The Implicit Calculus: A New Foundation for Generic Programming

BRUNO C. D. S. OLIVEIRA, The University of Hong Kong
TOM SCHRIJVERS, Ghent University
WONTAE CHOI, Seoul National University
WONCHAN LEE, Seoul National University
KWANGKEUN YI, Seoul National University
PHILIP WADLER, University of Edinburgh

Generic programming (GP) is an increasingly important trend in programming languages. Well-known GP mechanisms, such as type classes and the C++0x concepts proposal, usually combine two features: 1) a special type of interfaces; and 2) *implicit instantiation* of implementations of those interfaces.

Scala *implicits* are a GP language mechanism, inspired by type classes, that break with the tradition of coupling implicit instantiation with a special type of interface. Instead, implicits provide only implicit instantiation, which is generalized to work for *any types*. Scala implicits turn out to be quite powerful and useful to address many limitations that show up in other GP mechanisms.

This paper synthesizes the key ideas of implicits formally in a minimal and general core calculus called the implicit calculus ($\lambda_?$), and it shows how to build source languages supporting implicit instantiation on top of it. A novelty of the calculus is its support for *partial resolution* and *higher-order rules* (a feature that has been proposed before, but was never formalized or implemented). Ultimately, the implicit calculus provides a formal model of implicits, which can be used by language designers to study and inform implementations of similar mechanisms in their own languages.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Functional Languages, Object-Oriented Languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Languages

Additional Key Words and Phrases: Implicit parameters, type classes, C++ concepts, generic programming, Haskell, Scala

ACM Reference Format:
Oliveira, B. C. d. S., Schrijvers, T., Choi, W., Lee, W., Yi, K., Wadler, P. 201?. The implicit calculus: a new foundation for generic programming. ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 42 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Generic programming (GP) [Musser and Stepanov 1988] is a programming style that decouples algorithms from the concrete types on which they operate. Decoupling is achieved through parametrization. Typical forms of parametrization include parametrization by type (for example: *parametric polymorphism*, *generics* or *templates*) or parametrization by algebraic structures (such as a monoid or a group).

Author's addresses: B. C. d. S. Oliveira, Department of Computer Science, Hong Kong University; T. Schrijvers, Department of Applied Mathematics and Computer Science, Ghent University; W. Choi and W. Lee and K. Yi, . . . , Seoul National University; P. Wadler, . . . , University of Edinburgh.

A central idea in generic programming is *implicit instantiation* of generic parameters. Implicit instantiation means that, when generic algorithms are called with concrete arguments, the generic arguments (concrete types, algebraic structures, or some other form of generic parameters) are automatically determined by the compiler. The benefit is that generic algorithms become as easy to use as specialized algorithms. To illustrate implicit instantiation and its benefits consider a *polymorphic* sorting function:

$$\mathit{sort} : \forall \alpha.(\alpha \rightarrow \alpha \rightarrow \mathit{Bool}) \rightarrow \mathit{List}\ \alpha \rightarrow \mathit{List}\ \alpha$$

with 3 parameters: the type of the elements in the list (α); the comparison operator; and the list to be sorted. Instantiating all 3 parameters explicitly at every use of sort would be quite tedious. It is likely that, for a given type, the sorting function is called with the same, explicitly passed, comparison function over and over again. Moreover it is easy to infer the type parameter α. GP simplifies such calls further by also inferring the comparison operator.

$$\mathit{isort} : \forall \alpha.(\alpha \rightarrow \alpha \rightarrow \mathit{Bool}) \Rightarrow \mathit{List}\ \alpha \rightarrow \mathit{List}\ \alpha$$

By using $\Rightarrow$ instead of $\rightarrow$, the function isort declares that the comparison function is implicit (that is: automatically inferable). The function is used as in:

$$\begin{array}{l}
\mathbf{implicit}\ \mathit{cmpInt} : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool}\ \mathbf{in} \\
\mathbf{implicit}\ \mathit{cmpChar} : \mathit{Char} \rightarrow \mathit{Char} \rightarrow \mathit{Bool}\ \mathbf{in} \\
\mathbf{implicit}\ \mathit{cmpPair} : \forall \alpha\ \beta.(\alpha \rightarrow \alpha \rightarrow \mathit{Bool}, \beta \rightarrow \beta \rightarrow \mathit{Bool}) \Rightarrow (\alpha,\beta) \rightarrow (\alpha,\beta) \rightarrow \mathit{Bool}\ \mathbf{in} \\
\quad (\mathit{isort}\,[2,1,3], \mathit{isort}\,[(\texttt{'b'},1),(\texttt{'b'},0),(\texttt{'c'},2)])
\end{array}$$

The two calls of isort each take only one explicit argument: the list to be sorted. Both the type of the elements and the comparison operator are *implicitly* instantiated. The element type parameter is automatically inferred from the type of the input list. For example in the first isort call the element type call is Int, whereas in the second call the element type is $(\mathit{Char},\mathit{Int})$. More interestingly, the implicit comparison operator is automatically determined in a process called *resolution*. Resolution is a type-directed process that uses a set of *rules*, the *implicit* (or *rule*) environment, to find a value that matches the type required by the function call.

The **implicit** construct extends the implicit environment with new rules. In other words, **implicit** is a *scoping* construct for rules similar to a conventional let-binding. In this example we introduce three rules into the implicit environment. Each rule adds a previously defined function (here cmpInt, $\mathit{cmpChar}$ and $\mathit{cmpPair}$) to the implicit environment.

