Implicit argument passing is part of many languages that boast a strong typesystem. It is an important tool for generic programming as it serves as an alternative to overloading to achieve adhoc polymorphism. In Scala, function arguments of any type can be marked *implicit*. The compiler will conduct a type-directed search for a fitting expression when such a function is called. This is in sharp contrast to for example Haskell, where implicit argument passing is limited to type class instances. The flexibility of Scala’s impuls allows them to be used to emulate type classes, but it also permits other uses, such as abstraction over boilerplate value passing.

The specifics of an implementation are always tailored to the host language, but Oliveira et al. tried to capture the essence of implicit argument passing in a small formal system based on System F. The focus of their formalization is on the accompanying resolution algorithm, its properties and a semantics for resolution. This is different from the formal specifications of e.g. type classes in Haskell or instance-arguments in Agda, which both focus less on search and more on decidability of type inference.

Oliveira et al. define two sets of resolution rules, labeled *ambiguous* and *deterministic* respectively. Although the ambiguity is deemed problematic, the former set of rules is said to “correspond to the intuition of logical implication checking” [OSC+12]. That is: implicit instances are either implicit values or the natural implications of those values. This is exactly the intuition we would like to capture, but, as we’ll show, this very general flavor of resolution is *undecidable*.

To regain decidability some of the expressiveness is conceded. The resulting *deterministic rules* are indeed decidable as also proven by the authors[OSC+12]. But it comes at a cost: deterministic resolution is considerably less powerful than for example Scala’s resolution.

Scala’s implicit resolution differs significantly from the formal system $\lambda_\Rightarrow$. Scala’s implementation is significantly more expressive and yet, the system it constitutes is still believed to be decidable. Our goal is to take a step towards a formal system that more accurately depicts Scala’s flavor of implicit resolution. The formalization will help us prove essential properties such as decidability (with termination of search as an important component). A more accurate formalization will also help to evaluate the expressiveness of resolution in Scala and help with the exploration of alternatives.
Contributions

We discuss in detail the “Calculus of Implicits” and the two flavors of resolution that appear in [OSC+12] and explore the possibilities of a stronger version of implicit resolution. We motivate this approach by proving some essential properties about the design space. We propose a stronger formalization $\lambda^S_\subseteq$ and give it a strong foundation by relating it to its competitors.

We give special attention to termination of algorithmic resolution, which we show lies at the heart of strong decidable resolution; rather than it just being an observable property of the calculus. We define a family of resolution judgments that we call finite fragments of $\lambda^S_\subseteq$ and show that it is the most expressive formulation of decidable resolution for System F.

The formal system is described in two parts: in chapter 1, the syntax and type system is presented, leaving the main resolution judgment abstract. In the other chapters we will evaluate several implementations of the resolution judgment. We finish with a semantics for our formulation of the calculus.

Notation & Conventions

In the work presented we often blur the distinction between types and values and their Curry-Howard counterparts propositions and proofs. Both interpretations are useful (and we believe necessary) to fully appreciate the subject.

Whenever we introduce notation we try to give it a clear intuitive interpretation as well; hoping that this may assist the reader to read the proofs.

All proofs are mechanized in the dependently typed language Agda. The mechanization not only serves as a sanity check, but also assists exploration and future development.
Chapter 1

The Informal Language of Implicits

In order to understand the more intricate parts of a language of implicits, we should first try to get a feeling for such a language. In this chapter we build our mental model by describing the core concepts of implicit argument passing and outlining the semantics of implicit resolution. Our mental model is based on the specification (and implementation) of implicits in Scala.

This is a crucial step because it defines what our formalization ought to behave like: our mental model will provide the frame of reference for the quality of the calculus. A perfect formalization would mean that every (reasonable) expectation based on our mental model corresponds to a theorem in the formal system. This is reflected in this report by the fact that that many of the discussions can be recognized to appear in three parts: 1) a discussion that concludes what ought to be, 2) a proposition in formal language and 3) a proof (or refutation) of that proposition.

At all times it’s important to differentiate between the mental model and the formal system. As soon as we blur this distinction, we risk impairing our ability to reason within the system, confusing theorems in our mental model with theorems in the system. Although the mental model is based on the Scala implementation of implicit argument passing, some concessions are made.

1.1 Implicit Values & Rules

Implicits are values that are passed around based on a type, rather than by name. In order to understand their usefulness we can review one of the use cases. Maybe the most important one is its role in type-class based polymorphism. Consider the following
abstraction:

```scala
// the shared property
abstract class Serializable {
    def serialize(): Array[Byte]
}

// the type specific implementation
class Int extends Serializable {
    def serialize() = ???
}

// the function that uses the abstraction
def send(object: Serializable): Unit = {
    someCall(object.serialize())
}
```

The downside of abstraction through subtyping is that we have to figure out beforehand to what abstractions a type will belong. And, in the many languages that don’t support multiple inheritance, we have to somehow linearize the inheritance chain. Another way to make the same abstraction would be to implement the property separately.

```scala
abstract class Serializer[T] {
    def serialize(obj: T): Array[Byte]
}

object IntSerializer extends Serializer[Int] {
    def serialize(obj: Int) = ???
}

def send[T](object: T, serializer: Serializer[T]) = {
    val bytes = serializer.serialize(object)
    // transmit bytes
}
```

`Serializer[T]` is usually called a *type class* and the argument `serializer` of function `send` acts as a witness of the fact that `T` has the property of being serializable (i.e. it belongs to the type class). The downside of this approach is of course that the programmer now has to pass the witness along with the object itself. The solution to which is to let the compiler find the witness for you based on the argument types. That is: pass the witness implicitly.
implicit object intSerializer extends Serializer[Int] {
  def serialize(obj: Int) = obj.bytes()
}

def send[T](object: T)(implicit serializer: Serializer[T]) = {
  someCall(serializer.serialize(object))
}

// usage
send(3)

// from which the compiler infers
send(3)(IntSerializer)

// the following fails to compile,
// because no instance of Serializer[String] is in the implicit scope
send("Hi")

This saves us the trouble of finding and passing the right instance ourselves. The real power of implicits however is only unleashed when the resolution algorithm can derive new instances by itself. This can be accomplished by special, implicitly applyable functions that take implicit arguments. To make the distinction with regular function values, we will refer to these as rules. A reasonable rule in light of our example would be one that prescribes how to serialize a list of serializable things:

implicit def listSerializer[T](implicit serializer: Serializer[T]): Serializer[List[T]] = ?

// usage
send(List(3, 4, 5))

// from which the compiler infers
send(List(3, 4, 5))(listSerializer(intSerializer))

From a compiler viewpoint adding rules to the mix of course complicates resolution significantly. Now that we have outlined the practical usage of implicits, we will move the focus to this problem of implicit resolution.

1.2 Resolution

The engine powering implicit argument passing is implicit resolution: i.e. the problem of finding an instance of a type, given a collection of implicit values and rules. The idea that a type \( a \) is implicitly resolvable ultimately means that we can build an expression \( e \) from the values and rules in the implicit context, such that \( e \) has type \( a \). This is an informal
semantics of resolution.

When we apply the Curry-Howard correspondence, we have that types are propositions, rules are implications and the values from the implicit context are facts. We can extrapolate from there and will find that the ways in which we are allowed to combine rules and values, are the inference rules of a formal system. Saying that a type \( a \) can be resolved under a context, is analogous to saying that there exists a derivation of its corresponding proposition within this formal system.

The fact that ultimately we need to derive an expression in the host language from a resolution derivation, is a constraint on the inference rules and axioms of the formal system: we will need to show that the system has a semantics with the well-typed expressions of our host language as its semantic domain. This is the meta-perspective on implicit resolution that we will take in the majority of this report. We will freely use (and even mix) terminology from the formal system, its interpretation and even its semantics whenever appropriate.

All these definitions are purposefully vague and can be instantiated to yield both powerful and trivial forms of resolution. In the following sections we narrow our focus and explore some of the fundamental properties of resolution.

From this point on we will start using notation from the formal language to describe resolution problems. The types of the formal system are the types from System F extended with an arrow type \( a \rightarrow b \), representing the type of rules; contrasting the type of lambda abstractions \( a \rightarrow b \). We'll also use the notation \( \Delta \vdash_r a \) to express the judgment that a goal-type \( a \) can be resolved under the implicit context \( \Delta \), where \( \Delta \) is a list of types, representing the implicit values and rules in scope. The type \( a \) in this judgement is referred to as the “resolution goal” or the “query”. The syntax of types and contexts is summarized in fig. 2.1.

It is useful to realize that while the usual interpretation of \( a \rightarrow b \) is “implication”, in implicit resolution this interpretation is reserved for \( a \Rightarrow b \). The type \( a \rightarrow b \) is left uninterpreted as an higher order proposition. In other words: rules can be applied during resolution to derive new values, while lambdas can only be passed around, because they have no meaning attached to them.

### 1.2.1 Decidability

The decidability of a resolution problem depends on the queries that we expect it to resolve; or more precise: it depends on the inference rules that we allow. In this section we consider two versions of resolution: second order and first order resolution respectively. We interpret \( \Delta \vdash_r a \) in the most general sense: “Assuming \( b \) for all \( b \in \Delta \), does \( a \) follow logically?”. Where all types should be read as propositions.
Decidability of Second Order Resolution

System F is often referred to as the second order lambda calculus $F_2$, because it allows quantification over proper types (of kind $\ast$), but not over type constructors of higher kinds [Pie02, 30.4]. Under Curry-Howard, this corresponds to the fragment of second order propositional logic without existential quantification. The following proposition thus has an equivalent in System F:

$$\forall a. a \to a$$

But the equivalent of a proposition which quantifies over a predicate $f$, corresponding to a type of kind $\ast \to \ast$, is precluded:

$$\forall f. \forall a. \forall b. (a \to b) \to f a \to f b$$

We say resolution is of second order if the full expressiveness of System F’s polymorphism is available for reasoning. In other words, a proof of $\Delta \vdash r a$ can be any System F expression using $\Delta$ as its context that has type $a$. We can prove that second order resolutions is undecidable by relating it to the type inhabitation problem $\Gamma \vdash ? : t$ that reads: given a System F type $t$, is there a term $x$ such that $\Gamma \vdash x : t$? A sketch of the proof is presented below while the formal proof is included in section 3.1 when we have formalized second order resolution.

1.2.1 Theorem: 2nd order resolution is undecidable

Proof sketch. We prove the theorem by constructing a resolution problem “does $\Delta$ resolve $a$?”, or $\Delta \vdash_r? a$, for all type-inhabitation problems $\Gamma \vdash ? : a$, where $\Delta$ is a list of the types of all the bindings in $\Gamma$. Function-types are translated to rule-types everywhere in order to retain their Curry-Howard interpretation. Clearly $\Delta \vdash_r? r$ and $\Gamma \vdash ? : t$ pose the same question under the proposed semantics of resolution and are therefore equivalent.

The undecidability of second order resolution now follows from the undecidability of the type inhabitation problem for System F [BDS13, 73].

By Theorem 1.2.1, second order resolution serves as an upper bound on the expressiveness of a formalization of decidable implicit resolution.

Decidability of First Order Resolution

We define first order resolution analogously to second order resolution, restricting ourselves to the first order subset of System F (i.e. the simply typed lambda-calculus). Concretely this
means that a proof of $\Delta \vdash \gamma$, $\alpha$ must correspond to a STLC expression that is well-typed under a typing context that contains bindings for every type in $\Delta$. In contrast to second order resolution, this fragment is decidable.

1.2.2 **Theorem:** First order resolution is decidable

*Proof sketch.* Similarly we can construct an equivalent first-order type-inhabitation problem $\Gamma \vdash \gamma \in a$ for every first-order resolution problem $\Delta \vdash \gamma \rightarrow a$, where $\Gamma$ is a typing context consisting of bindings $x : b$ for every type $b \in \Delta$.

The conclusion follows from the decidability of the type-inhabitation problem for the simply typed $\lambda$-calculus [BDS13, 73]. □

First order resolution is a lower bound on the achievable expressiveness of a decidable formalization of resolution.

**Decidability of Alternatives**

Theorems 1.2.1 and 1.2.2 act as an upper and lower bound respectively on the achievable expressiveness of decidable resolution formalizations. The fact that second order resolution is undecidable, does not mean that we should settle for first order resolution. Clearly the restriction to a monomorphic language would be a huge setback as the example from section 1.1 demonstrated the usefulness of polymorphic rules. Instead we consider alternative fragments of System F that contain polymorphic types, but have a decidable type inhabitation problem.

One specific problem that makes the polymorphic case more complicated is that detecting diverging resolution paths becomes nontrivial. To understand this, consider the following example:

```scala
// a sorting method using the X type class
def sort[T](xs: List[T])(implicit ord: Ordering[T]) = ???

// one rule to rule them all
implicit def magic[T](implicit ord: Ordering[List[T]]): Ordering[T] = ???

sort(List(1, 2, 3))
```

// ... expands to
```
sort(List(1, 2, 3))(magic(magic(magic(...))))
```

A naive search algorithm would loop indefinitely trying to complete the expansion. The stack of resolution goals would grow indefinitely:

1. Ordering[Int]
2. Ordering[List[Int]]

3. Ordering[List[List[Int]]]

4. etc.

Although in this case it’s clear that the expansion is infinite, this is not always as obvious. If we were to add the following value to the implicit context of above’s example, a finite expansion using magic suddenly does exists:

```scala
implicit def endOfTheLine: Ordering[List[List[Int]]] = ???

sort(List(1, 2, 3))
// ...expands to
sort(List(1, 2, 3))(magic(magic(magic(endOfTheLine))))
```

In the monomorphic case such infinite regress, where each goal on the resolution stack is unique for the resolution stack, cannot occur. Because there are only a finite number of rules and every rule matches only a single monomorphic goal-type, every non-terminating expansion must repeat one of the previous goals in the resolution stack within a finite number of steps. This can always be detected, whereupon the resolution algorithm can backtrack.

It follows that if we want both polymorphism and decidability of resolution, we have to impose some other restriction on resolvable expressions. Oliveira et al. choose a particular restriction and prove that the resulting calculus has decidable type-checking. In contrast, we pursue a more general approach that abstracts over such restrictions.

### 1.3 Implicits in Existing Languages

Scala wasn’t the first language to incorporate some form of implicit argument passing and it also wasn’t the last. Haskell has an implementation of implicits in the form of support for type classes. Other languages have since adopted some of Scala’s ideas[DP11][WBY15]. In the following sections we summarize and compare implementations in four languages.

#### 1.3.1 Haskell

Haskell has an implementation of implicit argument passing in the form of type classes. Type classes are a special structure and are not a first-class citizen of Haskell; i.e. they cannot be abstracted over in the usual way.
Decidability of instance resolution played an important role in the Haskell implementation; type class and instance declarations are constrained in such a way that type inference and resolution are decidable.

In Haskell 98 these restrictions are quite conservative and only allows types of a very specific form \( C \ a \) to appear in class contexts. The GHC extension FlexibleInstances lifts this restriction and allows arbitrary types in class contexts, as longs as they fulfill the so called Paterson and Coverage conditions [Has]. Together these conditions ensure that type inference and instance resolution are decidable.

Even more can be accomplished using the extensions MultiParameterTypeClasses and FunctionalDependencies. The latter can be used to control type inference during instance resolution [Jon00], particularly to prevent ambiguity when using multi-parameter type classes. The functional dependencies extension has been formalized and its resolution problem has been proven decidable [SDPJS07].

### 1.3.2 Scala

We’ve sketched the main ingredients of implicits in Scala: implicit values, implicit functions (or rules) and implicit application. Implicit application is the trigger of *implicit resolution*. The inference rule for implicit application is very simple: every expression that takes an implicit argument is implicitly applied immediately. If a rule should be passed as a regular function, the programmer needs to make this explicit.

Implicit argument passing should not be confused with another Scala feature called implicit conversions, where a function application is inferred, rather than a function argument.

A limitation of Scala’s implicits is that implicit rules are not represented by their own type. The annotation that a parameter is implicit is syntactical. This limitation simplifies both resolution and inference of implicit application but also has its drawbacks:

- Implicit rule types do not compose
- Functions and rules cannot return rules
- Functions and rules cannot take rules as parameters

All of the above have practical applications however. Imagine a rule type:

```scala
implicit T => S
```

Where the annotation `implicit` belongs to the rule argument `T`. The following usage of such a type was shown in a Scala Days 2015 presentation [Ode15]:
By type expansion, `readInt` requires an implicit *capability* that it may throw an exception, thus tracking this possible effect when it’s called. This idea of tracking effects using implicit capabilities stems from the observation that both effects and implicits are additive and propagate along the call-graph [Ode15].

Another application of first-class rules is the construction of higher order rules. The following curry-rule for example, takes rules that take a single tupled argument and returns a rule that accepts two arguments:

```scala
implicit def curry[T,S,U](implicit f: (implicit (T, S) => U)): (implicit T, S) => U
```

Whether or not this is a good idea from a practical standpoint is entirely up for debate.