To infer an implicit comparison function for isort, the resolution mechanism uses the rules in the implicit environment to construct a value of the right type. In the first call of isort the function cmpInt can be used directly because it matches the type of the comparison operation needed for that call. In the second call of isort a comparison function of type $(\mathit{Char},\mathit{Int}) \rightarrow (\mathit{Char},\mathit{Int}) \rightarrow \mathit{Bool}$ is needed, but no function matches this type directly. However it is possible to combine the polymorphic function $\mathit{cmpPair}$ with cmpInt and $\mathit{cmpChar}$ to create a function of the desired type. The ability to compose functions in a type-directed manner illustrates the real power of the resolution mechanism: a finite set of rules can be used to automatically create specific instances at an infinite number of types.

1.1. Existing Approaches to Generic Programming

The two main strongholds of GP are the C++ and the functional programming (FP) communities. Many of the pillars of GP are based on the ideas promoted by Musser and Stepanov [1988]. These ideas were used in C++ libraries such as the Standard

Template Library [Musser and Saini 1995] and Boost [Boost 2010]. In the FP community, Haskell *type classes* [Wadler and Blott 1989] have proven to be well suited to GP, although their original design did not have that purpose. As years passed the FP community created its own forms of GP [Jansson and Jeuring 1996; Gibbons 2003; Lämmel and Jones 2005].

Garcia et al.'s [2003] comparative study of programming language support for GP was an important milestone for both communities. According to that study many languages provide some support for GP. However, Haskell did particularly well, largely due to type classes. A direct consequence of that work was to bring the two main lines of work on GP closer together and promote cross-pollination of ideas. Haskell adopted *associated types* [Chakravarty et al. 2005b; Chakravarty et al. 2005a], which was the only weak point found in the original comparison. For the C++ community, type classes presented an inspiration for developing language support for *concepts* [Musser and Stepanov 1988; Gregor et al. 2006; Siek and Lumsdaine 2005a].

Several researchers started working on various approaches to concepts (see Siek's work [Siek 2011] for a historical overview). Some researchers focused on integrating concepts into C++ [Dos Reis and Stroustrup 2006; Gregor et al. 2006], while others focused on developing new languages with GP in mind. The work on System F^G [Siek and Lumsdaine 2005a; 2011] is an example of the latter approach: Building on the experience from the C++ generic programming community and some of the ideas of type classes, Siek and Lumsdaine developed a simple core calculus based on System F which integrates concepts and improves on type classes in several respects. In particular, System F^G supports *scoping* of rules (in the context of C++ rules correspond to *models* or *concept_maps*).

During the same period Scala emerged as new contender in the area of generic programming. Much like Haskell, Scala was not originally developed with generic programming in mind. However Scala included an alternative to type classes: *implicits*. Implicits were initially viewed as *a poor man's type classes* [Odersky 2006]. Yet, ultimately, they proved to be quite flexible and in some ways superior to type classes. In fact Scala turns out to have fine support for generic programming [Oliveira and Gibbons 2010; Oliveira et al. 2010].

A distinguishing feature of Scala implicits, and a reason for their power, is that resolution works for *any type*. This allows Scala to simply reuse standard OO interfaces/classes (which are regular types) to model concepts, and avoids introducing another type of interface in the language. In contrast, with type classes, or the various concept proposals, resolution is tightly coupled with the type class or concept-like interfaces.

1.2. Limitations of Existing Mechanisms

Twenty years of programming experience have given the FP community insights about the limitations of type classes. Some of these limitations were addressed by concept proposals and others by implicits. We list these limitations next. As far as we know, no existing language or language proposal overcomes all limitations, as our proposal does.

Global scoping. In Haskell, rules (in the context of Haskell rules correspond to *type-class instances*) are global and there can be only a single rule for any given type [Kahl and Scheffczyk 2001; Camarão and Figueiredo 1999; Dijkstra and Swierstra 2005; Dreyer et al. 2007] in the entire program. Locally scoped rules are not available. Several researchers have already proposed to fix this issue: with named rules [Kahl and Scheffczyk 2001] or locally scoped ones [Camarão and Figueiredo 1999; Dijkstra and Swierstra 2005; Dreyer et al. 2007]. However none of those proposals have been adopted.

Both proposals for concepts and Scala implicits offer scoping of rules and as such do not suffer from this limitation.

Second class interfaces. Haskell type classes are second-class constructs compared to regular types: in Haskell, it is not possible to abstract over a type class [Hughes 1999]. Yet, the need for first-class type classes is real in practice. For example, Lämmel and Peyton Jones [2005] desire the following type class for their GP approach:

$$\begin{array}{l}\textbf{class}\ (\mathit{Typeable}\ \alpha, \mathit{cxt}\ \alpha) \Rightarrow \mathit{Data}\ \mathit{cxt}\ \alpha\ \textbf{where}\\ \quad \mathit{gmapQ} :: (\forall \beta.\mathit{Data}\ \mathit{cxt}\ \beta \Rightarrow \beta \rightarrow r) \rightarrow \alpha \rightarrow [r]\end{array}$$

In this type class, the intention is that the ctx variable abstracts over a concrete type class. Unfortunately, Haskell does not support type class abstraction. Proposals for concepts inherit this limitation from type classes. Concepts and type classes are usually interpreted as predicates on types rather than types, and cannot be abstracted over as regular types. In contrast, because in Scala concepts are modelled with types, it is possible to abstract over concepts. Oliveira and Gibbons [2010] show how to encode this example in Scala.

No higher-order rules. Finally type classes do not support higher-order rules. As noted by Hinze and Peyton Jones [2001], non-regular Haskell datatypes like:

$$\textbf{data}\ \mathit{Perfect}\ f\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{Perfect}\ f\ (f\ \alpha))$$

require type class instances such as:

$$\begin{array}{l}\textbf{instance}\ (\forall \beta.\mathit{Show}\ \beta \Rightarrow \mathit{Show}\ (f\ \beta), \mathit{Show}\ \alpha) \Rightarrow\\ \quad \mathit{Show}\ (\mathit{Perfect}\ f\ \alpha)\end{array}$$

which Haskell does not support, as it restricts instances (or rules) to be first-order. This rule is *higher-order* because it assumes another rule, $\forall \beta.\mathit{Show}\ \beta \Rightarrow \mathit{Show}\ (f\ \beta)$, that contains an assumption itself. Also note that this assumed rule is polymorphic in β.

Both concept proposals and Scala implicits inherit the limitation of first-order rules.

1.3. Contributions

This paper presents $\lambda_?$, a minimal and general core calculus for implicits and it shows how to build a source language supporting implicit instantiation on top of it. Perhaps surprisingly the core calculus itself does not provide implicit instantiation: instantiation of generic arguments is explicit. Instead $\lambda_?$ provides two key mechanisms for generic programming: 1) a type-directed resolution mechanism and 2) scoping constructs for rules. Implicit instantiation is then built as a convenience mechanism on top of $\lambda_?$ by combining type-directed resolution with conventional type-inference. We illustrate support for implicits with a simple source language.

The calculus is inspired by Scala implicits and it synthesizes core ideas of that mechanism formally. In particular, like Scala implicits, a key idea is that resolution and implicit instantiation work for any type. This allows those mechanisms to be more widely useful and applicable, since they can be used with other types in the language. The calculus is also closely related to System F^G, in that rules available in the implicit environment are lexically scoped and scopes can be nested.

A novelty of our calculus is its support for partial resolution and higher-order rules. Although Hinze and Peyton Jones [2001] have discussed higher-order rules informally and several other researchers noted their usefulness [Trifonov 2003; Rodriguez et al. 2008; Oliveira and Gibbons 2010], no existing language or calculus provides support for them. Higher-order rules are just the analogue of higher-order functions in the implicits world. They arise naturally once we take the view that resolution should work for any type. Partial resolution adds additional expressive power and it is especially useful in the presence of higher-order rules.

From the GP perspective $\lambda_?$ offers a new foundation for generic programming. The relation between the implicit calculus and Scala implicits is comparable to the relation between System F^G and various concept proposals; or to the relation between formal calculi of type classes and Haskell type classes: The implicit calculus is a minimal and general model of implicits useful for language designers wishing to study and inform implementations of similar GP mechanisms in their own languages.

In summary, our contributions are as follows.

— Our *implicit calculus* $\lambda_?$ provides a minimal formal model for implicits, which can be used for the study of implicits and GP.
— Our resolution mechanism is more expressive than existing mechanisms in the literature. It works for any type, supports local scoping, first-class interfaces, higher-order rules, as well as partial resolution. The mechanism is based on unification and resembles logic programming.
— We provide a semantics in the form of a translation from $\lambda_?$ to System F. We prove our translation to be type-preserving, ensuring soundness. The translation also serves as an effective implementation technique.
— We present a small source language built on top of $\lambda_?$ via a type-directed encoding, to demonstrate how $\lambda_?$ supports implicit instantiation and can be used to model concepts with higher-order rules.
— Finally, both $\lambda_?$ and the source language have been implemented and the source code for their implementation is available at `http://i.cs.hku.hk/~bruno/implicit`.

Organization. Section 2 presents an informal overview of our calculus. Section 3 describes a polymorphic type system that statically excludes ill-behaved programs. Section 4 provides the elaboration semantics of our calculus into System F and correctness results. Section 5 presents the source language and its encoding into $\lambda_?$. Section 6 discusses comparisons and related work. Section 7 concludes.

This paper is a rewrite and expansion of the conference paper by Oliveira et al. [2012]. It has one additional author (Wadler), whose main contribution was to suggest a simplification to the formulation of $\lambda_?$ in Section 3. The previous work had separate syntactic classes for types (τ) and type rules (ρ), and a complex construct $\forall\bar{\alpha}.\bar{\rho} \Rightarrow \tau$ that abstracts over many types and rules at once; this paper unifies types and rules into a single syntactic class (ρ) and has separate constructs $\forall\alpha.\rho$ and $\rho_1 \Rightarrow \rho_2$ that abstract over a single type or rule at a time.

The new formulation of $\lambda_?$ also differs from the conference version in that resolution is generalized further and made more expressive. Section 3 presents a discussion about the differences in terms of expressivity. Furthermore, (also in Section 3) we now include a treatment of termination and present an algorithm for resolution. Neither of these were discussed in the conference version. Finally, our formalization is more detailed, having several additional lemmas proving properties of the calculus; and we have significantly expanded our discussion of related work.

2. OVERVIEW OF THE IMPLICIT CALCULUS $\lambda_?$

Our calculus $\lambda_?$ combines standard scoping mechanisms (abstractions and applications) and types à la System F, with a logic-programming-style query language. At the heart of the language is a threefold interpretation of types:

$$\textit{types} \cong \textit{propositions} \cong \textit{rules}$$

Firstly, types have their traditional meaning of classifying terms. Secondly, via the Curry-Howard isomorphism, types can also be interpreted as propositions – in the context of GP, the type proposition denotes the availability in the implicit environment of a value of the corresponding type. Thirdly, a type is interpreted as a logic-programming style rule, i.e., a Prolog rule or Horn clause [Kowalski 1974]. Resolution [Kowalski et al. 1971] connects rules and propositions: it is the means to show (the evidence) that a proposition is entailed by a set of rules.

Next we present the key features of $\lambda_?$ and how these features are used for GP. For readability purposes we sometimes omit redundant type annotations and slightly simplify the syntax.

Fetching values by types. A central construct in $\lambda_?$ is a query. Queries allow values to be fetched by type, not by name. For example, in the following function call

$$foo\ ?Int$$

the query $?Int$ looks up a value of type Int in the implicit environment, to serve as an actual argument.

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

$$\lambda_? Int.?Int + 1$$

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

The type of the rule above is:

$$Int \Rightarrow Int$$

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

$$(\lambda_? Int.?Int + 1)\ \textbf{with}\ 1$$

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

$$\textbf{implicit}\ 1\ \textbf{in}\ (?Int + 1)$$

Both expressions return 2.