The main ideas of implicit resolution in Scala are captured in the official Scala specification [Sca]. The specification is imprecise with respect to many of the details and more conservative than the implementation with regards to for example the constraints that ensure termination.

Where Haskell only permits class declarations whose type obeys the necessary constraints to always guarantee termination, Scala has adopted an approach where every declaration is permitted and termination of resolution is enforced simply by detecting possible divergence. Effectively this effectively also constrains the rules that can be used to derive instances but comparatively it allows some flexibility: rules that may result in divergence in some cases can still be applied in safe cases.

### 1.3.3 Agda

The language Agda (which we use to mechanize our proofs) has its own implementation of implicit argument passing: *instance arguments*. They are introduced in the paper “On the Bright Side of Type Classes: Instance Arguments in Agda” [DP11]. The authors of said paper explicitly do not permit rules, such that resolution reduces to searching the scope for values of the required type. The Agda implementation *does* permit rules if they are marked explicitly using the keyword `instance`.

Instance arguments are inspired by Scala’s implicits and do not require a separate structure to encode type classes, like Haskell does require. In contrast to Scala’s implicits, rules in Agda are first-class citizens and have their own function space. In contrast to Haskell, instances are not global but scoped by Agda’s module system. Something that is unique about Agda’s instance arguments compared to Scala’s or Haskell implicits is that rules can be fully dependently typed.
Agda’s instance argument resolution is not immune to termination problems. In fact it doesn’t guarantee termination at all and can easily be tricked into a loop:

```agda
open import Data.Bool
open import Data.List
instance
  loop : ∀ {a} {A : Set a} {{list : List A}} ↦ List A
  loop {{list}} = inst

test : List Bool
test = loop
```

In this example the function `loop` is available as a rule during instance resolution. It has one instance argument `inst : List A`.

Readers familiar with Agda should not confuse its implicit arguments with the implicit arguments which are the subject of this thesis. What Agda refers to as implicit arguments are arguments which can be inferred by type inference. Because Agda is a dependently typed language, these inferred arguments can be objects as well as types. Consider the following example from the mechanized proofs in this report:

```agda
wk : ∀ {ν} {Δ Δ′ : ICtx ν} {a r} ↦ Δ ⊢ r ↓ a ↦ Δ ⊆ Δ′ ↦ Δ′ ⊢ r ↓ a
wk p sub = ...
```

Which is called with only two parameters:

```agda
wk p sub
```

In the above type of `weakening`, the parameters surrounded with braces are implicit. They can be inferred from the passed explicit parameters because they appear as part of their types `Δ ⊢ r ↓ a` and `Δ ⊆ Δ′`. They differ crucially from our notion of implicit arguments in the sense that no additional instances are passed to the function, based on the context at the call site; these implicit arguments are just names for instances that are part of the types of explicitly passed arguments.

### 1.3.4 Coq

Coq is perhaps the most esoteric language in this shortlist of languages. We have included it here, because implicit resolution can be interpreted using the Curry Howard correspondence as automatic proof search. The language Coq is a proof assistant that supports this using proof search tactics. At any point in a proof, one can unleash such a tactic on the
set of open assumptions to solve some of the open goals. Many tactics exist for particular
type of goals, but one that is of particular interest is auto.

The auto tactic is described as “naively trying to apply lemmas and assumptions in all
possible ways” [Use]. This agrees with the intuition that we have of implicit resolution
(although it’s not the most flattering description).

It can apply dependently typed, polymorphic functions, as long as the instantiation can
be deduced from the goal [Use]. This latter restriction does not apply to the similar tactic
eauto, which can apply such functions by introducing an existential type variable. In
programming languages such existential implicit application is usually unwanted, because
it leaves runtime behavior ambiguous.

In any case termination is guaranteed simply by enforcing a maximum recursion depth.
Because one explicitly invokes the tactic in Coq, and the maximum depth is a parameter,
this is a reasonable choice. In Scala, where implicit resolution is triggered implicitly, such
an approach would either enforce an arbitrary maximum depth or be rather clumsy in its
use.
Chapter 2

The Formal Language of Implicits

In the paper “The Implicit Calculus: A new Foundation for Generic Programming”, Oliveira et al. presented a syntax and typesystem for a language of implicits that they call $\lambda_{\Rightarrow}$. They also presented two implementations of the main resolution judgment $\Delta \vdash_r a$: ambiguous and deterministic resolution.

In this section we present a slightly modified syntax and typesystem that we will also refer to as $\lambda_{\Rightarrow}$. For the moment we will leave the main resolution judgment $\Delta \vdash_r a$ abstract. In subsequent chapters we will focus on different formalizations of resolution. We distinguish the different languages that result from different flavors of resolution using superscripts; e.g. $\lambda^D$ is the language where $\Delta \vdash_r a$ is implemented by the rules of deterministic resolution as formalized by Oliveira et al.

Syntactically the language of implicits is an extension of System F. The differences with Oliveira’s language of implicits [OSC+12] are only slight and are discussed in section 2.2.

2.1 Syntax & Welltypedness

A complete overview of the syntax is presented in Figure 2.1. Like Oliveira et al. we partition the System F types into simple-types $\tau$ and context-types $a$. This partitioning is based on the observation that simple types are the base cases of type-directed resolution (section 3.2) [OSC+12]. The System F types are extended with the type of rules $a \Rightarrow b$, where we refer to the domain of a rule as the rule context. Context types $a$ have a “head”, which is a simple type, defined by $a \triangleleft$ (Figure 2.2).

The System F terms are extended with rule-abstraction $\rho (x : a). e$ (analogous to the usual $\lambda$ for term-abstraction), explicit rule application $r \langle e \rangle$ and implicit rule application $r \langle \rangle$. Expressions are presented here with named arguments; $\alpha$-renaming is assumed to be used appropriately to avoid capturing variables in substitutions.
We use ML-style let-bindings as syntactic sugar for the combination of function abstraction and application in the following way:

\[
\begin{align*}
\text{let } e : a \text{ in } f &:= (\lambda(x:a).f) \cdot e \\
\text{implicit } e : a \text{ in } r &:= (\rho(x:a).r)(e)
\end{align*}
\]

\[
\begin{array}{c|c}
\hline
\text{Types} & \tau, a, b, c \\
\tau := & \alpha \\
& a \rightarrow b \\
\hline
a, b, c := & \tau \\
& a \Rightarrow b \\
& \forall \alpha, \rho \\
\end{array}
\]

\[
\begin{array}{c|c}
\hline
\text{Expressions} & e, f, r \\
e, f, r := & x \\
& e[a] \\
& \lambda(x:a).e \\
& \rho(x:a).e \\
& f \cdot e \\
& r(e) \\
& \Lambda\alpha.e \\
& r(\epsilon) \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
\hline
\text{Contexts} & \Gamma, \Delta \\
\Gamma := & \emptyset \\
& \Gamma; (\alpha : *) \\
& \Gamma; (x : a) \\
\Delta := & \emptyset \\
& \Delta; (\alpha : *) \\
& \Delta; a \\
\hline
\end{array}
\]

Figure 2.1: Syntax of \(\lambda_\varnothing\)

The typing judgment is parameterized with the usual (explicit) typing context \(\Gamma\) and the implicit context \(\Delta\). The former is a list of typings \(x : a\) whereas the latter is just a list of the types of values that are implicitly available. Both context also contain records of all open type-variables \(\alpha : \ast\). This duplication is useful, because it allows us to formulate the resolution judgment independent of \(\Gamma\). Leaving the implicit resolution judgement \(\Delta \vdash_r a\) abstract for the moment the typesystem is summarized in Figure 2.3.

Because the syntax of \(\lambda_\varnothing\) is strictly an extension of System F, the usual System F typing
\[
\begin{align*}
\tau \triangleleft & = \tau \\
(a \Rightarrow b) \triangleleft & = b \triangleleft \\
(\forall \beta . a) \triangleleft & = a \triangleleft
\end{align*}
\]

Figure 2.2: The head of a context type \( a \triangleleft \)

...rules are unaltered. The typing rule for rule-abstraction \texttt{R-ABS} is almost identical to the typing rule for term-abstraction, except that the variable introduced by the parameter is also added to the implicit context. The most significant addition to the system is the typing rule for rule application, which depends on a hypothesis \( \Delta \vdash_r a \) witnessing the fact that the type \( a \) can be derived from the implicit context \( \Delta \).

Additionally, rule-contexts are constrained in \texttt{TY-RABS} to types \( a \) that satisfy the predicate \( \emptyset \vdash_{\text{UNAMB}} a \). This predicate ensures that we can uniquely determine instantiations of all polymorphic values in the implicit context that are used during resolution. We will elaborate on the necessity of this predicate in section 3.2.1.
Figure 2.3: Typesystem of $\lambda_\Rightarrow$

Figure 2.4: Unambiguous types
2.2 Design Considerations

While the syntax and typesystem are rather straightforward, a critical reader might notice that there are some slight differences with the formulation by Oliveira et al. These differences accentuate our effort to make the calculus resemble Scala’s implementation more closely. We would like to elaborate on these choices here, and also explain the concessions that were made in the choice of the base typesystem.

2.2.1 Implicit Application

Oliveira’s calculus defines a query operator \( ?a \), which functions as the trigger of implicit resolution. We choose a form that is both close to Scala’s implementation and also follows the pattern set by type and term application: *implicit application*. If \( r \) is an expression that has the type \( a \Rightarrow b \), then \( r \langle \rangle \) is the implicit application of rule \( r \) and has type \( b \) if and only if \( a \) can be implicitly resolved under \( \Delta \).

Although this syntactical form does not exist directly in Scala, implicit application is the main mechanism to trigger implicit resolution. Instead of an explicit syntactical form, the Scala compiler infers when it should implicitly apply a rule. Since the proposed calculus does not concern itself with inference, the application of rules was made explicit.

The syntax for implicit application supersedes the query operator, since we can define a function \( \text{implicitly} \) that fulfills its role using implicit abstraction and implicit application. Interestingly, the reverse is not possible.

\[
\text{let } \text{implicitly} : \forall \alpha.(\alpha \Rightarrow \alpha) = \Lambda \alpha. (\rho (x : \alpha).x)
\]

When \( \text{implicitly} \) is applied implicitly with the type argument \( \text{Int} \) like so:

\[
\text{implicitly}[\text{Int}] \langle \rangle
\]

it derives an expression of type \( \text{Int} \) from the implicit context (if it can be resolved) and returns it. The same function is part of Scala’s prelude and is defined identically:

\[
@\text{inline def implicitly}[T](\text{implicit e: T}) = e // for summoning implicit values from the nether world
\]

Perhaps it’s illustrative to show how \( \text{implicitly} \) is typed:
It almost seems like we can improve upon our definition of *implicitly* by moving the implicit application into the body of the type-abstraction:

\[
\text{let } \text{implicitly}' : \forall \alpha. \alpha = \Lambda \alpha. ((\rho (x : \alpha) . x) \langle \rangle )
\]

Note how the type simplified, such that we would be able to call it like so:

\[
\text{let } n : \text{Int} = \text{implicitly}' [\text{Int}]
\]

The body of *implicitly'* isn’t typable however, because we’d need a proof that for all \( \Delta \), \( \Delta \vdash_{\rho} \alpha \) would hold. This would of course make resolution and in turn the whole calculus trivial; as was also evident from the type of *implicitly*, whose interpretation would amount to “All theorems hold”.

### 2.2.2 Rule Types

With the inclusion of a special rule-type, we stray a bit from the current Scala implementation. We chose to include this type not only because it was also already included in Oliveira’s calculus of implicits, but also because such a type is already being considered for future versions of Scala[Ode15].

In the previous section we mentioned that in contrast to Scala we use explicit syntax for the implicit application of rules. It should be noted that inferring rule applications may be significantly more difficult in a calculus where, rules are first class citizens and may for example be returned from a function without being applied first.
2.2.3 The Underlying Typesystem

Like Oliveira et al we choose to adopt System F as our base typesystem. It might seem that if one aspires to be have a formalization of implicits that is faithful to Scala, one should use Scala’s typesystem. Scala’s typesystem is not yet entirely formalized however, although efforts are being made[AMO12]. Moreover, even while limited to System F, many interested challenges present itself that are mostly orthogonal to possible extensions of the base typesystem.

2.3 $\lambda \Rightarrow$ in Agda

In the Agda proofs we represent $\lambda$-terms using the nameless De Bruijn-syntax [ACCL91]. Not only does this ensure the well-formedness of both term- and type-variables with respect to the context, but it also ensures that expressions are invariant under $\alpha$-conversion, sidestepping the trouble of $\alpha$-equivalence.

Using De Bruijn-indices also means that both the usual typing context $\Gamma$ and implicit context $\Delta$ look slightly different. In the Agda formalization they are homogeneous lists of bindings and types respectively; neither of them “contain” type-variables. Instead they are both parameterized over the number of open type variables.
Chapter 3

Ambiguous and Deterministic Resolution

Oliveira et al. formalize two flavors of resolution: ambiguous and deterministic resolution. The former is a very expressive but ambiguous formalization of resolution, while the latter is deterministic, but considerably weaker. Both calculi are more expressive than first-order resolution and permit instantiation of polymorphic rules.

In the following section we will explore both flavors thoroughly and we will see that neither of them suffices as a formalization of implicits that is faithful to Scala.

3.1 Ambiguous Resolution $\lambda_{\geq}^A$

The rules of ambiguous resolution are described by Oliveira as “corresponding to the intuition of logical implication” and are repeated in 3.1. The label “ambiguous” is used to indicate two problems with the rules:

- The rules are not syntax-directed,
- The rules allow multiple, equally valid resolution derivations of a type

The former is caused by rules with overlapping conclusions; the latter is caused by the rule $R-IVar$, whose hypothesis choses a suitable value from the implicit context non-deterministically.

Ambiguous resolution should be seen as the formalization of implicit resolution for System F in the broadest sense. It seems, in fact, that every typing rule of System F has a
corresponding implicit derivation rule in $\lambda^A_{\Rightarrow}$. This leads us to believe that every well-typed System F expression has a corresponding derivation in $\lambda^A_{\Rightarrow}$, which would make ambiguous resolution a formalization of second order resolution. In that case we can make Oliveira’s observation that “a deterministic resolution algorithm is non-obvious” a little bit more precise: $\lambda^A_{\Rightarrow}$ is undecidable.

### 3.1.1 Undecidability of $\lambda^A_{\Rightarrow}$

The proof of the undecidability of $\lambda^A_{\Rightarrow}$ follows the technique sketched in section 1.2.1. We first describe a translation back and forth between the types and contexts from the language of implicits and System F types and contexts. We proceed by showing that $\lambda^A_{\Rightarrow}$’s resolution derivations are equivalent to well-typed System F expressions\(^1\). Finally we obtain the main theorem from the fact that the type inhabitation problem is undecidable.

We use the symbol $\lfloor \_ \rfloor$ to denote “lowering” from the calculus into System F; its counterpart $\lceil \_ \rceil$ denotes “raising” System F representations into the calculus. The translation of types is straightforward. We add an uninterpreted arrow type $a \rightarrow b$ to System F to translate lambda- and rule-types to different types, such that the translation is reversible. Types are translated to and from System F by similar functions $\lfloor \_ \rfloor_{tp}$ and $\lceil \_ \rceil_{tp}$ respectively. Implicit contexts $\Delta$ are translated to normal typing contexts $\Gamma$ and vice versa by the pair of functions $\lfloor \_ \rfloor_{ctx}$ and $\lceil \_ \rceil_{ctx}$.

When translating $\Delta$ down to some typing context $\Gamma$, we have to invent names for the bindings in $\Gamma$. There are many ways to implement this; we choose one that allows us

\(^1\) The translation differs from the semantics of the language of implicits under ambiguous resolution in that it really only translates resolution derivations; detached from the regular typing context $\Gamma$ and expressions in which these derivations are normally embedded. Also, special care is taken to ensure that the translation is invertible, because we want to prove equivalence.
to formulate the translations without having to pass around a mapping. Instead we choose an injective mapping $M : \mathbb{N} \to L$ in advance, where $L$ is your favorite countably infinite alphabet. We then use the index of the elements in $\Delta$ to find their unique name $M_{\text{index}}(a, \Delta)$. The index is counted from the tail to the head, such that the index of elements in $\Delta$ does not change when $\Delta$ is extended. That is, the function $\text{index}$ can be defined as (let $|\Delta|$ denote the number of types in $\Delta$):

\[
\begin{align*}
\text{index } a (\Delta; a) &= |\Delta| \\
\text{index } a (\Delta; b) &= \text{index } \Delta \\
\text{index } a (\Delta; (\alpha : *)) &= \text{index } \Delta
\end{align*}
\]

The left- and right-compositions of both these lowering/raising function-pairs reduce to the identity-functions, such that they establish an isomorphism between types and contexts in these two formal languages.

The translation into System F is presented in Figure 3.3 as a recursive function $\lfloor _ \rfloor_A$ that takes $\lambda^A$ derivations down to System F terms. We hope that the functional presentation helps to give the reader an intuition for the translation that is being made. The translations from welltyped System F terms to $\lambda^A$ are too involved to be presented functionally and are presented as an induction proof\footnote{The mechanized version of this equivalence proof is of course a function; The difference between the proofs here and there is that in the mechanized version we have a well-defined metalanguage to manipulate proof objects, whereas we lack that here. Moreover, we expect that most readers will be more familiar with reading inductive proofs.}.

We do not repeat the inductive definition of System F’s main typing judgment $\Gamma \vdash e \in a$ separately, because their similarity to the typing rules of the calculus. When we perform induction on said judgment, we include them there.
| \[\alpha\]_{tp} | = | \alpha | | \[\alpha\]_{tp} | = | \alpha | |
|------------------|---------|------------------|---------|
| \[a \rightarrow b\]_{tp} | = | \[a\]_{tp} \rightarrow \[b\]_{tp} | \[a \rightarrow b\]_{tp} | = | \[a\]_{tp} \rightarrow \[b\]_{tp} |
| \[a \Rightarrow b\]_{tp} | = | \[a\]_{tp} \rightarrow \[b\]_{tp} | \[a \rightarrow b\]_{tp} | = | \[a\]_{tp} \Rightarrow \[b\]_{tp} |
| \[\forall \beta. a\]_{tp} | = | \forall \beta. \[a\]_{tp} | \[\forall \beta. a\]_{tp} | = | \forall \beta. \[a\]_{tp} |

*Figure 3.2: Translation of types and contexts between \(\lambda\Rightarrow\) and System F*
Figure 3.3: Translation of $\lambda^A_\Delta$ derivations to System-F expressions
We can prove that for all \( p : \Delta \vdash a \), the expression \( \lceil p \rceil_A \) is welltyped and has type \( \lfloor a \rfloor_{tp} \).

The proof of this lemma is omitted here for brevity, but follows a similar strategy as the proof of Lemma 3.1.2.

3.1.1 **Lemma [\( \lambda^A \rightarrow \text{System F} \)]**: \( p : \Delta \vdash^A a \) implies \( \lfloor \Delta \rfloor_{ctx} \vdash \lfloor p \rfloor_A \in \lfloor a \rfloor_{tp} \).

In the other direction we have the following lemma:

3.1.2 **Lemma [Welltyped System F \( \rightarrow \lambda^A \)]**: \( \Gamma \vdash e \in a \) implies \( \lceil \Gamma \rceil_{ctx} \vdash \lceil a \rceil_{tp} \).

**Proof.** By induction on the structure of \( \Gamma \vdash e \in a \):

**Case:** \( h_0 \): \( (x : a) \in \Gamma \)

\[
\frac{\Gamma \vdash x \in a}{(F-VAR)}
\]

We observe that \( h_0 \) implies \( a \in \lceil \Gamma \rceil_{ctx} \), such that the conclusion follows immediately from R-VAR.

**Case:** \( h_0 \): \( \Gamma ; (x : a) \vdash e \in b \)

\[
\frac{\Gamma \vdash \lambda(x : a).e \in a \rightarrow b}{(F-ABS)}
\]

Applying the induction hypothesis to \( h_0 \) we derive \( \lceil \Gamma \rceil_{ctx} \vdash \lceil a \rceil_{tp} \Rightarrow \lceil b \rceil_{tp} \), from which we conclude \( \lceil \Gamma \rceil_{ctx} \vdash^A \lceil a \rceil_{tp} \Rightarrow \lceil b \rceil_{tp} \) by R-ABS.

**Case:** \( h_0 \): \( \alpha \notin \Gamma \)

\[
\frac{\Gamma \vdash \Lambda\alpha.e \in \forall\alpha.b}{(F-TABS)}
\]

Similarly using R-TABS.

**Case:** \( h_0 \): \( \Gamma \vdash f \in a \rightarrow b \)

\[
\frac{h_0 ; \Gamma \vdash e \in a}{(F-APP)}
\]

Similarly using R-APP.

**Case:** \( h_0 \): \( \Gamma \vdash e \in \forall\alpha.b \)

\[
\frac{\Gamma \vdash e[c] \in b \{\alpha \mapsto c\}}{(F-TAPP)}
\]

Similarly using R-TAPP.

From the back and forth translations of Lemma 3.1.2 and Lemma 3.1.1 and the observation that the left- and right-compositions of the type and context translations reduce to identity, we have the following theorem, consisting of two parts:

3.1.3 **Theorem [Welltyped System F \( \leftrightarrow \lambda^A \)]**: Welltyped System F expressions are equivalent to \( \lambda^A \) derivations, or:

32
\[ \Delta \vdash^A a \iff \exists e. \left( \Delta \mid_{ctx} \vdash e \in [a]_{tp} \right) \]

and:

\[ \Gamma \vdash e \in a \iff \left[ \Gamma \right]_{ctx} \vdash^A [a]_{tp} \]

Using the equivalence we can complete the proof that \( \lambda^A \) is undecidable; we proceed along the lines that we sketched in section 1.2.

### 3.1.4 Theorem: \( \lambda^A \) is undecidable.

*Proof.* We can easily show that we can build a decider for System F’s inhabitation problem from a decider for resolution according to the rules of \( \lambda^A \). It is known that the former problem is undecidable [BDS13, page 73], such that the undecidability of the latter follows by contradiction.

A type inhabitation problem instance \( \Gamma \vdash ? \in a \) can be read as: does there exist an expression \( e \) such that \( \Gamma \vdash e \in a \). Theorem 3.1.3 tells us that if such an expression exists, then a derivation of \( \left[ \Gamma \right]_{ctx} \vdash^A [a]_{tp} \) exists. Assuming a decider for ambiguous resolution we can find that derivation and using the equivalence we get the well-typed expression \( e \).

The conclusion follows as described. \( \square \)

### 3.1.2 Finiteness & Termination

In section 1.2.1 we discussed how a naive resolution algorithm could loop trying to recursively resolve a goal-type that doesn’t reduce in size. We chose an inductive definition of ambiguous resolution, such that all derivations are finite by construction, but it’s important to realize that the finiteness of a declarative resolution judgment, does not necessarily guarantee a finite search-space for a resolution algorithm.

In particular, looking at the derivation rule that deals with rule application R-IAPP, we note that the hypothesis \( \Delta \vdash a \) does not enforce \( a \) to be “smaller” than \( b \):

\[
\begin{array}{c}
h_1: \Delta \vdash a \\
h_2: \Delta \vdash a \Rightarrow b \\
\hline
\Delta \vdash b
\end{array} \quad \text{(R-IAPP)}
\]

This means that we cannot read this rule as a *reduction step* to reduce a goal of type \( b \) to a goal of type \( a \) using a rule \( a \Rightarrow b \), because if we would start doing so, we might never stop.
In other words: if we were to construct a recursive algorithm from the resolution rules, recursive resolution of \( a \) would not necessarily be a smaller problem than the resolution of \( b \).

Solving this discrepancy between the inherit finiteness of a declarative calculus and the possible infinite regress of the associated algorithm is an ongoing theme in this report.

### 3.2 Deterministic Resolution \( \lambda^D \)

The deterministic resolution rules of \( \lambda^D \) (fig 3.4) are constructed specifically to ensure that they are syntax-directed and deterministic, thus eliminating the two main issues of \( \lambda^A \).

To accomplish this, resolution is split into three phases:

1. **Simplification** of the queried type to a simple type \( \tau \),
2. **Weak matching** of the types \( a \in \Delta \) against \( \tau : a \bowtie \tau \)
3. **Strong matching** of the first weak match \( a \) against \( \tau : \Delta \vdash a \downarrow \tau \)

Simplification is represented by the rules \( \text{r-tabs} \) and \( \text{r-iabs} \) (figure 3.4). These rules simplify the query until a simple type remains. If the query was a type-abstraction or a rule-type, than the simplification step leads to additional assumptions to be captured in the implicit context \( \Delta \).

The resulting simplified query \( \tau \) is then weakly matched against the context; a step represented in the rule \( \text{r-simp} \). The idea of a “first weak match” is implemented by the relation \( \Delta \langle \tau \rangle = a \), which reads: \( a \) is the first value in \( \Delta \) that weakly matches \( \tau \).

The definition of weak matching of a context type \( a \) with a simple type \( \tau \) is captured in the relation \( a \bowtie \tau \), which simply reads: \( a \) weakly matches \( \tau \). The context type \( a \) is simplified in the same syntax-directed manner as the query, but here the semantics of abstraction are inverted: type-abstractions \( \forall \beta. a \) make a query more general, such that less implicit values will match it. This is in contrast to type-abstraction in an implicit value, which makes it match more queries. Similarly, rule abstraction \( a \Rightarrow b \) in a query gives us an additional assumption \( a \) to work with, while rule abstraction \( c \Rightarrow d \) in the implicit context give us an extra proof obligation \( c \).

As can be seen from the rule \( \text{m-iabs} \), a rule type \( a \Rightarrow b \) weakly matches a simple type \( \tau \) if \( \tau \) weakly matches the return type \( b \). We know however that a rule can only really result in its declared return type if its context can be derived. The fact that this condition is not a hypothesis of \( \text{m-iabs} \) is what makes weak matching weak.

This deficiency is corrected in the stronger matching relation \( \Delta \vdash r \downarrow \tau \), which verifies
whether the first weakly matched implicit value can really be applied to return the queried type.

We will further investigate the odd separation of weak and strong matching in section 3.3.

Figure 3.4: Oliveira’s deterministic resolution rules
3.2.1 Deciding $\lambda^D_{\Rightarrow}$

Because the rules of deterministic resolution are type-directed, the recursive algorithm presented by Oliveira et al. follows the declarative rules very naturally. The three phases of resolution are again clearly distinguishable. The only notable difference between the rules and the algorithm is due to the derivation rule for instantiating polymorphic implicit values I-TABS and its counterpart M-TABS:

$$h_1: \Delta \vdash a\{\beta \mapsto b\} \downarrow \tau\quad \text{(I-TABS)}$$

The type $b$ that is substituted into the body of the abstraction appears out of thin air in this rule. Finding these substitutions in the algorithm is delayed until the head $\tau'$ of the type $a$ is reached. Until then the type variables introduced by type-abstractions are accumulated in a parameter $\vec{\alpha}$ and matching continues with the body. Once the matching algorithm reaches the head of the implicit value, the appropriate instantiations $\vec{\phi}$ follow, because it needs to satisfy the equation $\tau'\{\vec{\alpha} \mapsto \vec{\phi}\} \equiv \tau$. At most one substitution $\vec{\phi}$ satisfies this relation because the rules in the context must satisfy the $\vdash\text{unamb}$ predicate; ensuring that all variables in $\alpha$ appear in the head $\tau'$.

Consider for example the following polymorphic rule to derive lexical orderings from the introduction:

$$r_1 := \forall \alpha.\ Ordering\ \alpha \Rightarrow\ Ordering\ [\alpha]$$

When matching the goal $Ordering\ [Int]$ against this rule, the algorithm cannot immediately guess a proper instantiation for $\alpha$. Instead it continues matching against the head of the rule $Ordering\ [\alpha]$. The substitution $\vec{\phi} := \{\alpha \mapsto Int\}$ now follows from unifying this with the goal-type. Now compare this with the following example, which does not satisfy the $\vdash\text{unamb}$ predicate:

$$r_2 := \forall \alpha.\ Ordering\ \alpha \Rightarrow\ Ordering\ [Int]$$

While it seems that matching the same goal $Ordering\ [Int]$ against $r_2$ should succeed, doing so does not lead to a substitution for $\alpha$; in fact $r_2 \nvdash Ordering\ [Int]$ holds for any instantiation of $\alpha$. This demonstrates the value of the $\vdash\text{unamb}$ predicate on context types, which prevents rules such as $r_2$ from ending up in the implicit context.

There is a subtlety here worth mentioning: the restriction that type variables must appear in the head of an implicit value type is limited to variables bound in the implicit value; other type variables cannot be a source of ambiguity as they are not instantiated by resolution.
For example, an implicit value with the following type doesn’t violate the condition, even though its head does not contain the free type variable \( \beta \).

\[
\forall \alpha. (\beta \Rightarrow \alpha) \Rightarrow \alpha
\]

The condition \( \vdash_{\text{UNAMB}} \) is also called the “Bound Variable Condition”; it appears in the functional-dependency-conditions on type classes in Haskell [SDPJS07].

To guarantee termination of the recursive algorithm that corresponds to the rules however, the types of rules must be restricted more. Such a recursive algorithm is still vulnerable to the trap that we tried to sketch in section 1.2.1: the algorithm cannot predict beforehand whether a search path will ultimate be finite, or diverge. In section 5.1.2 we look at a stronger condition on rules in the context \( \vdash_{\text{TERM}} \) that does guarantee termination.

Assuming termination, Oliveira et al. prove that there is a resolution algorithm that is sound w.r.t. \( \lambda_{\rightarrow}^D \). The completeness proof is left as an exercise for the reader, but is said to proceed similar to the soundness proof, such that we have the following theorem:

3.2.1 **Theorem**: \( \lambda_{\rightarrow}^D \) is decidable

### 3.3 1½ Order Resolution

At first sight, \( \lambda_{\rightarrow}^D \) seems to do the impossible: a decidable form of resolution that does not limit itself to the first order fragment of System F. Of course there is a catch: it only resolves a subset of all expressions that are derivable under the rules of \( \lambda_{\rightarrow}^A \). We make this precise using the soundness of \( \lambda_{\rightarrow}^D \) w.r.t. \( \lambda_{\rightarrow}^A \) and then proving there are theorems in \( \lambda_{\rightarrow}^A \) that are non-theorems in \( \lambda_{\rightarrow}^D \). Soundness is already proven by Oliveira et al., but we include a version of this proof here for completeness. For the soundness proof we need the following lemma, which is mutually inductive with soundness.

3.3.1 **Lemma**: For every \( \Delta \vdash a \downarrow \tau \) we have \( \Delta \vdash_{\rightarrow}^A a \rightarrow \Delta \vdash_{\rightarrow}^A \tau \).

**Proof.** By induction on the structure of \( \Delta \vdash a \downarrow \tau \) and assuming \( \Delta \vdash_{\rightarrow}^A a \) we derive the implication \( \Delta \vdash_{\rightarrow}^A \tau \) for every case:

**Case:** \( \Delta \vdash \tau \downarrow \tau \) (i-simp), \( h_1 : \Delta \vdash_{\rightarrow}^A \tau \)

Trivial.

**Case:** \( h_1 : \Delta \vdash_{\rightarrow}^A a \quad h_2 : \Delta \vdash b \downarrow \tau \) (i-abs), \( h_3 : \Delta \vdash_{\rightarrow}^A a \rightarrow b \)

Applying the IH to \( h_2 \) we obtain \( \Delta \vdash_{\rightarrow}^A b \rightarrow \Delta \vdash_{\rightarrow}^A \tau \).
Using the soundness IH on $h_1$ we get $\Delta \vdash^A \Gamma$ and, using $h_3$, $\Delta \vdash^A b$ follows from i-iabs.

The conclusion $\Delta \vdash^A \tau$ is now inevitable, following from implication-elimination.

Case: \[
\frac{h_1: \Delta \vdash a \{\beta \mapsto b\} \downarrow \tau \quad h_2: \Delta \vdash^A \forall \beta. a}{\Delta \vdash^A \forall \beta. a} \quad (i\text{-tabs})
\]

The IH is $\Delta \vdash^A r\{\beta \mapsto a\} \rightarrow \Delta \vdash^A \tau$. We derive $\Delta \vdash^A r\{\beta \mapsto a\}$ from $h_2$ using r-tapp and the conclusion $\Delta \vdash^A \tau$ follows immediately.

\[ \square \]

3.3.2 Theorem [Soundness of $\lambda_D^\Rightarrow$]: For any context $\Delta$ and goal $\Gamma$, $\Delta \vdash^D \Gamma$ implies $\Delta \vdash^A \Gamma$.

Proof. By induction on the structure of $\Delta \vdash^D \Gamma$:

Case: \[
\frac{h_1: \Delta(\tau) = a \quad h_2: \Delta \vdash a \downarrow \tau}{\Delta \vdash^A \tau} \quad (r\text{-simp})
\]

$h_1$ states that $a$ is the first rule in $\Delta$ such that $a \triangleleft \tau$. From $h_2$ we get the additional knowledge that $\Delta \vdash a \downarrow \tau$. We apply r-ivar to obtain $\Delta \vdash^A a$. The conclusion now follows from Lemma 3.3.1.

Case: \[
\frac{h_1: \Delta; a \vdash^D \Gamma \quad h_2: \Delta \vdash r \downarrow b}{\Delta \vdash^A r \downarrow b} \quad (r\text{-tabs})
\]

Trivially from the induction hypothesis and the rule r-iabs.