Rule Currying. Like traditional lambdas, rule abstractions can be curried. Here is a rule that computes an $Int \times Bool$ pair from implicit Int and $Bool$ values:

$$\lambda_? Int.\lambda_? Bool.(?Int + 1, \neg\ ?Bool)$$

In the body of the second rule abstraction, two implicit values (of type Int and $Bool$ respectively) are available in the implicit environment. The type of this rule is :

$$Int \Rightarrow Bool \Rightarrow Int \times Bool$$

Using two rule applications it is possible to provide the implicit values to the two rule abstractions. For example:

$$\begin{array}{l}\textbf{implicit } 1 \textbf{ in}\\ \quad \textbf{implicit } \mathit{True} \textbf{ in}\\ \quad\quad (?\mathit{Int} + 1, \neg\, ?\mathit{Bool})\end{array}$$

which returns $(2, \mathit{False})$.

Higher-order rules. $\lambda_?$ supports higher-order rules. For example, the rule

$$\lambda_? \mathit{Int}.\lambda_?(\mathit{Int} \Rightarrow \mathit{Int} \times \mathit{Int}).?(\mathit{Int} \times \mathit{Int})$$

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

$$\begin{array}{l}\textbf{implicit } 3 \textbf{ in}\\ \quad \textbf{implicit } (\lambda_? \mathit{Int}.(?\mathit{Int}, ?\mathit{Int} + 1)) \textbf{ in}\\ \quad\quad ?(\mathit{Int} \times \mathit{Int})\end{array}$$

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

Polymorphic rules and queries. $\lambda_?$ allows polymorphic rules. For example, the rule

$$\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))$$

abstracts over a type using standard type abstraction and then uses a rule abstraction to provide a value of type α in the implicit environment of the rule body. This rule has type

$$\forall\alpha.\alpha \Rightarrow \alpha \times \alpha$$

and can be instantiated to multiple rules of monomorphic types

$$\mathit{Int} \Rightarrow \mathit{Int} \times \mathit{Int}, \mathit{Bool} \Rightarrow \mathit{Bool} \times \mathit{Bool}, \ldots$$

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

$$\begin{array}{l}\textbf{implicit } 3 \textbf{ in}\\ \quad \textbf{implicit } \mathit{True} \textbf{ in}\\ \quad\quad \textbf{implicit } (\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))) \textbf{ in}\\ \quad\quad\quad (?(\mathit{Int} \times \mathit{Int}), ?(\mathit{Bool} \times \mathit{Bool}))\end{array}$$

Polymorphic rules can also be used to resolve polymorphic queries:

$$\begin{array}{l}\textbf{implicit } (\Lambda\alpha.(\lambda_?\alpha.(?\alpha, ?\alpha))) \textbf{ in}\\ \quad ?(\forall\alpha.\alpha \Rightarrow \alpha \times \alpha)\end{array}$$

Combining higher-order and polymorphic rules. The rule

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

prescribes how to build a pair of integer pairs, inductively from an integer value, by consecutively applying the rule of type

$$\forall\alpha.\alpha \Rightarrow \alpha \times \alpha$$

twice: first to an integer, and again to the result (an integer pair). For example, the following expression returns $((3,3),(3,3))$:

$$\begin{array}{l}\textbf{implicit}\ 3\ \textbf{in}\\ \quad\textbf{implicit}\ (\Lambda\alpha.(\lambda_?\alpha.(?\alpha,?\alpha)))\ \textbf{in}\\ \quad\quad ?((\mathit{Int}\times\mathit{Int})\times(\mathit{Int}\times\mathit{Int}))\end{array}$$

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

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

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

Overlapping rules. Two rules overlap if their return types intersect, i.e., when they can both be used to resolve the same query. Overlapping rules are allowed in $\lambda_?$ through nested scoping. The nearest matching rule takes priority over other matching rules. For example consider the following program:

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

In this case $\lambda n.n+1$ (of type $\mathit{Int}\to\mathit{Int}$) is the lexically nearest match in the implicit environment and evaluating this program results in 2. However, if we have the following program instead:

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

Then the lexically nearest match is $\Lambda\alpha.(\lambda x.x)$ (of type $\forall\alpha.\alpha\to\alpha$) and evaluating this program results in 1.

3. THE $\lambda_?$ CALCULUS

This section formalizes the syntax and type system of $\lambda_?$. In Section 4 (building on top of this type system) we will present the formalization of the type-directed translation to System F. However, to avoid duplication and facilitate readability, we present the rules of the type system and type-directed translation together. We use grey boxes to indicate parts of the rules which belong to the type-directed translation. These greyed parts will be explained in Section 4 and can be ignored in the remainder of this section.

3.1. Syntax

This is the syntax of the calculus:

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

Types ρ comprise four constructs: type variables α; function types $\rho_1\to\rho_2$; type abstraction $\forall\alpha.\rho$; and the novel rule type $\rho_1\Rightarrow\rho_2$. In a rule type $\rho_1\Rightarrow\rho_2$, type ρ_1 is called the *context* and type ρ_2 the *head*.

Expressions e include three abstraction-eliminination pairs. The binder $\lambda(x : \rho).e$ abstracts expression e over values of type ρ, is eliminated by application $e_1\, e_2$ and refers to the bound value with variable x. The binder $\Lambda\alpha.e$ abstracts expression e over types, is eliminated by type application $e\,\rho$ and refers to the bound type with type variable α (but α itself is not a valid expression). The binder $\lambda_?\rho.e$ abstracts expression e over implicit values of type ρ, is eliminated by implicit application e_1 **with** e_2 and refers to the implicitly bound value with implicit query $?\rho$. Without loss of generality we assume that all variables x and type variables α in binders are distinct. If not, they can be easily renamed apart to be so.

Using rule abstractions and applications we can build the **implicit** sugar that we have used in Sections 1 and 2.

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

The notation $\overline{\lambda_?\rho.}$ is a shortform for $\lambda_?\rho_1.\ \ldots\ \lambda_?\rho_n.$. Correspondingly, the notation $\overline{\textbf{with}\ e}$ is a shortform for **with** $e_1 \ldots$ **with** e_n.

For brevity, we have kept $\lambda_?$ small. Examples may use additional syntax such as built-in integers, integer operators and boolean literals and types.

3.2. Type System

Figure 1 presents the static type system of $\lambda_?$. The type system is based on the type system of System F, and every System F term is typeable in our system.

Well-Formed Types. The judgement $\Gamma \vdash \rho$ denotes the well-formedness of types with respect to type environment Γ. A type environment Γ records the type variables α and the variables x with associated type ρ in scope:

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

Types ρ are well-formed iff their free type variables occur in the type environment (`WF-VarTy`).

Well-Typed Expressions. The typing judgment $\Gamma \mid \Delta \vdash e : \rho$ means that expression e has type ρ under type environment Γ and implicit environment Δ. The implicit environment Δ is defined as:

$$\text{Implicit Environments}\ \ \Delta\ ::=\ \epsilon \mid \Delta, \rho \leadsto x$$

Most of the rules are entirely standard; only three deserve special attention. Firstly, rule (`Ty-IAbs`) extends the implicit environment with the type of an implicit value. The side condition $\epsilon \vdash_{unamb} \rho$ states that the type ρ_1 must be unambiguous; we explain this concept in Section 3.4. Secondly, rule (`Ty-IApp`) eliminates an implicit abstraction by supplying a value of the required type. Finally, rule (`Ty-Query`) resolves a particular unambiguous type ρ against the implicit environment. It is defined in terms of the auxiliary judgement $\Delta \vdash_r \rho$, which is explained next.

3.3. Resolution

The underlying principle of resolution in $\lambda_?$ originates from resolution in logic. Intuitively, $\Delta \vdash_r \rho$ holds if Δ entails ρ, where the types in Δ and ρ are read as propositions. Following the Curry-Howard correspondence, we read α as a propositional variable, $\forall\alpha.\rho$ as universal quantification, and rule types $\rho_1 \Rightarrow \rho_2$ as implication. We do not give a special interpretation to the function type $\rho_1 \rightarrow \rho_2$, treating it as an uninterpreted predicate. Unlike traditional Curry-Howard, we have two forms of arrow, functions and rules, and the important twist on the traditional correspondence is that we choose to treat rules as implications, leaving functions as uninterpreted predicates.

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

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

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

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

$$(\texttt{Ty-Var})\quad \frac{(x : \rho) \in \Gamma}{\Gamma \mid \Delta \vdash x : \rho \rightsquigarrow x}$$

$$(\texttt{Ty-Abs})\quad \frac{\Gamma \vdash \rho_1 \qquad \Gamma; x : \rho_1 \mid \Delta \vdash e : \rho_2 \rightsquigarrow E}{\Gamma \mid \Delta \vdash \lambda x : \rho_1.e : \rho_1 \to \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E}$$

$$(\texttt{Ty-App})\quad \frac{\begin{array}{c}\Gamma \mid \Delta \vdash e_1 : \rho_1 \to \rho_2 \rightsquigarrow E_1 \\ \Gamma \mid \Delta \vdash e_2 : \rho_1 \rightsquigarrow E_2\end{array}}{\Gamma \mid \Delta \vdash e_1\, e_2 : \rho_2 \rightsquigarrow E_1\, E_2}$$

$$(\texttt{Ty-TAbs})\quad \frac{\alpha \notin \Delta \qquad \Gamma, \alpha \mid \Delta \vdash e : \rho \rightsquigarrow E_1}{\Gamma \mid \Delta \vdash \Lambda \alpha.e : \forall \alpha.\rho \rightsquigarrow \Lambda \alpha.E_1}$$

$$(\texttt{Ty-TApp})\quad \frac{\Gamma \vdash \rho_1 \qquad \Gamma \mid \Delta \vdash e : \forall \alpha.\rho_2 \rightsquigarrow E}{\Gamma \mid \Delta \vdash e\, \rho_1 : \rho_2[\rho_1/\alpha] \rightsquigarrow E\ |\rho_1|}$$

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

$$(\texttt{Ty-IApp})\quad \frac{\begin{array}{c}\Gamma \mid \Delta \vdash e_1 : \rho_2 \Rightarrow \rho_1 \rightsquigarrow E_1 \\ \Gamma \mid \Delta \vdash e_2 : \rho_2 \rightsquigarrow E_2\end{array}}{\Gamma \mid \Delta \vdash e_1\ \textbf{with}\ e_2 : \rho_1 \rightsquigarrow E_1\ E_2}$$

$$(\texttt{Ty-Query})\quad \frac{\Gamma \vdash \rho \qquad \epsilon \vdash_{unamb} \rho \qquad \Delta \vdash_r \rho \rightsquigarrow E}{\Gamma \mid \Delta \vdash ?\rho : \rho \rightsquigarrow E}$$

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

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

$$(\texttt{UA-Simp})\quad \frac{\bar{\alpha} \subseteq \textit{ftv}(\rho)}{\bar{\alpha} \vdash_{unamb} \rho}$$