Case: \[
\frac{h_1: \beta \notin \Delta \quad h_2: \Delta; (\beta : \ast) \vdash^D \Gamma \quad h_3: \Delta \vdash^A \forall \beta. a}{\Delta \vdash^A \forall \beta. a} \quad (r\text{-tabs})
\]

Trivially from the induction hypothesis and the rule r-tabs.

\[ \square \]

3.3.3 Theorem [Incompleteness of $\lambda_D^\Rightarrow$]: $\lambda_D^\Rightarrow$ resolves a strictly smaller number of queries than $\lambda_A^\Rightarrow$.

Proof. The conclusion follows from Theorem 3.3.2 together with the example in Figure 3.5 that demonstrates that there are theorems in $\lambda_A^\Rightarrow$ that are non-theorems in $\lambda_D^\Rightarrow$.

$\lambda_D^\Rightarrow$ has the very useful properties of being deterministic and decidable, while not limiting itself to a first order subset of System F. The fact that it’s incomplete w.r.t. the ambiguous system is unsurprising, but the example in Figure 3.5 concisely demonstrates that there are seemingly trivial queries that $\lambda_D^\Rightarrow$ cannot resolve! Both the strengths and weaknesses of $\lambda_D^\Rightarrow$ stem from the the implementation of the second fase of $\lambda_D^\Rightarrow$: matching.

38
let $\Delta := \emptyset; \text{Bool}; (\text{Int} \Rightarrow \text{Bool})$

$$\text{Bool} \in \Delta$$

$$\Delta \vdash_r \text{Bool} \quad \text{R-IVar}$$

(a) Ambiguous rules can derive $\text{Bool}$

\[
\frac{\text{Bool} \not\in \text{Int}}{\text{Int} \Rightarrow \text{Bool} \not\in \text{Int} \quad \text{Int} \Rightarrow \text{Bool} \not\in \text{Int}}
\]

\[
\frac{\exists r. \Delta(\text{Int}) = r}{\Delta \not\vdash_r \text{Int} \Rightarrow \text{Bool}}
\]

(b) Deterministic rules can not derive $\text{Bool}$

Figure 3.5: Incompleteness of the deterministic w.r.t. the ambiguous rules

In $\lambda^A_\Rightarrow$, the derivation rule R-IVAR allows a non-deterministic choice of any “matching” value from the implicit context. Matching in $\lambda^D_\Rightarrow$ is implemented using $\Delta(\tau) = a$, which imposes an ordering on the implicit context to ensure determinacy. The actual matching against types in the implicit context is described by the $a \triangleleft \tau$ relation, which suspiciously ignores the context of rules in the m-iabs case. The relation $a \triangleleft \tau$ thus describes a weak match of $a$ with $\tau$, which does not guarantee a successful derivation. $\Delta \vdash a \downarrow \tau$ is then used to verify that all proof obligations of any rules used can actually be met, thus implementing a strong match.

There are up- and downsides to this separation of concerns. The upside is that search using weak matching sidesteps all issues that stem from a search with a matching relation that is mutually recursive with the resolution judgment $\Delta \vdash_r a$. It’s not immediately clear whether such a recursive relation is well-founded; such that its decidability is much less obvious (see section 3.4). The downside is that because the search is conducted using a weak form of matching, there is no backtracking if a weak match fails to satisfy the stronger predicate $\Delta \vdash a \downarrow \tau$. The example in Figure 3.5 demonstrates that this inability to backtrack severely restricts the expressiveness of the formal system. We capture the main point of the example in the following unfortunate truth about $\lambda^D_\Rightarrow$:

3.3.4 Lemma [Inverse of weakening for $\lambda^D_\Rightarrow$]: Assuming two implicit contexts $\Delta$ and $\Delta'$ such that $\Delta' \supseteq \Delta$ we have that $\Delta \vdash^D_r a$ does not imply $\Delta' \vdash^D_r a$.

Proof. We give a counterexample of $\Delta \vdash^D_r a \rightarrow \Delta' \vdash^D_r a$. Assume the following implicit contexts:
\[ \Delta := \emptyset; \text{Bool} \]

\[ \Delta' := \Delta; (\text{Int} \Rightarrow \text{Bool}) \]

Such that \( \Delta' \supseteq \Delta \) holds. Obviously we have:

\[
\frac{\text{ Bool } \triangleleft \text{ Bool}}{\Delta (\text{Bool}) = \text{Bool}} \quad \frac{\Delta \vdash \text{Bool} \downarrow \text{Bool}}{\Delta \vdash r \text{Bool}}
\]

The derivation in Figure 3.5 however proves that \( \Delta' \not\vdash r \text{Bool} \).

Lemma 3.3.4 is very counter-intuitive from a proof search perspective: if a context \( \Delta \) provides enough evidence to prove \( a \), then any extension \( \Delta' \) such that \( \Delta' \supseteq \Delta \) should surely also contain sufficient evidence.

Some readers familiar with Scala might observe that the same actually also holds for Scala’s implicits: adding a value to the implicit context may cause resolution to be ambiguous or diverge. In those cases the Scala compiler will abort resolution and report an error.

Indeed there are practical reasons for the fact that weakening may not hold for some implementations of resolution in a real language. As a property of a formal language however, it is rather suspicious. The example used in the proof demonstrates to what extent the formal language is weakened.

With this result we can now summarize the results of this chapter quite nicely with the following map of inclusions:

![Diagram](image)

Figure 3.6: The design space between \( \lambda_\Rightarrow^D \) and \( \lambda_\Rightarrow^A \).
We have shown that, while decidable, $\lambda^D_\triangleright$ is not the ideal formalization and we have also proven that $\lambda^A_\triangleright$ is both the maximally expressive formalization of resolution and undecidable. The remainder of this report explores the design space in between $\lambda^D_\triangleright$ and $\lambda^A_\triangleright$.

### 3.4 Naively Strengthening Matching

We noted that the determinacy and decidability come at the high price of not being able to resolve seemingly trivial queries. We argued that this was caused by the weak match implemented by $\tau \triangleleft a$. We can explore an alternative formulation $\lambda^{D+}_\triangleright$ based on the idea of using the stronger matching predicate $\Delta \vdash a \downarrow \tau$ during the search. The main benefit of matching using this stronger relation, would be the ability to backtrack when we fail to satisfy one of the proof obligations of an implicit rule.

In chapter 4 we describe such a stronger resolution relation and discuss algorithms to decide it. It is useful to motivate the approach taken there by demonstrating how the straightforward approach outlined here falls short.

#### 3.4.1 The Rules of $\lambda^{D+}_\triangleright$

The essence of $\lambda^{D+}_\triangleright$ is to retire $\tau \triangleleft a$ and replace it with the stronger relation $\Delta \vdash a \downarrow \tau$. These two relations were designed to mirror each other [OSC+12], so this results in a very similar set of rules. While the changes look subtle, the three main judgments now all operate in mutual recursion with each other. The resulting rules are summarized in Figure 3.7. The relation $\text{first } a \in \Delta' \rightarrow \Delta \vdash a \downarrow \tau$ is a 4-ary relation, which we write like this to stress its meaning: $a$ is the first type in $\Delta'$ such that $a$ strongly matches $\tau$ under $\Delta$.

We can prove the inverse of Lemma 3.3.4 to demonstrate that $\lambda^{D+}_\triangleright$ does not suffer from the same limitation as $\lambda^{D}_\triangleright$.

#### 3.4.1 Theorem [Weakening for $\lambda^{D+}_\triangleright$]: Assuming two implicit contexts $\Delta$ and $\Delta'$ such that $\Delta' \supseteq \Delta$ we have that $\Delta \vdash^{D+}_\triangleright a$ implies $\Delta' \vdash^{D+}_\triangleright a$.

Proof sketch. The complete proof is left as an exercise for the reader, but the following example should demonstrate the essential difference with $\lambda^{D}_\triangleright$ (compare with the example in Figure 3.5):
The crucial observation in order to prove that $\lambda^D_\supseteq$ subsumes $\lambda^D_\Rightarrow$ is summarized in the following lemma:

3.4.2 Lemma $[\lambda^D_\Rightarrow$ matching sound w.r.t $\lambda^D_\supseteq]:$ For all implicit contexts $\Delta$, we have that
\[ \Delta(\tau) = a \quad \text{and} \quad \Delta \vdash a \downarrow \tau \] together imply:

\[ \text{first } a \in \Delta \rightarrow \Delta \vdash a \downarrow \tau \]


With this lemma we can prove that \( \lambda_{\gg}^{D^+} \) strictly subsumes \( \lambda_{\gg}^{D} \).

3.4.3 Theorem \([\lambda_{\gg}^{D^+} \text{ strictly stronger than } \lambda_{\gg}^{D}]\):

Proof. We can establish soundness straightforwardly with lemma 3.4.2.

The conclusion follows when we combine it with lemma 3.4.1 which proofs that there are theorems in \( \lambda_{\gg}^{D^+} \) that are non-theorems in \( \lambda_{\gg}^{D} \). \[\square\]

3.4.2 Deciding \( \lambda_{\gg}^{D^+} \)

If there is an algorithm that decides \( \lambda_{\gg}^{D^+} \), then we would have a nice result: a stronger, deterministic calculus of implicits. When you write this relation as an inductive datatype in Agda however, the compiler won’t accept it because the relation \( \Delta \vdash r \downarrow \tau \) is not strictly positive. The problematic argument path is highlighted gray in figure 3.7 and the negative occurrence is colored red.

Closer inspection of the l-head and l-tail rules reveals that there is something almost paradoxical to our definition. By branching on \( \Delta \vdash r \downarrow \tau \) we assume that this proposition either holds or doesn’t, which in an intuitionistic setting only holds if we prove the relation to be decidable. It would be reasonable to question the meaning of the relation if it turns out undecidable.

The exact meaning of failing to be strictly positive is rather tricky. Strict positivity is used by Agda as an approximation to decide whether a datatype is well-founded. But it’s a “sufficient condition”; not a “necessary” one. In other words: it might be possible that \( \lambda_{\gg}^{D^+} \) is well-founded, but it is certainly not obvious from the definition.

The fact that it remains unclear whether \( \lambda_{\gg}^{D^+} \) is well-founded makes us conclude that \( \lambda_{\gg}^{D^+} \) is unpractical (if not inconsistent), as a formalization of implicit resolution. The idea to strengthen matching however was a sensible one and we will take this with us in the next chapter.
Chapter 4

\(\lambda^S\): Syntax Directed Resolution

Eventhough \(\lambda^{D+}\) turned out to be an unpractical formulation of a calculus of implicits, it seems like the underlying idea of strengthening matching is a good one. If the strict positivity issue could be circumvented, one might be able to find a decision procedure. In this chapter we propose a variation of \(\lambda^{D+}\), that we’ll coin \(\lambda^S\), based on this idea. In order to get rid of the negative occurrence, we have to give up determinism of the declarative calculus. The idea is that determinism is an algorithmic quality that does not necessarily need to be captured in the declarative rules of the system. Instead it is more natural to give a deterministic algorithm that decides the more general relation\(^1\).

A resolution algorithm for \(\lambda^S\) is still vulnerable to diverging search branches. To ensure termination of an algorithm for \(\lambda^S\), we must put additional constraints on either the types in the implicit context or the calculus rules. In this chapter we will try to formulate an algorithm for \(\lambda^S\) independent of formulating a termination condition. To convince us that a decision procedure exists, provided that we choose a suitable termination condition, we will take an approach similar to [AA11] and give a resolution algorithm that is “partially complete”: i.e. it either finds the solution or diverges, but does not fail finitely.

4.1 The Rules of \(\lambda^S\)

The negative occurrence \((\Delta; s) \not\triangleright s \Downarrow \tau\) in \(\lambda^{D+}\) appears in the rule L-TAIL. Together with its positive counterpart in the rule L-HEAD it ensures that matching against the context in R-SIMP is deterministic: it prefers the first match over subsequent matches. To get rid of the negative occurrence, we give up determinism and go back to a more general, but

\(^1\) This is not uncommon for formal systems. Another example is the typing relation for many languages with subtype polymorphism, where we often have that when a typing algorithm assigns the type \(T\) to a term \(t\), the typing relation \(t : S\) holds for all types \(S\) such that \(T <:: S\). I.e. the algorithm finds the minimum type.
ambiguous form of matching, resembling the rules from the ambiguous calculus, where we had:

$$\frac{a \in \Delta}{\Delta \Downarrow r^A a} \quad (\text{R-IVar})$$

This results in the following rules, where we highlighted the differences with $\lambda_{D^+}$:

![Figure 4.1: The rules of $\lambda_S$](image)

4.2 Expressiveness of $\lambda_S$

Let’s first establish that $\lambda_S$ is at least as expressive as $\lambda_{D^+}$:

4.2.1 Theorem [Completeness w.r.t. $\lambda_{D^+}$]: For every deterministic derivation $\Delta \Downarrow r^D a$ we have $\Delta \Downarrow r^S a$

Proof. Straight-forward by induction on the shape of $a$ once we recognize that $\langle \tau \rangle = b$ and $\Delta \Downarrow D a \Downarrow \tau$ together imply the slightly weaker observation that $a \in \Delta$ and $\Delta \Downarrow S a \Downarrow \tau$. □
Of course we also want to formally establish that $\lambda_S^S$ is more expressive than $\lambda_D^S$. We can prove the better counterpart of Lemma 3.3.4 to concisely demonstrate how the ability to backtrack on matching makes $\lambda_S^S$ a significant improvement over $\lambda_D^S$. We’ll first prove the related Lemma 4.2.2, which is mutually inductive with Lemma 4.2.3.

4.2.2 Lemma: Assuming $\Delta \vdash_S a \downarrow \tau$, we have that $\Delta' \supseteq \Delta$ implies $\Delta' \vdash_S a \downarrow \tau$.

Proof. By induction on the shape of the type $a$:

Case: $\Delta \vdash \tau \downarrow \tau$ (i-simp)

Trivial.

Case: $\frac{h_1: \Delta \vdash_r a \quad h_2: \Delta \vdash b \downarrow \tau}{\Delta \vdash a \Rightarrow b \downarrow \tau}$ (i-iabs)

Using the induction hypothesis we obtain $\Delta' \vdash b \downarrow \tau$ and $\Delta' \vdash_r a$. The conclusion follows straightforwardly from i-iabs. \qed

Case: $\frac{h_1: \Delta \vdash a\{\beta \mapsto b\} \downarrow \tau}{\Delta \vdash \forall \beta. a \downarrow \tau}$ (i-tabs)

Using the induction hypothesis we obtain $\Delta' \vdash a\{\beta \mapsto b\} \downarrow \tau$. The conclusion follows from i-tabs.

4.2.3 Lemma [\lambda_S^S Resolution Weakening]: Assuming $\Delta \vdash_r^S a$ we have that $\Delta' \supseteq \Delta$ implies $\Delta' \vdash_r^S a$.

Proof. Again by induction on the shape of $a$:

Case: $\frac{h_1: a \in \Delta}{\Delta \vdash_r \tau}$ (r-simp)

Of course we have $a \in \Delta'$. Additionally, by Lemma 4.2.2 we have $\Delta' \vdash a \downarrow \tau$. The conclusion follows by r-simp.

Case: $\frac{h_1: (\Delta; a) \vdash_r b}{\Delta \vdash_r a \Rightarrow b}$ (r-iabs)

The induction hypothesis is: $(\Delta'; a) \supseteq \Delta$ implies $(\Delta'; a) \vdash_r^S b$. Assuming $\Delta' \supseteq \Delta$, we have $(\Delta'; a) \supseteq \Delta$ by extension. The conclusion $\Delta' \supseteq \Delta \Rightarrow \Delta' \vdash_r^S (a \Rightarrow b)$ now follows from implication introduction.

Case: $\frac{h_1: \Delta; (\beta : *) \vdash_r a}{\Delta \vdash_r \forall \beta. a}$ (r-tabs)
Straightforward in a similar fashion by \( r\text{-tabs} \) once we observe that \( \Delta' \supseteq \Delta \) implies \( \Delta' ; (\beta : *) \supseteq \Delta ; (\beta : *) \). 

Using Lemma 4.2.3 we can show that in \( \lambda_\triangleright^S \) we have the positive counterpart of the example in the proof of Theorem 3.3.3:

4.2.4 **Corollary:** In contrast to \( \lambda_\triangleright^D \), \( \lambda_\triangleright^S \) can derive:

\[
(\emptyset; \text{Bool}; (\text{Int} \Rightarrow \text{Bool})) \vdash r \text{ Bool}
\]

**Proof.** We start by proving \( (\emptyset; \text{Bool}) \vdash r \text{ Bool} \):

\[
\begin{align*}
\text{Bool} \triangleleft \text{Bool} & \quad \text{(m-simp)} \\
(\emptyset; \text{Bool})\langle \text{Bool} \rangle = \text{Bool} & \quad \text{(l-head)} \\
(\emptyset; \text{Bool}) \vdash \text{Bool} & \quad \text{(l-simp)} \\
(\emptyset; \text{Bool}) \vdash r \text{ Bool} & \quad \text{(r-simp)}
\end{align*}
\]

After which, the conclusion \( (\emptyset; \text{Bool}; (\text{Int} \Rightarrow \text{Bool})) \vdash r \text{ Bool} \) follows from weakening (Lemma 4.2.3).