Fig. 1. Type System and Type-directed Translation to System F

$$\boxed{\Delta \vdash_r \rho \rightsquigarrow E}$$

$$(\texttt{R-TAbs})\ \frac{\alpha \notin \Delta \qquad \Delta \vdash_r \rho \rightsquigarrow E}{\Delta \vdash_r \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E} \qquad (\texttt{R-TApp})\ \frac{\Delta \vdash_r \forall\alpha.\rho \rightsquigarrow E}{\Delta \vdash_r \rho[\rho'/\alpha] \rightsquigarrow E\ |\rho'|}$$

$$(\texttt{R-IVar})\ \frac{\rho \rightsquigarrow x \in \Delta}{\Delta \vdash_r \rho \rightsquigarrow x} \qquad (\texttt{R-IAbs})\ \frac{\Delta, \rho_1 \rightsquigarrow x \vdash_r \rho_2 \rightsquigarrow E \qquad x\ \mathit{fresh}}{\Delta \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E}$$

$$(\texttt{R-IApp})\ \frac{\Delta \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow E_2 \qquad \Delta \vdash_r \rho_1 \rightsquigarrow E_1}{\Delta \vdash_r \rho_2 \rightsquigarrow E_2\ E_1}$$

Fig. 2. Ambiguous Resolution

Figure 2 provides a first (ambiguous) definition of the resolution judgement $\Delta \vdash_r \rho$ that corresponds to the intuition of logical implication checking. However, it suffers from two problems:

(1) The definition is *not syntax-directed*; several of the inference rules have overlapping conclusions. Hence, a deterministic resolution algorithm is non-obvious.
(2) More importantly, the definition is *ambiguous*: a derivation can be shown by multiple different derivations. For instance, there are two different derivations for $\mathit{Int}, \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int} \vdash_r \mathit{Int}$:

$$\frac{\mathit{Int} \in (\mathit{Int}, \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int})}{\mathit{Int}, \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int} \vdash_r \mathit{Int}} \qquad \frac{\dfrac{\mathit{Bool} \in (\mathit{Int}, \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int})}{\mathit{Int}, \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int} \vdash_r \mathit{Bool}}}{\mathit{Int}, \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int} \vdash_r \mathit{Int}}$$

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

3.4. Deterministic Resolution

To help defining deterministic resolution, we provide a variant of the syntax of the calculus:

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

This variant of the syntax splits types into *simple* types and *context* types. *Simple types* τ comprise two type constructs (type variables α and function types $\rho_1 \rightarrow \rho_2$). *Context types* ρ comprise the types which participate in the (recursive) resolution of rules. The type abstraction $\forall\alpha.\rho$ as well as the novel *rule types* $\rho_1 \Rightarrow \rho_2$ are the main constructs, while other (simple) types act as base cases in the resolution process. Expressions remain unchanged.

To solve the two problems Figure 3 shows a syntax-directed and unambiguous variant of resolution. The main judgement $\Delta \vdash_r \rho$ is defined by mutual recursion with the auxiliary judgement $\Delta; \rho \vdash_\downarrow \tau$. The former judgement handles proper context types ρ in the obvious way and delegates to the latter judgement for simple types τ. Note that the stratification of types into context and simple types makes all rules syntax-directed.

$$\boxed{\Delta \vdash_r \rho \rightsquigarrow E}$$

$$\texttt{(R-IAbs)}\ \frac{\Delta, \rho_1 \rightsquigarrow x \vdash_r \rho_2 \rightsquigarrow E \qquad x\ \mathit{fresh}}{\Delta \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E} \qquad \texttt{(R-TAbs)}\ \frac{\alpha \notin \Delta \qquad \Delta \vdash_r \rho \rightsquigarrow E}{\Delta \vdash_r \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E}$$

$$\texttt{(R-Simp)}\ \frac{\Delta\langle\tau\rangle = \rho \rightsquigarrow x \qquad \Delta; \rho \rightsquigarrow x \vdash_\downarrow \tau \rightsquigarrow E}{\Delta \vdash_r \tau \rightsquigarrow E}$$

$$\boxed{\Delta; \rho \rightsquigarrow E_1 \vdash_\downarrow \tau \rightsquigarrow E_2}$$

$$\texttt{(I-IAbs)}\ \frac{\Delta \vdash_r \rho_1 \rightsquigarrow E_2 \qquad \Delta; \rho_2 \rightsquigarrow E_1\ E_2 \vdash_\downarrow \tau \rightsquigarrow E_3}{\Delta; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E_1 \vdash_\downarrow \tau \rightsquigarrow E_3}$$

$$\texttt{(I-Simp)}\ \frac{}{\Delta; \tau \rightsquigarrow E \vdash_\downarrow \tau \rightsquigarrow E} \qquad \texttt{(I-TAbs)}\ \frac{\Delta; \rho[\rho'/\alpha] \rightsquigarrow E_1\ |\rho'| \vdash_\downarrow \tau \rightsquigarrow E_2}{\Delta; \forall\alpha.\rho \rightsquigarrow E_1 \vdash_\downarrow \tau \rightsquigarrow E_2}$$

$$\boxed{\Delta\langle\tau\rangle = \rho \rightsquigarrow x}$$

$$\texttt{(L-Head)}\ \frac{\rho \lhd \tau}{(\Delta, \rho \rightsquigarrow x)\langle\tau\rangle = \rho \rightsquigarrow x} \qquad \texttt{(L-Tail)}\ \frac{\rho_1 \not\lhd \tau \qquad \Delta\langle\tau\rangle = \rho_2 \rightsquigarrow y}{(\Delta, \rho_1 \rightsquigarrow x)\langle\tau\rangle = \rho_2 \rightsquigarrow y}$$

$$\boxed{\rho \lhd \tau}$$

$$\texttt{(M-Simp)}\ \frac{}{\tau \lhd \tau} \qquad \texttt{(M-TAbs)}\ \frac{\rho[\rho'/\alpha] \lhd \tau}{\forall\alpha.\rho \lhd \tau} \qquad \texttt{(M-IAbs)}\ \frac{\rho' \lhd \tau}{\rho'' \Rightarrow \rho' \lhd \tau}$$

Fig. 3. Deterministic Resolution and Translation to System F

Rule types $\rho_1 \Rightarrow \rho_2$ are resolved by pushing ρ_1 to the implicit environment and then resolving ρ_2 under that environment (R-TAbs). Type abstractions $\forall\alpha.\rho$ are resolved by peeling off the universal quantifier and then resolving ρ against the implicit environment (R-IAbs). A simple type τ is resolved in terms of the *first* matching context type ρ found in the implicit environment (R-Simp). The bias towards the first avoids ambiguity when there are multiple matching context type.