From soundness (Theorem 4.2.1) and Corollary 4.2.4, we immediately have the following theorem:

4.2.5 **Theorem:** \( \lambda_\triangleright^S \) is strictly more expressive than \( \lambda_\triangleright^D \).

Additionally it’s useful to establish that \( \lambda_\triangleright^S \) is still sound with respect to our reference \( \lambda_\triangleright^A \).

4.2.6 **Theorem [\( \lambda_\triangleright^S \text{ Soundness} \):** For every \( \Delta, a \) and \( \tau \) we have:

1. For every \( \Delta \vdash r^S a \) we have \( \Delta \vdash r^A a \)
2. Every \( \Delta \vdash r^A a \) and \( \Delta \vdash^S a \downarrow \tau \) together imply \( \Delta \vdash r^A \tau \).

**Proof.** We prove the two parts simultaneously using induction on the structure of \( \Delta \vdash r^S a \) and \( \Delta \vdash^S a \downarrow \tau \) respectively:

1. Case: \[
\begin{align*}
& h_1 : a \in \Delta \quad h_2 : \Delta \vdash a \downarrow \tau \\
& \Delta \vdash r^S a \quad \text{(r-simp)}
\end{align*}
\]
We derive $\Delta \vdash^r_A a$ from $h_1$ using the rule r-ivar. The conclusion follows from the inner induction hypothesis.

**Case:**

$h_1: (\Delta; a) \vdash^r b$

\[
\Delta \vdash^r_A a \Rightarrow b
\]

Immediately from the induction hypothesis by R-IABS.

**Case:**

$h_1: \Delta; (\beta : *) \vdash^r a$

\[
\Delta \vdash^r \forall \beta. a
\]

Immediately from the induction hypothesis by R-TABS.

2. **Case:**

\[
\Delta \vdash^r \tau \downarrow \tau
\]

(i-SIMP), $h_1 : \Delta \vdash^A_r \tau$

Trivial by $h_1$.

**Case:**

\[
\Delta \vdash^r \tau \downarrow \tau, \quad h_2 : \Delta \vdash^r b \downarrow \tau
\]

(i-IABS), $h_3 : \Delta \vdash^A_r a \Rightarrow b$

We apply the outer IH to $h_1$ to obtain:

\[
\Delta \vdash^A_r a
\]

Now we can derive $\Delta \vdash^A_r b$ from this and $h_3$ using R-IAPP. Finally we apply the inner IH to this result and $h_2$ and conclude:

\[
\Delta \vdash^A_r \tau
\]

**Case:**

\[
\Delta \vdash^r \tau \downarrow \tau, \quad h_2 : \Delta \vdash^r a \{\beta \mapsto b\}
\]

(i-IABS), $h_2 : \Delta \vdash^A_r \forall \beta. a$

We apply R-TAPP to $h_2$ and obtain:

\[
\Delta \vdash^A_r a \{\beta \mapsto b\}
\]

The conclusion now follows from $h_1$ by application of the inner IH.

4.2.1 Closing the Gap

The above properties are essential results that prove that we have accomplished what we set out to do: develop a formalization of implicit resolution that is significantly more expressive than $\lambda_D$, but sound w.r.t. $\lambda_A$. One might pose the question how much more expressive $\lambda_S$ actually is, which leads us to one of our main results: $\lambda_S$ is **complete** with respect to $\lambda_A^\eta$. The proof of this theorem is not nearly as straightforward as the proofs relating different formalizations that we have seen so far and critically depends on the observation that derivations of $\lambda_S$ are equivalent to $\eta$-long-$\beta$-normal (EBNF) System F.
expressions. Since we also have a proof that $\lambda^A_\Rightarrow$ derivations are equivalent to well-typed System F expressions, and since it is known that every F-expression has an equivalent EBNF expression [Abe08, Thm. 4], we have the following path from $\lambda^A_\Rightarrow$ to $\lambda^S_\Rightarrow$:

$$\lambda^A_\Rightarrow \iff F \iff EBNF \iff \lambda^S_\Rightarrow \quad (4.1)$$

Well-typed $\eta$-long-$\beta$-normal forms of type $a$ are characterized by the judgment $\Gamma \vdash e \uparrow a$. This judgment is mutually inductive with the judgment $\Gamma \vdash e \downarrow a$ for so-called neutral $\eta$-long-$\beta$-normal forms (or “stuck” terms) of type $a$ [Abe08]. It also uses a predicate $\_ \vdash \text{Base}$ to distinguish base types (simple types) from polymorphic types (context types) in System F. The derivation rules of these judgments are all given in Figure 4.2.

To prove that $\lambda^S_\Rightarrow$ derivations and $\eta$-long-$\beta$-normal are equivalent we’ll use the same translation of types and contexts as we used in section 3.1 for the equivalence between $\lambda^A_\Rightarrow$ and well-typed System-F expressions. We’ll prove that the two are equivalent by giving translations back and forth. Like the translation of $\lambda^A_\Rightarrow$ into system F, we present the translation into EBNF as two mutually recursive functions $\lfloor \_ \rfloor \uparrow$ and $\lfloor \_ \rfloor \downarrow$ (Figure 4.3) that take $\lambda^S_\Rightarrow$ derivations to System-F terms. The translations from EBNF to $\lambda^S_\Rightarrow$ are too involved to be presented functionally and are presented as an induction proof.

We can prove that for all $p$, $\lfloor p \rfloor_\eta \downarrow$ is indeed a welltyped expression of type $\lfloor a \rfloor_{tp}$ and is in $\eta$-long-$\beta$-normal form. The proof of this lemma is omitted here for brevity, but follows a similar pattern as the proof of Lemma 4.2.9.

4.2.7 Lemma [\lambda^S_\Rightarrow \text{ INTO EBNF}]: $p : \Delta \vdash^S a$ implies $\lfloor \Delta \rfloor_{ctx} \vdash [p]_\eta \uparrow [a]_{tp}$.

Proof. See Agda module Implicits.Resolution.Infinite.NormalFormIso \hfill \square

We give the proof in the other direction as two mutually inductive lemmas, translating neutral and normal forms respectively. Let $\text{IH}_{ne}$ refer to the induction hypothesis on neutral-terms, and let $\text{IH}_{nf}$ be the induction hypothesis on normal-terms.
\[
\begin{align*}
\Gamma \vdash e \Downarrow a \\
\quad (x : a) \in \Gamma \quad (\text{N-VAR}) \\
\Gamma \vdash x \Downarrow a \\
\Gamma \vdash f \Downarrow (a \to b) \quad \Gamma \vdash e \Uparrow a \quad (\text{N-APP}) \\
\Gamma \vdash fe \Downarrow b \\
\Gamma \vdash f \Downarrow \forall \beta .a \\
\Gamma \vdash f[b] \Downarrow a\{\beta \mapsto b\} \quad (\text{N-TAPP})
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e \Uparrow a \\
\Gamma \vdash x \Uparrow \alpha \quad (\text{N-BASE}) \\
a \vdash_{\text{BASE}} \quad \Gamma \vdash e \Downarrow a \\
\Gamma \vdash \lambda (x : a).e \Uparrow a \to b \quad (\text{N-ABS}) \\
\Gamma ; (\beta : *) \vdash e \Uparrow a \quad (\text{N-TABS}) \\
\Gamma ; (\beta : *) \vdash e \Uparrow a \quad (\text{N-TABS}) \\
\Gamma \vdash \Lambda \beta .e \Uparrow \forall \beta .a \quad (\text{N-TABS})
\end{align*}
\]

\[
\begin{align*}
a \vdash_{\text{BASE}} \\
\alpha \vdash_{\text{BASE}} \quad (\text{BASE-TVAR}) \\
a \to b \vdash_{\text{BASE}} \quad (\text{BASE-ABS})
\end{align*}
\]

Figure 4.2: $\eta$-long-$\beta$-normal System-F expressions [Abe08]
Figure 4.3: Translation of $\lambda^S$ into $\eta$-long-$\beta$-normal System-F expressions
4.2.8 **Lemma [Neutral expressions into \( \lambda^S \Sigma \):]** \( \Gamma \vdash t \downarrow a \) and \( [\Gamma]_{ctx} \vdash [a]_{tp} \downarrow \tau \) together imply \( [\Gamma]_{ctx} \vdash r \downarrow \tau \) for some \( (x : r) \in [\Gamma]_{ctx} \).

**Proof.** We’ll perform induction on the structure of \( \Gamma \vdash t \downarrow a \).

**Case:**

\[
\begin{array}{c}
h_1: (x : a) \in \Gamma \\
\Gamma \vdash x \downarrow a \\
\end{array}
\]

\((N-VAR), h_2 : [\Gamma]_{ctx} \vdash [a]_{tp} \downarrow \tau
\]

Trivial from \( h_2 \).

**Case:**

\[
\begin{array}{c}
h_1 : \Gamma \vdash f \downarrow (a \to b) \\
h_2 : \Gamma \vdash e \uparrow a \\
\end{array}
\]

\((N-APP), h_3 : [\Gamma]_{ctx} \vdash [b]_{tp} \downarrow \tau
\]

Applying IH \( n_f \) to \( h_2 \) gives us \( [\Gamma]_{ctx} \vdash_r [a]_{tp} \). Consecutively we apply i-iabs to \( h_3 \) to derive \( [\Gamma]_{ctx} \vdash [a \to b]_{tp} \downarrow \tau \). We conclude by applying IH \( n_e \) to \( h_1 \).

**Case:**

\[
\begin{array}{c}
h_1 : \Gamma \vdash f \downarrow \forall \beta.a \\
\Gamma \vdash f[b] \downarrow a\{\beta \mapsto b\} \\
\end{array}
\]

\((N-TAPP), h_2 : [\Gamma]_{ctx} \vdash [a]_{tp} \{\beta \mapsto [b]_{tp}\} \downarrow \tau
\]

Applying i-tabs to \( h_2 \) we derive \( [\Gamma]_{ctx} \vdash [\forall \beta.a]_{tp} \downarrow \tau \). We conclude by applying IH \( n_e \) for \( h_1 \).

\[
\square
\]

4.2.9 **Lemma [EBNF into \( \lambda^S \Sigma \):]** \( \Gamma \vdash t \uparrow a \) implies \( [\Gamma]_{ctx} \vdash_r [a]_{tp} \)

**Proof.** We’ll perform induction on on the structure of \( \Gamma \vdash t \uparrow a \):

**Case:**

\[
\begin{array}{c}
h_1 : a \vdash_{BASE} \\
h_2 : \Gamma \vdash e \uparrow a \\
\end{array}
\]

\((N-BASE)
\]

An essential property here is that System F’s base types translate to simple types, such that we can apply i-simp to derive \( [\Gamma]_{ctx} \vdash [a]_{tp} \downarrow [a]_{tp} \). The conclusion follows from the application of IH \( n_e \) for \( h_2 \).

**Case:**

\[
\begin{array}{c}
h_1 : \Gamma; (x : a) \vdash e \uparrow b \\
\end{array}
\]

\((N-ABS)
\]

By application of IH \( n_f \) on \( h_1 \) we have \( ([\Gamma]_{ctx}; (x : [a]_{tp}) \vdash_r [b]_{tp} \). The conclusion \( [\Gamma]_{ctx} \vdash_r [a \to b]_{tp} \) follows trivially from r-iabs.

**Case:**

\[
\begin{array}{c}
h_1 : \Gamma; (\beta : *) \vdash e \uparrow a \\
\end{array}
\]

\((N-TABS)
\]

By application of IH \( n_f \) on \( h_1 \) we have \( [\Gamma]_{ctx}; (\beta : *) \vdash_r [a]_{tp} \). The conclusion \( [\Gamma]_{ctx} \vdash_r [\forall \beta.a]_{tp} \) follows trivially from r-tabs.

\[
\square
\]

52
4.2.10 **Theorem** $[\lambda^S \Leftrightarrow \text{EBNF}]$: $\lambda^S$ derivations are equivalent to welltyped System F expressions in $\eta$-long-$\beta$-normal form, or, more precisely:

$$\Delta \vdash^S r \iff \exists t. [\Delta]_{ctx} \vdash [r]_{tp}$$  \hfill (4.2)

**Proof.** By Lemma 4.2.9 and 4.2.7

With that equivalence in our pocket, we are ready to tackle the main result of this section:

4.2.11 **Theorem** $[\lambda^S \Leftrightarrow \lambda^A]$: $\Delta \vdash^A a$ and $\Delta \vdash^S a$ are equivalent.

**Proof.** We prove that every derivation $\Delta \vdash^A a$ of ambiguous resolution has a corresponding derivation of the same type in syntax-directed resolution; i.e. we prove completeness of $\lambda^S$ with respect to $\lambda^A$. Using soundness we can then immediately conclude that $\lambda^S$ and $\lambda^A$ are equivalent.

From $\Delta \vdash^A a$ we can derive $\Delta \vdash^S a$ by first using Theorem 3.1.3 to get a corresponding welltyped F expressions. We then normalize that F expression to $\eta$-long-$\beta$-normalform, relying on the fact that every welltyped F expression has such a normal form [Abe08, theorem 4], to obtain: $\exists e. [\Delta]_{ctx} \vdash e \uparrow [a]_{tp}$. Finally we use Theorem 4.2.10 to derive $\Delta \vdash^S a$ as promised. The following prooftree should summarize our translation:

$$\Delta \vdash^A a \quad \Rightarrow \quad \exists e. [\Delta]_{ctx} \vdash e \uparrow [a]_{tp}$$

$$\Rightarrow \quad \exists e'. [\Delta]_{ctx} \vdash e' \uparrow [a]_{tp}$$

$$\Rightarrow \quad \Delta \vdash^S a$$

We may conclude that $\lambda^S$ is complete w.r.t. $\lambda^A$, or $\Delta \vdash^A a \rightarrow \Delta \vdash^S a$. Consequently we derive the equivalence $\Delta \vdash^A a \Leftrightarrow \Delta \vdash^S a$ using this completeness result and the soundness result from Theorem 4.2.6. 

4.3 **A Partial Algorithm for** $\lambda^S$

Now that we have established that $\lambda^S$ is equivalent to $\lambda^A$, we have the following result:

4.3.1 **Theorem:** Implicit resolution in $\lambda^S$ is undecidable.

**Proof.** We show that a decider $A^S$ for $\lambda^S$ is also a decider for $\lambda^A$. Given a $\lambda^A$ resolution problem “Does $\Delta$ resolve $a$”, we feed $\Delta$ and $a$ to $A^S$. We distinguish the two possible decisions of $A^S$:

Case: $\Delta \vdash^S a$
From the equivalence (Theorem 4.2.11) we immediately also have $\Delta \vdash^A a$.

**Case:** $\neg \Delta \vdash^S a$

Assuming $\Delta \vdash^A a$ leads to $\Delta \vdash^S a$, using their equivalence. From the contradiction we conclude $\neg \Delta \vdash^A a$.

We finish the proof using Theorem 3.1.4, which states that implicit resolution in $\lambda^A_\subseteq$ is undecidable. The conclusion follows from the contradiction.

While this possibly disappoints the reader, it is a natural (and even unsurprising) result, as even the subset $\lambda^D_\subseteq$ was undecidable without the termination condition that Oliveira et al. imposed on the rules in the context. Eventhough we haven’t gained decidability or determinism in $\lambda^S_\subseteq$, comparing it to $\lambda^A_\subseteq$, we have gained syntax directedness. The syntax directed rules lead us to an algorithm for $\lambda^S_\subseteq$, in the same way that Oliveira et al. derived an algorithm for $\lambda^D_\subseteq$. The algorithm for $\lambda^S_\subseteq$ is of course partial: it either gives a decision or it diverges.

Even though the algorithm is partial, it is a useful result: we’ll still be able to prove soundness. Furthermore, similar to the approach described by A. Abel and T. Altenkirch in “A Partial Type Checking Algorithm for Type:Type” [AA11], we’ll be able to establish an appropriate, partial completeness result. Where partial completeness means that the algorithm will not fail finitely: whenever a derivation exists, the algorithm will either find it or it diverge.

Complementing such a partial algorithm with a suitable termination condition would essentially remove all diverging search paths from the problem. And we end up with a total function that decides a decidable fragment of $\lambda^S_\subseteq$. The described technique thus allows one to establish essential properties such as soundness and completeness independent of termination.

The algorithm consists of two main functions $\text{match}$ and $\text{resolve}$, that work in mutual recursion with each other. They are partial algorithms for their relational counterparts $\Delta \vdash^S a \downarrow \tau$ and $\Delta \vdash^S a$.

We will first define $\text{match}$ and prove it sound and partially complete w.r.t. the matching relation $\Delta \vdash^S a \downarrow \tau$ in section 4.3.1. Then we will do the same for $\text{resolve}$ and the main resolution judgment $\Delta \vdash^S a$ in section 4.3.2.

## 4.3.1 A Partial Matching Algorithm

In $\lambda^D_\subseteq$ two types of matching were used: a weak match $a \triangleleft \tau$ and a strong match $\Delta \vdash a \downarrow \tau$. The algorithm described by Oliveira et al uses the overlap between these two relations to optimize. During weak matching, they gather the contexts of the applied rules; if a weak
match is found they simply recursively try to resolve the gathered contexts. They then prove that the strong match follows from the weak match and a successful resolution of those contexts. This optimization works because their algorithm does not backtrack on failed context resolution.

In contrast, our algorithm does need to be able to backtrack; to accomplish this, matching needs to be interleaved with context resolution.

Algorithmic matching of a context type $a$ with some simple type $\tau$ is defined by structural recursion on $a$ (Figure 4.4). Besides $a$ and $\tau$ it also gets the implicit context $\Delta$ as an input and a set of unification variables $\bar{\alpha}$. The latter is passed down during matching, because instantiation of type variables cannot be non-deterministically guessed by the algorithm at the point where they are introduced; instead this is delayed until the head of the type $a$ is reached, where the correct instantiations can be computed by unification with $\tau$ [OSC+12].

For type abstraction $\forall \beta . a$, $match$ adds the type variable $\beta$ to the set of unification variables $\bar{\alpha}$ and then continues matching with $a$. On rule types $a \Rightarrow b$ it continues matching on the return type $b$. If this is successful, the instantiations of all open type variables $\bar{\alpha}$ are returned as a vector $\bar{u}$. This is used to instantiate the polymorphic domain of the rule $a$. This instantiated domain $a\{\bar{\alpha} \mapsto \bar{u}\}$ is then the subject of recursive resolution. Matching a simple type $\tau'$ against $\tau$ is done by trying to unify the former with the latter under the set of open type variables $\bar{\alpha}$. For this we assume a function $mgu$ which takes the set of unification variables and the two types to unify as inputs and returns either nothing when the types are not unifiable, or just $\bar{u}$ if $\tau'\{\bar{\alpha} \mapsto \bar{u}\} \equiv \tau$ [McB03].

$$match : \text{List Type} \rightarrow \text{SimpleType} \rightarrow \text{Type} \rightarrow \text{Bool}$$

$$match \Delta \tau \tau = \text{case} (\text{match}' \emptyset \Delta \tau \tau) \text{ of}$$

$$\text{just } \_ \rightarrow \text{true}$$

$$\text{nothing} \rightarrow \text{false}$$

where

$$\text{match}' : \text{List TyVar} \rightarrow \text{Type} \rightarrow \text{Maybe} (\text{List Type})$$

$$\text{match}' \bar{\alpha} (a \Rightarrow b) =$$

$$(\text{match}' \bar{\alpha} \Delta \tau b) \geqslant$$

$$(\lambda \bar{u} \rightarrow \text{if} (\text{resolve} \Delta (a \{\bar{\alpha} \mapsto \bar{u}\})) \text{ then } (\text{just} \ \bar{u}) \ \text{else} \ \text{nothing})$$

$$\text{match}' \bar{\alpha} (\forall' \beta . a) =$$

$$\text{fmap} \ \text{tail} (\text{match}' (\beta :: \bar{\alpha}) \Delta \tau a)$$

$$\text{match}' \bar{\alpha} \tau' = \ mgu \ \bar{\alpha} \tau \tau'$$

Figure 4.4: The function $match$.

We can prove the following soundness lemma, which is mutually coinductive with the main

\[\text{We use Haskell’s Maybe datatype to signal that a function may fail; it has two constructors just } x \text{ and nothing that indicate success and failure respectively. We also assume the usual Monad and Functor instances for this type, such that we may use fmap and } \geqslant \text{ freely.} \]
soundness result of the resolution algorithm (Theorem 4.3.5):

4.3.2 Lemma [Soundness of matching]: If “match $\vec{\alpha} \Delta \tau \ a$” terminates with just $\vec{u}$, then $\Delta \vdash a\{\vec{\alpha} \mapsto \vec{u}\} \downarrow \tau$ holds.

Proof. To mix it up, we proceed by mixed induction/coinduction on $a$. We focus on the interesting case where the recursive calls terminate with success$^3$.

Case: $\tau'$

If $mgu \vec{\alpha} \tau' \tau$ returns just $\vec{u}$, then it follows from the soundness of unification [McB03] that

$$\tau'\{\vec{\alpha} \mapsto \vec{u}\} \equiv \tau$$

The conclusion follows immediately from $1$-SIMP.

Case: $a \Rightarrow b$

Focusing on the interesting case, let:

$$match' \vec{\alpha} \tau \ b \equiv just \ \vec{u}$$

$$resolve \ \Delta \ a\{\vec{\alpha} \mapsto \vec{u}\} \equiv true$$

From the induction hypothesis we have:

$$\Delta \vdash b\{\vec{\alpha} \mapsto \vec{u}\} \downarrow \tau$$

And, by the coinductive hypothesis about soundness of $resolve$, we have:

$$\Delta \vdash_r a\{\vec{\alpha} \mapsto \vec{u}\}$$

The conclusion follows from $1$-IABS.

Case: $\forall \beta. a$

Let:

$^3$Failure is propagated, such that soundness holds trivially for those cases.
match' (β :: ⃗ α) Δ τ a ≡ just (u :: ⃗ u)

From the induction hypothesis we have:

\[ Δ ⊢ a{β :: ⃗ α ↦ u :: ⃗ u} ↓ τ \]

Which is the same as:

\[ Δ ⊢ (a{⃗ α ↦ ⃗ u}){β ↦ u} ↓ τ \]

By i-tabs we can conclude:

\[ Δ ⊢ (∀β.𝑎){⃗ α ↦ ⃗ 𝑢} ↓ 𝜏 \]

For the top level use of match we have ⃗ α ≡ ∅, such that \( r{⃗ α ↦ ⃗ 𝑢} \) reduces to \( r \), leading to the main soundness theorem for the matching algorithm.

Algorithmic completeness for matching would mean that the existence of a derivation \( Δ ⊢ 𝑟 ↓ 𝜏 \) implies that match \( Δ 𝜏 𝑟 \) will terminate with just \( ⃗ 𝑢 \). Obviously this is stronger than what we can prove about match as we have established that it may diverge. The following partial completeness lemma can be proven however:

4.3.3 Lemma [Partial Completeness of Matching]: For every derivation \( Δ ⊢ 𝑟{⃗ α ↦ ⃗ 𝑢} ↓ 𝜏 \) we have: if match' \( ⃗ α 𝜏 r \) terminates, it terminates with just \( ⃗ 𝑢' \), assuming \( ⃗ α ⊢ \text{unamb} r \) holds.

Proof. The proof proceeds by mixed induction/coinduction on \( r \). The proof is mutually coinductive with the proof of the completeness of resolve. For brevity we focus here on the interesting cases where the recursive calls terminate. Other cases are trivial. Let \( h_0: Δ ⊢ r ↓ τ \).

Case: \( τ', h_0 \)

Usually the case for simple types is trivial, but it’s not entirely obvious yet why mgu \( ⃗ α τ';τ \) must find a unifier. The cause of this is that while we have established that \( r \) is simple, \( r{⃗ α ↦ ⃗ 𝑢} \) is not necessarily simple; which means that we know very little about \( h_0 \). If we distinguish the different cases for \( τ' \), the problem simplifies. For each subcase we prove that a unifier for \( τ' \) and \( τ \) exists. The conclusion follows from the completeness of mgu.

Subcase: \( β \) where \( β ∈ ⃗ α, h_0 \)
In this case we don’t know in advance what \( h_0 \) looks like, but \( \beta \in \bar{\alpha} \) means that we are free to instantiate \( \beta \) to any type we like, including \( \tau \). Which means that \( \beta \) is trivially unifiable with \( \tau \).

**Subcase:** \( a \to b \), \( \Delta \vdash a\{\bar{\alpha} \mapsto \bar{u}\} \to b\{\bar{\alpha} \mapsto \bar{u}\} \downarrow \tau \) \( \text{(t-SIMP)} \)

By inversion of \( h_0 \), we can conclude that \( (a \to b)\{\bar{\alpha} \mapsto \bar{u}\} \equiv \tau \). By definition this means that \( \bar{u} \) is a unifier for \( a \to b \) and \( \tau \).

**Subcase:** \( \beta \) where \( \beta \notin \bar{\alpha} \), \( \Delta \vdash \beta \downarrow \tau \) \( \text{(t-SIMP)} \)

The shape of \( h_0 \) is due to the fact that \( \beta \notin \bar{\alpha} \), which means that \( \beta \) is not affected by the substitution. By inversion on \( h_0 \) we must again conclude that \( \beta \equiv \tau \), which makes the unification trivial.

**Case:** \( a \Rightarrow b \), \( \Delta \vdash r_1\{\bar{\alpha} \mapsto \bar{u}\} \quad \Delta \vdash r_2\{\bar{\alpha} \mapsto \bar{u}\} \downarrow \tau \) \( \text{(i-iabs)} \)

From \( h_2 \), by applying the IH, we get:

\[
\text{match } \bar{\alpha} \Delta \tau \equiv \text{just } \bar{u}'
\]

Soundness then additionally tells us that \( \Delta \vdash r_2\{\bar{\alpha} \mapsto \bar{u}'\} \downarrow \tau \). Thanks to the unambiguity condition \( \bar{\alpha} \vdash_{\text{unamb}} r_2 \) we have \( \bar{u}' \equiv \bar{u} \).

Having established that, we can use \( h_1 \) and the induction hypothesis about the completeness of \( \text{resolve} \) to establish:

\[
\text{resolve } \Delta r_1\{\bar{\alpha} \mapsto \bar{u}'\} \equiv \text{true}
\]

The conclusion follows.

**Case:** \( \forall \beta, r \), \( \Delta \vdash a\{\bar{\alpha} \mapsto \bar{u}\}\{\beta \mapsto b\} \downarrow \tau \) \( \text{(t-TABS)} \)

We can collapse the two substitutions into one to derive the following from \( h_1 \):

\[
\Delta \vdash a\{\beta :: \bar{\alpha} \mapsto u :: \bar{u}\} \downarrow \tau
\]

Applying the IH to this, we obtain:
\[ \text{match} (\beta :: \bar{\alpha}) \Delta \tau \ a \equiv \text{just} (u' :: \bar{u}') \]

This leads to the inevitable conclusion that:

\[ \text{fmap} \ \text{tail} \ (\text{match} (\beta :: \bar{\alpha}) \Delta \tau \ a) \equiv \text{just} \ \bar{u}' \]

4.3.2 A Partial Resolution Algorithm

Matching is by far the most interesting part of the algorithm. The rest of the algorithm follows immediately from the syntax directed rules of \( \lambda_S \) and can be formulated as follows:

\[
\begin{align*}
\text{match1st} & : \text{List} \ \text{Type} \rightarrow \text{List} \ \text{Type} \rightarrow \text{SimpleType} \rightarrow \text{Bool} \ ot \\
\text{match1st} \ \Delta \ \emptyset \ \tau & = \ \text{now} \ \text{false} \\
\text{match1st} \ \Delta \ (a :: \bar{\rho}) \ \tau & = \ \text{match} \ \Delta \ \tau \ a \ \triangleright= \ f \\
\text{where} \\
f \ \text{true} & = \ \text{now} \ \text{true} \\
f \ \text{false} & = \ \text{match1st} \ \Delta \ \bar{\rho} \ \tau
\end{align*}
\]

\[
\begin{align*}
\text{resolve} & : \text{List} \ \text{Type} \rightarrow \text{Type} \rightarrow \text{Bool} \ \bot \\
\text{resolve} \ \Delta \ (a \Rightarrow b) & = \ \text{resolve} \ (\Delta; \ a) \\
\text{resolve} \ \Delta \ (\forall' \ \beta . \ a) & = \ \text{resolve} \ (\Delta; \ (\beta : \ast)) \\
\text{resolve} \ \Delta \ \tau & = \ \text{match1st} \ \Delta \ \Delta \ x
\end{align*}
\]

The function \( \text{resolve} \) implements the unfolding phase of the algorithm. Once the goal has been reduced to a simple type, the algorithm continues with the matching phase, which is carried out by \( \text{match1st} \). It simply searches for the first rule \( r \) in a list of rules such that \( \text{match} \ \Delta \ \tau \ r \) returns \text{true}. The following soundness theorems follows straight-forwardly from the soundness of \( \text{match} \):

4.3.4 Lemma [Soundness of \( \text{match1st} \)]: If “\( \text{match1st} \ \Delta \ \bar{\rho} \ \tau \ r \)” terminates with true, then the following holds:

\[ \exists \ r. r \in \bar{\rho} \land \Delta \vdash r \downarrow \tau \]

4.3.5 Theorem [Soundness of \( \text{resolve} \)]: If “\( \text{resolve} \ \Delta \ r \)” terminates with true, then \( \Delta \vdash^S r \) holds.
Proof. By induction on $r$, as usual. We again focus on the “happy path”, where the recursive calls terminate with $true$.

Case: $\tau$

Trivially from the soundness of $match1st$ and $r$-simp.

Case: $a \Rightarrow b$

Assuming the recursive call terminates with true, we have $\Delta; a \vdash_r b$ from the induction hypothesis. The conclusion follows from $r$-iabs.

Case: $\forall \beta. a$

Again, assuming the recursive call terminates with true, the induction hypothesis gives us $\Delta; (\beta : \ast) \vdash_r a$. We conclude by $r$-tabs.

As was the case for $match$, we prove a partial completeness lemma about $match1st$:

4.3.6 Lemma [Partial Completeness of $match1st$]: Any $r \in \Delta$ such that $\Delta \vdash r \downarrow \tau$ implies that if $match1st$ terminates, it terminates with $true$.

Proof. Straight forward from the partial completeness of $match$ (Lemma 4.3.3).

And similarly we have a partial completeness result for $resolve$:

4.3.7 Theorem [Partial Completeness of $resolve$]: Any derivation $\Delta \vdash^S_r r$ implies that if $resolve$ terminates, it terminates with $true$.

Proof. Straight-forward by induction on $\Delta \vdash_r r$ and the completeness of $match1st$ (Lemma 4.3.6).

4.3.3 The Source of Non-Termination

While the soundness and partial completeness results look quite promising they are not very powerful theorems by themselves. This can be illustrated by the fact that the following implementation of $resolve$ also satisfies soundness and partial completeness:

\[
\text{resolve} : \text{List Type} \to \text{Type} \to \text{Bool} \\
\text{resolve} \Delta r = \text{resolve} \Delta r
\]

The cause of this is that both soundness and partial completeness only state things about terminating runs of $resolve$. Since all runs of this algorithm are non-terminating, every statement about the output of terminating runs can be proven by appealing to the
coinduction-hypothesis. We can strengthen the soundness and completeness results with a lemma that limits divergence of the algorithm to non-trivial cases:

4.3.8 Lemma: Non-terminating runs of resolve recurse into resolve infinitely many times from the $a \Rightarrow b$ case in match.

Proof. The recursion scheme of resolve is sketched in figure 4.5, where $i$ denotes the iteration.

![Figure 4.5: Recursion scheme of resolve](image)

The size measures on the recursive calls are monotonically decreasing during a run of the algorithm. The exception to this rule is the recursive call to resolve, which can potentially be called with a goal-type that is “bigger” than the previous goal. This means that every path that does not recurse into resolve is well-founded. It follows that every infinite path must recurse into resolve infinitely many times.

Lemma 4.3.8 not only precludes trivial divergence: it pins down the source of divergence. In the next chapter we will exploit this knowledge to essentially select a decidable fragment of $\lambda^S_\Rightarrow$. 

61
Chapter 5

Termination Of Resolution

There is an interesting mismatch between the declarative rules of $\lambda_S^S$ and any algorithm that tries to decide it: even if the resolution judgment is a well-founded relation, the resolution algorithm might still diverge. In other words: the knowledge that derivations will ultimately be finite, does not limit the search space. To ensure termination of resolution we have to impose an additional termination condition that does limit the depth of our search.

At the end of the previous chapter we pinned down the source of infinite regress in the partial algorithm for $\lambda_S^S$: the co-recursive call to resolve in matching. This is hardly surprising, as we already established in the introductory chapters that rules with a context-type that is not really smaller than their return-type could easily lead to diverging resolution:

\[
\text{implicit def magic[T](implicit ord: Ordering[List[T]]): Ordering[T] = ???}
\]

5.1 Finite Fragments of $\lambda_S^S$

The main idea of this chapter is to constrain recursive resolution to goals that are in some sense smaller than the current goal. We could define “smaller” in a number of ways; we will refer to such a definition as a termination measure for $\lambda_S^S$. Adding any termination measure to $\lambda_S^S$ results in a fragment that is trivially sound with respect to $\lambda_S^S$, because the termination measure only thins the set of resolvable types.