The partial function $\Delta\langle\tau\rangle$ returns the *first* matching context type found in the implicit environment. Whether a context type ρ matches a simple type τ is defined by $\rho \lhd \tau$. In essence, a context type matches a simple type if the simple type is an instance of its right-most head.

The judgement $\Delta; \rho \vdash_\downarrow \tau$ is defined by three rules that mirror the three rules of the judgement $\rho \lhd \tau$. These rules peal off from left to right the universal quantifiers and rule contexts until the target simple type is obtained:

— Universal quantifiers are eliminated by means of appropriate instantiation (`I-TAbs`). Note that thanks to the well-formedness condition on types, the type instantiation is unambiguous.
— Contexts are eliminated by means of recursive resolution (`I-IAbs`).

Note that while the rules (`I-TAbs`) and (`M-TAbs`) do not explain how the substitution $[\rho'/\alpha]$ should be obtained, there is in fact no ambiguity here. Indeed, there is at most one substitution for which the judgement holds. Consider the case of matching $\forall\alpha.\alpha \rightarrow \mathit{Int}$ with the simple type $\mathit{Int} \rightarrow \mathit{Int}$. Here the type ρ' is determined to be Int by the need for $(\alpha \rightarrow \mathit{Int})[\rho'/\alpha]$ to be equal to $\mathit{Int} \rightarrow \mathit{Int}$.

However, for the context type $\forall\alpha.\mathit{Int}$ ambiguity arises. When we match the head of this type Int with the simple type Int, the matching succeeds without actually determining how the type variable α should be instantiated. In fact, the matching succeeds under any possible substitution of α. In this particular case the ambiguity is harmless, because it does not affect the semantics. Yet, it is not so harmless in other cases. Take for instance the context type $\forall\alpha.(\alpha \rightarrow \mathit{String}) \Rightarrow (\mathit{String} \rightarrow \alpha) \Rightarrow (\mathit{String} \rightarrow \mathit{String})$.[1] Again the choice of α is ambiguous when matching against the simple type $\mathit{String} \rightarrow \mathit{String}$. Yet, now the choice is critical for two reasons. Firstly, if we guess the wrong instantiation ρ for α, then it may not be possible to recursively resolve $(\mathit{String} \rightarrow \alpha)[\alpha/\rho]$ or $(\alpha \rightarrow \mathit{String})[\alpha/\rho]$, while with a lucky guess both can be resolved. Secondly, for different choices of ρ the types $(\mathit{String} \rightarrow \alpha)[\alpha/\rho]$ and $(\alpha \rightarrow \mathit{String})[\alpha/\rho]$ can be resolved in completely different ways.

In order to avoid any problems, we conservatively forbid all ambiguous context types in the implicit environment with the $\epsilon \vdash_{unamb} \rho_1$ side-condition in rule (`Ty-IAbs`) of Figure 1.[2] The definition of $\bar{\alpha} \vdash_{unamb}$ is also given in Figure 1. Rule (`UA-TAbs`) takes care of accumulating the bound type variables $\bar{\alpha}$ before the head. Rule (`UA-IAbs`) skips over any contexts on the way to the head, but also recursively requires that these contexts are unambiguous. When the head is reached, the central rule (`UA-Simp`) checks whether all bound type variables $\bar{\alpha}$ occur in that type.

Finally, the unambiguity condition is also imposed on the queried type ρ in rule (`Ty-Query`) because this type too may extend the implicit environment in rule (`R-IAbs`).

3.5. Power of Resolution

The rules for deterministic resolution presented in this paper support all the examples described in Section 2. They are strictly more powerful than the rules presented in the conference version of the paper [Oliveira et al. 2012]. In other words, strictly more queries resolve with this article's rules than with the rules of the previous paper. For example, the query:

$$\mathit{Char} \Rightarrow \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int} \vdash_r \mathit{Char} \Rightarrow \mathit{Int}$$

does not resolve under the deterministic resolution rules of the conference paper. In order to resolve such rule types, it is necessary to add the rule type's context to the implicit environment in the course of the resolution process:

$$\mathit{Char} \Rightarrow \mathit{Bool}, \mathit{Bool} \Rightarrow \mathit{Int}, \mathit{Char} \vdash_r \mathit{Int}$$

[1]This type encodes the well-known ambiguous Haskell type $\forall\alpha.(\mathit{Show}\ \alpha, \mathit{Read}\ \alpha) \Rightarrow \mathit{String} \rightarrow \mathit{String}$ of the expression $\mathit{read} \circ \mathit{show}$.
[2]An alternative design to avoid such ambiguity would instantiate unused type variables to a dummy type, like GHC's `GHC.Prim.Any`, which is only used for this purpose.

but this was not supported by our previous set of rules. The new set of resolution rules do support this by means of rule (`RIAbs`), and queries like the above can now be resolved.

3.6. Algorithm

Figure 4 contains an algorithm that implements the non-algorithmic deterministic resolution rules of Figure 3. It differs from the latter in two important ways: 1) it computes rather than guesses type substitutions, and 2) it traverses a context type at most once per matching.

The toplevel relation of the algorithm is $\Delta \vdash_{alg} \rho$; it implements the non-algorithmic $\Delta \vdash_r \rho$ relation. The relation is defined in a syntax-directed manner, with one rule for each of the three possible forms of ρ. The first two of these three rules are essentially identical to the corresponding two rules of $\vdash_r$.