A fairly generic way to define terminating fragments of $\lambda_S^S$ is by way of some termination measure $\phi_i : \Phi$, which becomes part of the resolution judgements: $\Delta, \phi_i \vdash_r a$ and $\Delta, \phi_i \vdash a \downarrow \tau$. In the resolution algorithm, $\phi_i$ is mostly just passed around since all but one of the recursive calls were already structurally recursive. The exception was the recursive resolution of the context of an implicit rule. To ensure well-founded recursion in this case, we should at least enforce that we recurse with some $\phi_{i+1}$ such that $\phi_{i+1} < \phi_i$ for some
well-founded and decidable relation \( < \) on the termination measure. This corresponds to an additional hypothesis in i-iabs, like so:

\[
\frac{\phi_{i+1} < \phi_i \quad \Delta, \phi_{i+1} \vdash_r a \quad \Delta, \phi_i \vdash b \downarrow \tau}{\Delta, \phi_i \vdash a \Rightarrow b \downarrow \tau} \quad \text{(i-iabs)}
\]

Allowing any \( \phi_{i+1} \) such that \( \phi_{i+1} < \phi \) potentially grows the search space exponentially, because an algorithm cannot non-deterministically guess an appropriate value. We limit ourselves here for simplicity’s sake to values of \( \phi_{i+1} \) that can be computed from the information in the hypotheses of the rule i-iabs. We believe our results can be generalized for termination measures that depend on more context. With this restriction, we settle on the following definition of i-iabs:

\[
\phi_{i+1} := F(\phi_i, \Delta, a, b, \tau) \quad \phi_{i+1} < \phi_i \quad \Delta, \phi_{i+1} \vdash_r a \quad \Delta, \phi_i \vdash b \downarrow \tau \quad \text{(i-iabs)}
\]

We will refer to this family of variations of \( \lambda^S_\Sigma \), generated by different choices of \( F \), as finite fragments of \( \lambda^S_\Sigma \). Because hypotheses were only added to the formal system, we have the following rather obvious result about every finite fragment of \( \lambda^S_\Sigma \):

5.1.1 THEOREM [SOUNDNESS OF \( \lambda^S_\Sigma \)]: Every finite fragment of \( \lambda^S_\Sigma \) is sound w.r.t. \( \lambda^S_\Sigma \).

We have to adapt the algorithm slightly to account for the additional hypothesis in the rule i-iabs:

\[
\text{match}^\Phi : \Phi \to \text{List Type} \to \text{SimpleType} \to \text{Type} \to \text{Bool} \perp
\]

\[
\text{match}^\Phi \phi \Delta \tau a = \text{match}' \phi \emptyset \Delta \tau a
\]

where

\[
\text{match}' : \Phi \to \text{List TyVar} \to \text{Type} \to \text{Maybe (List Type)}
\]

\[
\text{match'} \phi_i \vec{\alpha} (a \Rightarrow b) = \text{let} \phi_{i+1} = F \phi_i \Delta a b \tau \text{ in}
\]

\[
\text{if} \phi_{i+1} < \phi_i \text{ then}
\]

\[
(\text{match}' \phi_i \vec{\alpha} b) \Rightarrow=
\]

\[
(\lambda \vec{u} \to \text{if resolve}^\Phi \phi_{i+1} \Delta (a \{ \vec{\alpha} \mapsto \vec{u} \}) \text{ then (just } \vec{u} \text{) else nothing})
\]

\[
\text{else nothing}
\]

\[
\text{match'} \ldots
\]

\[
\text{match1st}^\Phi : \Phi \to \text{List Type} \to \text{List Type} \to \text{SimpleType} \to \text{Bool}
\]

\[
\text{match1st}^\Phi \ldots
\]

63
When we draw the recursion scheme again (figure 5.1), the recursive call to \( \text{resolve}^\Phi \) is now constrained with the condition \( \phi_{i+1} < \phi_i \). This ensures that the recursion of the algorithm \( \text{resolve}^{\Phi \cdot N} \) is well-founded, leading to the following theorem about these finite fragments of \( \lambda^S \):

5.1.2 **Theorem:** Every finite fragment is decidable.

*Proof sketch.* We conclude from the recursion scheme that the recursion of the proposed algorithm is well-founded.

Formally soundness and completeness can be established using similar arguments as in the proofs of their partial counterparts in the elaboration of \( \lambda^S \). Intuitively they both follow quite naturally from the observations that:

1. \( \text{resolve}^\Phi \) terminates,
2. soundness of \( \text{resolve}^\Phi \) w.r.t. \( \lambda^S \) is preserved,
3. the only additional hypothesis of i-labs \( \phi_{i+1} < \phi_i \) is checked appropriately.

![Figure 5.1: Recursion scheme of \( \text{resolve}^{\Phi \cdot N} \)](image)

We consider several examples to give it shape.
5.1.1 \( \lambda_{S^N} \): Fixed Depth Recursion Stack

Probably the most crude approach to ensure termination is to simply limit the depth of the recursion stack. As already mentioned in section 1.3.4, this is the termination measure of Coq’s auto tactics.

In this case the termination measure is simply a counter \( \Phi := \mathbb{N} \), representing the remaining stack depth. We’ll refer to the resulting fragment of \( \lambda_{S} \) as \( \lambda^{S,N} \). The relation \(<\) reduces to the familiar well-founded strict-order on natural numbers and we choose \( F(\phi_i, \Delta, a, b, \tau) := \phi_i \triangleleft 1 \), where \( \triangleleft \) is defined as:

- \( \triangleleft \) : \( \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \)
  - \( m \triangleleft \text{zero} = m \)
  - \( \text{zero} \triangleleft \text{suc} n = \text{zero} \)
  - \( \text{suc} m \triangleleft \text{suc} n = m \triangleleft n \)

5.1.2 Oliveira’s Termination Condition

In section 3.2 we mentioned that deterministic resolution also relies on an additional well-formedness constraint on types \( a \vdash \text{TERM} \) to ensure that resolution terminates. The judgment is repeated from [OSC+12] in Figure 5.2. Oliveira et al. propose to adopt this termination condition as part of the well-formedness relation for types to integrate it into the typesystem.

In section 3.2.1 we discussed the predicate \( \vdash \text{UNAMB} \) and we noted that the side condition on type variables there, only applies to those type variables that are bound within an implicit value; because it’s only those variable that are instantiated by resolution. The termination condition does not make this distinction. This is unnecessarily strict: type variables bound outside a rule cannot increase the size of the goal during resolution and thus cannot lead to non-termination. For all intents and purposes these type-variables can be treated as fixed types of size 1 during resolution.

We could formulate \( \vdash \text{TERM} \) analogously to \( \vdash \text{UNAMB} \) to weaken the restriction while retaining termination. The resulting condition appears as one of the Paterson Conditions [SDPJS07]: an alternative set of conditions for type classes and instances in Haskell. Sulzmann et al. already note that this condition subsumes the bounded variable condition which is part of \( \vdash \text{UNAMB} \).

And while this approach would be sufficient for termination, it’s not necessary to quantify over all possible instantiations of a rule. Instead we can integrate the restriction on the size of the goal directly into the resolution rules. This way we do not preemptively reject all rule-types that do not satisfy the termination condition; but we disallow any usage of
\[ \begin{array}{c}
\tau \vdash_{\text{TERM}} \quad \forall \alpha. a \vdash_{\text{TERM}} \quad a \vdash_{\text{TERM}} \quad \forall \alpha, a \vdash_{\text{TERM}} \\
\hline
\end{array} \]

\[ \begin{array}{c}
a \vdash_{\text{TERM}} \quad b \vdash_{\text{TERM}} \quad ||a \prec|| < ||b \prec|| \\
\forall \alpha_i \in \text{ftv}(a) \cup \text{ftv}(b) : \quad \text{occ}_{\alpha_i}(a \prec) \leq \text{occ}_{\alpha_i}(b \prec) \\
\hline
a \Rightarrow b \vdash_{\text{TERM}} \quad (\text{TERM-TAB}) \\
\end{array} \]

\[ \begin{array}{c}
\text{occ}_{\alpha}(\_)
\end{array} \]

\[ \begin{array}{c}
\text{occ}_{\alpha}(\beta) = \begin{cases} 
1 & \alpha \equiv \beta \\
0 & \alpha \not\equiv \beta 
\end{cases} \\
\text{occ}_{\alpha}(a \rightarrow b) = \text{occ}_{\alpha}(a) + \text{occ}_{\alpha}(b) \\
\text{occ}_{\alpha}(a \Rightarrow b) = \text{occ}_{\alpha}(a) + \text{occ}_{\alpha}(b) \\
\text{occ}_{\alpha}(\forall \beta. a) = \text{occ}_{\alpha}(a)
\end{array} \]

\[ \begin{array}{c}
||\_||
\end{array} \]

\[ \begin{array}{c}
||\beta|| = 1 \\
||a \rightarrow b|| = 1 + ||a|| + ||b|| \\
||a \Rightarrow b|| = 1 + ||a|| + ||b|| \\
||\forall \beta. a|| = ||a||
\end{array} \]

Figure 5.2: Oliveira’s termination condition
such rules that may lead to divergence. This corresponds to the approach of the Scala implementation.

The resulting system can be formulated as a finite fragment, where the termination measure $\phi$ is the size of the head of the current goal-type $||a <||$:

\[
\frac{\phi_{i+1} := ||a <||}{\phi_i < \phi_{i+1}} \quad \Delta, \phi_{i+1} \vdash a \quad \Delta, \phi_i \vdash b \Downarrow \tau \quad (\text{r-ABS})
\]

We can prove that the resulting fragment $\lambda_S^<\Rightarrow$ is complete with respect to deterministic resolution when we assume the wellformedness-condition $a \vdash_{\text{TRIM}}$ holds for every $a$. We need the following lemma:

5.1.3 Lemma: Given $\Delta \vdash_D b \Downarrow \tau$ and $||a <|| < ||b <||$, we have $||a <|| < ||\tau||$.

Proof. Straight-forward by induction on the first assumption, once we observe that every type-instantiation can only increase the size of the head. Then $\Delta \vdash_D b \Downarrow \tau$ leads to $||b <|| < ||\tau||$, such that the conclusion follows by transitivity of $<$. \[\square\]

With this lemma we can prove the following theorem in two parts:

5.1.4 Theorem [Completeness of $\lambda_S^<\Rightarrow$]: For every context $\Delta$ and $\phi$ we have:

1. If $\Delta \vdash_r a$ and $||a <|| \leq \phi$ hold, then $\Delta, \phi \vdash_r S^< a$

2. If $\Delta \vdash_D a \Downarrow \tau$ and $||\tau|| < \phi$ hold, then $\Delta, \phi \vdash_r S^< a \Downarrow \tau$.

Proof. The two parts of the theorem are proven by mutual induction, first on the structure of $\Delta, \phi \vdash_r a$ and then on the structure of $\Delta, \phi \vdash_D a \Downarrow \tau$:

1. Case: $h_1: \Delta(\tau) = a$ \quad $h_2: \Delta \vdash a \Downarrow \tau$ (r-simp)

The second assumption of the lemma reduces to $||\tau|| \leq \phi$, such that we can apply the inner induction hypothesis to $h_2$ and derive:

$\Delta, \phi \vdash_r S^< \tau \Downarrow \tau$

Of course $h_1$ implies $r \in \Delta$, such that the conclusion follows from r-simp:

$\Delta, \phi \vdash_r S^< \tau$

67
The other cases follow from the IH in a straight-forward manner.

2. Case:

\[
\begin{align*}
  h_1 &: \Delta \vdash r \ a \\
  h_2 &: \Delta \vdash b \downarrow \tau
\end{align*}
\]

(i-abs)

Using the outer induction hypothesis on \( h_1 \) we derive:

\[
\Delta, ||a \triangleleft || \vdash S^<_r a
\]

Using the inner IH on \( h_2 \) we have:

\[
\Delta, \phi \vdash S^<_b \downarrow \tau
\]

We now make use of the fact that the decidable version of deterministic resolution relies on the wellformedness condition \( x \vdash_{\text{TERM}} \) to hold for all types \( x \). Invoking this, we have that \( (a \Rightarrow b) \vdash_{\text{TERM}} \) holds and consequently by inversion \( ||a \triangleleft || < ||b \triangleleft || \). By Lemma 5.1.3 and \( h_2 \) we derive \( ||a \triangleleft || < ||\tau|| \). From transitivity of < and the assumption \( ||\tau|| < \phi \) it then follows that:

\[
||a \triangleleft || < ||\phi \triangleleft ||
\]

With these three main ingredients the conclusion follows by i-abs:

\[
\Delta, \phi \vdash S^<_a \Rightarrow \ b \downarrow \tau
\]

The other cases follow from the induction hypothesis straight-forwardly.

The completeness Theorem 5.1.4 is a lowerbound on the expressiveness of the fragment \( \lambda^S_{<} \): it is at least as expressive as deterministic resolution. Theorem 5.1.1 gives an upperbound on its expressiveness: \( \lambda^S_{<} \) is at most as expressive as \( \lambda^S \). It can be shown that weakening still holds for \( \lambda^S \), which makes it significantly more expressive than \( \lambda^D \).

### 5.1.3 Scala’s Termination Condition

The exact termination condition that Scala implements is only specified by the compiler code. The Scala language specification describes a termination condition that is conservative in comparison to the implementation. The approach is slightly different than those that we have examined so far. Instead of imposing a condition that will ensure that only “smaller” recursive goals are added to the resolution stack, it verifies the latter directly: i.e. the algorithm keeps a stack of resolution goals and in the case of i-abs, it will not recurse on goals that are larger in some sense than any of the types on the stack.

It seems that the actual implementation is less strict. The implementation allows larger recursive goals on the stack as long as no rule is used recursively during resolution. It also permits goals to grow a bit as long as they converge soon.
We expect that the conservative formulation of the specification can be captured as a finite fragment quite naturally. Instead we tried to formulate the restriction that enforces that recursive rule application only occurs to derive goals that have decreased in size. A natural way to do this is to keep a stack of resolution goals per rule in the context; every time the rule is used, the resulting next resolution goal is added to the stack. With the limitation that goals on the stack must be of decreasing size, this ensures that every rule can only be used a finite number of times. In Scala’s typesystem, where the number of rules in the implicit context is finite, this is sufficient for termination.

Interestingly, this is less obvious for $\lambda_S$. Higher order rules may cause new rules to be added to the context, which adds a new dimension to the problem: the number of resolution rules in the context may grow infinitely large.

On the other hand we observe that rules added to the context during resolution seem to have limited originality: they have to be originate either from the context of rules in the initial implicit context or from the context of a rule-type initial goal. We hypothesize that a termination condition exists based on a resolution stack per rule “origin”. We expect that such a condition can be described as a finite fragment.
Chapter 6

The Semantics of Resolution

Uptil this chapter we have assumed that the different kinds of resolution that we’ve explored are meaningful based on our intuition that it corresponds to logical proofs. In this section we will prove that all fragments of $\lambda S$ are meaningful by relating resolution derivations $\Delta \vdash r_a$ to welltyped System F expressions $\Gamma \vdash e \in a$. Using the semantics of resolution derivations, we will be able to give a complete denotational semantics of the calculus into System F.

6.1 Translating Types and Contexts

Types are translated in the natural way by $\llbracket \_ \rrbracket_{t_p}$. Both lambda- and rule-abstractions will translate to lambdas. The translation differs from the earlier translations we used, where we mapped function types $a \to b$ to an uninterpreted arrow type $a \rightarrow b$ to ensure that the translation is bijective. The context translation $\llbracket \_ \rrbracket_{ctx}$ maps the type translation over a typing context $\Gamma$. 

70
6.2 Translating Resolution Derivations

By now we should have some intuition of how derivations correspond to expressions. Every rule just translates to a normal function and implicit arguments and implicit applications are made explicit.

The semantics of the main resolution judgement $\Delta \vdash a$ is simply that there should exists a welltyped System F expression that has the type $a$ under some context $\Gamma$, where $\Gamma$ contains a binding $(x : \mathcal{T})$ for every $b \in \Delta$, which we’ll indicate with $\Delta \sqsubseteq \Gamma$. The meaning of the match-judgment $\Delta \vdash a \downarrow \tau$ is that given any F-expression of type $a$, an F-expression exists of type $\tau$, again provided that $\Delta \sqsubseteq \Gamma$.

We define two mutually recursive functions $\llbracket \_ \rrbracket_r$ and $\llbracket \_ , \_ \rrbracket_d$ that implement this translation (Figure 6.2).

The following two mutually inductive theorems prove that the translation preserves types:

**6.2.1 Lemma $\llbracket \_ , \_ \rrbracket_d$ is Type Preserving**: Assuming $x : \Delta \vdash a \downarrow \tau$ and $\Gamma \vdash e \in \llbracket a \rrbracket_{tp}$ the following holds:

$$\Gamma \vdash \llbracket e , x \rrbracket_d \in \tau$$

Provided $\Delta \sqsubseteq \Gamma$.

*Proof.* To keep to the theme of this report so far, let’s perform some induction on $\Delta \vdash a \downarrow \tau$:

**Case:** $\Delta \vdash a \downarrow \tau \Downarrow \text{(p-simp)}, h_1 : \Gamma \vdash e \in \llbracket \tau \rrbracket_{tp}$
Trivial by $h_1$.

Case: $h_1 : \Delta \vdash a \quad h_2 : \Delta \vdash b \downarrow \tau$ (i-TABS), $h_3 : \Gamma \vdash f \in [a]_{tp} \rightarrow [b]_{tp}$

Applying the induction hypotheses to $h_1$ we derive $\Gamma \vdash [h_1]_r \in [a]_{tp}$.

Using $h_3$ and F-APP we can now derive $\Gamma \vdash f \cdot [h_1]_r \in [b]_{tp}$.

The conclusion follows from using the second induction hypothesis on $h_2$.

Case: $h_1 : \Delta \vdash a\{\beta \mapsto b\} \downarrow \tau$ (i-tabs), $h_2 : \Gamma \vdash f \in \forall \beta. [a]_{tp}$

Applying the typing rule F-TAPP on $h_2$ we have:

$$\Gamma \vdash f \left[[b]_{tp}\right] \in [a]_{tp} \{\beta \mapsto [b]_{tp}\}$$

$$\equiv \Gamma \vdash f \left[[b]_{tp}\right] \in [a\{\beta \mapsto b\}]_{tp}$$

Applying the induction hypothesis, using $h_1$, we arrive at the conclusion:

$$\Gamma \vdash \left[f \left[[b]_{tp}\right], h_1\right] \downarrow \in \tau$$

\[
6.2.2 \text{ Theorem } [\_\_\_]_r \text{ is Type Preserving: Let } p : \Delta \vdash r. a \text{ be any resolution derivation, then the following holds:}
\]

$$\Gamma \vdash [p]_r \in [a]_{tp}$$

Again provided that $\Delta \subseteq \Gamma$.

Proof. By induction on the structure of $\Delta \vdash r. a$

Case: $h_1 : a \in \Delta \quad h_2 : \Delta \vdash a \downarrow \tau$ (R-SMP)

Here we use the assumption $\Delta \subseteq \Gamma$; together with $h_1$ this implies $(x : [a]_{tp}) \in \Gamma$. By F-VAR, we may now conclude $\Gamma \vdash x \in [a]_{tp}$.

The conclusion now follows from the induction hypothesis (Lemma 6.2.1):

$$\Gamma \vdash [x, h_2] \downarrow \in \tau$$
Case: \[
\frac{h_1: (\Delta; a) \vdash_r b}{\Delta \vdash_r a \Rightarrow b \quad \text{(R-TABS)}}
\]

We apply the IH on \(h_1\) to derive:

\[
\Gamma; (x : [a]_{tp}) \vdash [h_1]_r \in [b]_{tp}
\]

Because a binding is added to the context, we verify that the sidecondition \(\Delta; a \subseteq \Gamma; (x : [a]_{tp})\) of the IH holds. Luckily this is implied by the assumption \(\Delta \subseteq \Gamma\).

The conclusion follows immediate when we apply \(f\text{-abs}\):

\[
\Gamma \vdash \lambda (x : [a]_{tp}) \cdot [h_1]_r \in a \rightarrow b
\]

Case: \[
\frac{h_1: \Delta; (\beta :: \ast) \vdash_r a}{\Delta \vdash_r \forall \beta. a \quad \text{(R-TABS)}}
\]

In a very similar fashion we apply the IH to derive:

\[
\Gamma; (\beta :: \ast) \vdash [h_1]_r \in [a]_{tp}
\]

And the conclusion follows from \(f\text{-tabs}\).

Note that thanks to our soundness and completeness results, the same translation also serves as a semantics for \(\lambda \Rightarrow_D\), \(\lambda \Rightarrow_A\) and all finite fragments of \(\lambda \Rightarrow_S\).

### 6.3 Translating Welltyped \(\lambda \Rightarrow\) Terms

With the semantics of resolution, the semantics of welltyped \(\lambda \Rightarrow\) terms is very straightforward. The denotational semantics is very similar to Oliveira’s denotational semantics of their version of \(\lambda \Rightarrow\) into System F, modulo the syntactical differences of the two calculi.

For completeness’ sake we include it here anyway as a function \([\_\_\_]\) and we prove that it preserves typings in Theorem 6.3.1.

#### 6.3.1 Theorem \([\_\_\_]\) Preserves Typings: For every welltyped \(\lambda \Rightarrow\) term \(x : \Gamma | \Delta \vdash e \in a\), we have a welltyped F expression:

\[
\left[\Gamma\right]_{ctx} \vdash [x] \in [a]_{tp}
\]
Proof. By induction on $x$.

Case: \[ h_1 : \Gamma ; (x : a) | \Delta \vdash a \quad h_2 : \emptyset \vdash_{\text{unamb}} a \]

\[
\frac{}{\Gamma | \Delta \vdash pa.e \in a \Rightarrow b}
\] \hspace{1cm} (TY-RABS)

Using the induction hypothesis on $h_1$ we derive:

\[
||\Gamma||_{ctx} : (x : [a]_{tp}) \vdash [h_1] \in [b]_{tp}
\]

The conclusion follows from F-ABS:

\[
||\Gamma||_{ctx} \vdash \lambda\left( (x : [a]_{tp}) \right) . [h_1] \in [a]_{tp} \rightarrow [b]_{tp}
\]

Case: \[ h_1 : \Gamma | \Delta \vdash e \in a \quad h_2 : \Gamma | \Delta \vdash f \in a \Rightarrow b \]

\[
\frac{}{\Gamma | \Delta \vdash r \langle e \rangle \in b}
\] \hspace{1cm} (TY-RAPP)

Applying the induction hypothesis on $h_1$ and $h_2$ we derive:

\[
||\Gamma||_{ctx} \vdash [h_1] \in [a]_{tp}
\]

and:

\[
||\Gamma||_{ctx} \vdash [h_2] \in [a]_{tp} \rightarrow [b]_{tp}
\]

The conclusion follows from F-APP:

\[
||\Gamma||_{ctx} \vdash [h_1] \cdot [h_2] \in [b]_{tp}
\]

Case: \[ h_1 : \Gamma | \Delta \vdash e \in a \Rightarrow b \quad h_2 : \Delta \vdash r . a \]

\[
\frac{}{\Gamma | \Delta \vdash r \langle \rangle \in b}
\] \hspace{1cm} (TY-RIAPP)

Similarly we apply the IH to $h_1$ and derive:

\[
||\Gamma||_{ctx} \vdash [h_1] \in [a]_{tp} \rightarrow [b]_{tp}
\]

Because R-ABS adds rule contexts to both the implicit context $\Delta$ and the typing context $\Gamma$, we have $\Delta \subseteq [\Gamma]_{ctx}$. Now we can apply Lemma 6.2.2 to $h_2$ to obtain:
The conclusion follows from F-APP:

\[ [\Gamma]_{\text{ctx}} \vdash [\llbracket h_2 \rrbracket_r \in [a]_{tp} \]

The other cases are trivial. \qed
\[
\begin{align*}
\Delta \vdash a \downarrow \tau & \quad \text{(r-simp)} \\
\Delta \vdash r \vdash a \rightarrow b & \quad \text{(r-abs)} \\
\Delta \vdash \forall \beta . a & \quad \text{(r-tabs)} \\
\end{align*}
\]

Where \((x : a) \in \Gamma???

Where \(x\) is a fresh name w.r.t. \(\Gamma\)

\[
\begin{align*}
\Delta ; a \vdash r \vdash b & \quad \text{(r-iabs)} \\
\Delta \vdash r \vdash \forall \beta . a & \quad \text{(r-tabs)} \\
\end{align*}
\]

Figure 6.2: Semantics of \(\lambda_S\) resolution derivations

76
\[
\begin{align*}
\frac{h_1 : (x : a) \in \Gamma}{\Gamma \vdash x \in a} & \quad \text{(TY-VAR)} \\
\frac{h_1 : \Gamma ; (x : a) \mid \Delta \vdash e \in b}{\Gamma \vdash \lambda (x : a).e \in a \to b} & \quad \text{(TY-ABS)} \\
\frac{h_1 : \Gamma \mid \Delta \vdash f \in a \to b \quad h_2 : \Gamma \vdash e \in a}{\Gamma \vdash f \cdot e \in b} & \quad \text{(TY-APP)} \\
\frac{h_1 : \alpha \not\in \Gamma}{\Gamma \vdash \Lambda \alpha. e \in \forall \alpha. b} & \quad \text{(TY-TABS)} \\
\frac{h_1 : \Gamma \mid \Delta \vdash e \in \forall \alpha. b \quad h_2 : \Gamma \mid \Delta \vdash c \in b}{{\Gamma \mid \Delta \vdash e[c] \in b} \quad \text{(TY-TAPP)}} & = [h_1] \left[ c \right]_{tp} \\
\frac{h_1 : \Gamma ; (x : a) \mid \Delta \vdash e \in b \quad h_2 : \emptyset \vdash \text{unamb} a}{\Gamma \vdash \rho a. e \in a \Rightarrow b} & \quad \text{(TY-RABS)} \\
\frac{h_1 : \Gamma \mid \Delta \vdash e \in a \quad h_2 : \Gamma \mid \Delta \vdash \rho (c) \in b}{\Gamma \vdash \tau \left( \begin{array}{c} \end{array} \right) \in b} & \quad \text{(TY-RAPP)} \\
\end{align*}
\]

Where \( x \) is fresh w.r.t. \( \Gamma \)

\[
\begin{align*}
\frac{h_1 : \Gamma \mid \Delta \vdash e \in a \quad h_2 : \Gamma \mid \Delta \vdash f \in a \Rightarrow b}{\Gamma \vdash \tau \left( \begin{array}{c} e \end{array} \right) \in b} & \quad \text{(TY-RAPP)} \\
\frac{h_1 : \Gamma \mid \Delta \vdash e \in b \quad h_2 : \Delta \vdash r a}{\Gamma \mid \Delta \vdash r \left( \begin{array}{c} \end{array} \right) \in b} & \quad \text{(TY-RAPP)} \\
\end{align*}
\]

Figure 6.3: The semantics of welltyped \( \lambda_S^S \) terms
Conclusions and Future Work

We set out to take a step towards a formalization of implicits that is faithful to what’s implemented in Scala. As part of this effort we have shown that the existing formalizations are unsufficient. We have proven $\lambda^A$ undecidable by relating it to the type inhabitation problem of System F. We also pinpointed the weaknesses of Oliveira’s deterministic resolution flavor $\lambda^D$. An interesting observation was that the straight forward way to improve $\lambda^D$ fell short.

This is why we decided to rethink the importance of determinism of the declarative rules of the calculus. We formulated an improvement of $\lambda^D$ that we coined “syntax directed resolution”, or $\lambda^S$. It was shown to be sound with respect to $\lambda^A$. Maybe surprisingly we could prove that while seemingly only slightly more powerful than $\lambda^D$, the improvement already closes the gap between $\lambda^D$ and $\lambda^A$: $\lambda^S$ was proven equivalent to $\lambda^A$.

The fact that $\lambda^S$ is syntax directed however, also makes it an *improvement* over $\lambda^A$ for the purposes of resolution. We show that we can define a family of finite fragments of $\lambda^S$ that have decidable resolution problems. A few example fragments are explored. An interesting finite fragment is the one that results from the termination condition of $\lambda^D$ as it’s both decidable and significantly more expressive than $\lambda^D$ itself.

The approach that we’ve taken, disentangles the concept of a finite fragment from specific termination conditions. We believe that this contributes both to our understanding of implicit resolution and to future elaborations.

Finally we present a semantics of $\lambda^S$, which should complete the picture.

We’ve made a lot of effort to present proofs in this report in a way that they are useful for the understanding of the reader. On the other hand we have also mechanized the proofs in Agda. The mechanized proofs not only underline the validity of the results, but again is also meant to support future development. The Agda compiler really can act as a *proof-assistant* and is useful in the interactive exploration of proofs and results.
Related Work

In section 1.3 we wrote about some of the existing work on implicits in other languages. We have also covered Oliveira’s work on the calculus of implicits in detail in e.g. section 3. Besides these efforts, there is some existing work that we know of that touches on subjects in this report that is worth mentioning.

Abel & Altenkirch’s Partial Completeness

The approach that we took in the chapters 4 and 5 is based on the approach of Abel and Altenkirch in [AA11]. They present an inductive typing relation $\Gamma \vdash t : A$ for a dependently typed language $\text{Type:Type}$. Then they formulate two version of algorithmic typing $\Gamma \vdash ^\mu t \sqsubseteq A$ and $\Gamma \vdash ^\nu t \sqsubseteq A$ that are identical except that the former uses an inductive $\beta$-equivalence relation, while the latter uses a coinductive equivalence relation. Due to this distinction, $\Gamma \vdash ^\nu t \sqsubseteq A$ permits typings that do not have a normalize, while $\Gamma \vdash ^\mu t \sqsubseteq A$ precludes those typings.

They then proof that terminating runs of the algorithm correspond to finite derivations $\Gamma \vdash ^\mu t \sqsubseteq A$, while non-terminating runs correspond to infinite derivations $\Gamma \vdash ^\nu t \sqsubseteq A$.

Soundness of the algorithm is now established by proving that the inductive version of algorithmic typing $\Gamma \vdash ^\mu t \sqsubseteq A$ implies declarative typing $\Gamma \vdash t : A$. They argue that because $\text{Type:Type}$ is undecidable, an appropriate completeness result is that algorithmic typing does not fail finitely [AA11]; i.e. they prove that a declarative typing derivation $\Gamma \vdash t : t$ implies a (possibly infinite) derivation $\Gamma \vdash ^\nu t \sqsubseteq A$.

Their soundness and partial completeness result for the algorithm essentially pin down the source of non-termination. Although we follow a similar technique, our approach differs from Abel & Altenkirch’s because we chose to give the algorithm in a functional, rather than a relational style. We do this because we have seen that trying to hold on to determinism in a relational settings is complicated by issues surrounding strict positivity (section 3.4.2). We then pin down the source of non-termination using the recursion scheme of the partial algorithm; it was outside our scope to formalize this argument.

Partiality Monad

The mechanization of the results was done in the language Agda. This includes the results about the partial algorithm. Because the algorithm is not guaranteed to terminate, it falls into the category of co-programs. Even though we allow the algorithm to diverge, it should still be a well-behaved mathematical function; i.e. for a given input, its output, although possibly infinite, should be well-defined. This property of co-programs is referred to as productivity and is guaranteed by the use of guarded corecursion: the delay of recursive
calls by wrapping them in a lazy constructor. One way to accomplish this is embedding the program in the *partiality monad*.

Danielsson [Dan12] discusses the partiality monad and gives a definition in Agda and proceeds to give functional operational semantics defined using the partiality monad. Stucki uses the partiality monad in an formalization of System F [Stu15] with iso-recursive types; like Danielsson he describes an operational semantics with a type soundness proof.

**Type-directed Expression Synthesis**

Tihomir et al [GKKP13] describe an algorithm to synthesize simply-typed $\lambda$-calculus expressions from the values in scope in a type-directed manner. To accomplish this they generate expression in $\eta$-long-$\beta$-normal form. Their focus is on an efficient implementation in order to be able to make suggestions to the programmer in real-time.

**Future Work**

Future development is possible in a number of directions. One direction that one might pursue is to explore termination conditions that result in even larger fragments of $\lambda^S$. We have already hypothesized in section 5.1.3 that some variation of Scala’s termination condition might work for $\lambda^S$. Having a version of Scala’s termination condition formalized would be a nice result.

Another angle to explore would be that of implicit scoping. Because every rule-abstraction introduces a new scope block, scoping is almost trivial. Scala’s scoping rules are far more involved. There are for example rules about multiple matches in one implicit scope “block” and a match with a “more precise type” might be preferred over a match that is lexically closer.

We also observed in section 2.2.2 that inference of implicit application becomes non-trivial with the introduction of rule-types. It would be interesting to see if, and under what conditions, inferring implicit application is decidable.

Perhaps the most interesting line of future work would follow from replacing the base typesystem of System F with something stronger. Scala has a very extensive typesystem, packed with features that may have an impact on implicit resolution. It would be interesting to see if our approach still works and our results still hold when for example subtyping or higher order polymorphism is added to the mix.
Acknowledgments

I owe many thanks to Sandro Stucki, who guided this research from the start. Thanks for all the insights, references, discussions, detailed commentary on this report and even the help with planning. Many thanks also to Sébastiène Doeraene, who helped to debug this report.

Thanks to the rest of the LAMP group at EPFL for the opportunity to present this work there and for their feedback. Then I want to thank Erik Meijer for teaching me many things about functional programming and types. And finally I owe thanks to my lovely fiancee, who encouraged me from the start and motivated me towards the end.
Bibliography