The last rule, for the form τ, differs significantly in the algorithm. Essentially, it captures the two relations $\Delta\langle\tau\rangle = \rho$ and $\Delta; \rho \vdash_{\downarrow} \tau$ into the single $\Delta \vdash_{match1st} \tau \hookrightarrow \bar{\rho}$. The two traversals of the intermediate rule type ρ are thereby replaced by a single traversal. This avoids the need to compute the instantiation of the context type ρ twice.

The other major change is that recursive invocations of $\vdash_r$ are no longer performed where the recursive contexts $\bar{\rho}$ are encountered in the auxiliary relation. Instead, the $\bar{\rho}$ involved are accumulated and returnd by $\vdash_{match1st}$, to be resolved afterwards. In a sense, we change from a post-order to a pre-order traversal of the conceptual resolution tree. This change in schedule is an outflow of the change from guessing to computing context type instantiations, which is explained below.

The two rules for the $\Delta \vdash_{match1st} \tau \hookrightarrow \bar{\rho}$ are similar to those of $\Delta\langle\tau\rangle = \rho$: they are set up to commit to the first matching ρ in the environment Δ. They are defined in terms of the auxiliary relation $\rho; \bar{\rho}; \bar{\alpha} \vdash_{match} \tau \rightarrow \bar{\rho}'$. The latter relation is the algorithmic counterpart that combines $\rho \lhd \tau$ and $\Delta; \rho \vdash_{\downarrow} \tau$. As already indicated, the part not included in this relation are the recursive invocations $\Delta \vdash_r \rho_i$ ($\rho_i \in \bar{\rho}'$). Instead $\bar{\rho}$ is an accumulating parameter for the ρ_i so they can be returned in $\bar{\rho}'$.

Essentially, the relation $\vdash_{match}$ peals off the universal quantifiers and rule contexts from the ruletype ρ until it hits the simple type τ'. The algorithm proceeds in this way because it can compute (rather than guess) the necessary type instantiation for the universal quantifiers by matching the context type's head τ' against the target simple type τ. This explains why type instantiation is postponed, and, since recursive resolution depends on type instantiation, also why recursive resolution is postponed even further.

Example 3.1. Consider for instance the matching of simple type $\mathit{Int} \rightarrow \mathit{Int}$ against context type $\forall\alpha.\alpha \Rightarrow (\alpha \rightarrow \alpha)$. Just by looking at the outer quantifier $\forall\alpha$ we do not know what α should be. Hence, we peal off the quantifier, postpone α's instantiation and proceed with $\alpha \Rightarrow (\alpha \rightarrow \alpha)$. At this point, we cannot recursively resolve the context α because we have not determined α yet. Hence, we must postpone its resolution and proceed with $\alpha \rightarrow \alpha$. Now we can determine the substitution $\theta = [\alpha/\mathit{Int}]$ by performing a matching unification with the target simple type $\mathit{Int} \rightarrow \mathit{Int}$. This substitution θ enables the postponed recursive resolution of $\alpha\theta = \mathit{Int}$.

The above informal description is formalized as follows. There is one rule for each of the three cases: pealing off a context, pealing off a universal quantifier and handling the simple type τ':

(1) As already said, the contexts are collected in the accumulating parameter $\bar{\rho}$ and be returned in $\bar{\rho}'$

(2) In the $\forall\alpha$ rule of $\vdash_{match}$ we find the main difference between the algorithmic and the non-algorithmic definitions. The non-algorithmic definition *guesses* an appropriate instantiation ρ' for the type variable α, while the algorithmic definition *computes* this instantiation. This computation does not happen in the $\forall\alpha$ rule; that rule only accumulates the type variables in the parameter $\bar{\alpha}$ of the relation.
(3) The simple type rule checks whether the target type τ matches the simple type τ'. Matching means that the rules checks whether there is a most general unifier θ' (see below) of τ and τ' whose domain consists only of the accumulated type variables $\bar{\alpha}$. The rule returns the accumulated contexts $\bar{\rho}$, but is careful to apply the unifier θ to them in order to take the matching into account.

Matching Unification. Figure 5 lists the algorithm for computing the most general matching unifier. For $\theta = mgu_{\bar{\alpha}}(\rho_1, \rho_2)$ we have that $\rho_1 = \rho_2\theta$ and $dom(\theta) \subseteq \bar{\alpha}$. Moreover, θ subsumes any other matching unifier. The algorithm itself is fairly straightforward and needs little explanation. Only rule (UAbs) deserves two notes. Firstly, we assume that α-renaming is used implicitly to use the same name β for both bound type variables. Secondly, we have to be careful that β does not escape its scope through θ, which could happen when computing for example $mgu_{\alpha}(\forall\beta.\beta, \forall\beta.\alpha)$.

3.7. Termination of Resolution

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

As an example of non-termination consider

$$Char \Rightarrow Int, Int \Rightarrow Char \vdash_r Int$$

which loops, using alternatively the first and second rule in the implicit environment. The source of this non-termination are the mutually recursive definitions of the $\vdash_r$ and $\vdash_{\downarrow}$ relations: a type is resolved in terms of a rule type whose head it matches, but this requires further resolution of the rule type's body.

3.7.1. Termination Condition. The problem of non-termination has been widely studied in the context of Haskell's type classes, and a set of modular syntactic restrictions has been imposed on type class instances to avoid non-termination [Sulzmann et al. 2007]. Adapting these restrictions to our setting, we obtain the termination judgement $\vdash_{term} \rho$ defined in Figure 6.

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

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

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

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

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

$$(\texttt{Alg-TAbs}) \quad \frac{\alpha \notin \Delta \quad \Delta \vdash_{alg} \rho \rightsquigarrow E}{\Delta \vdash_{alg} \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E}$$

$$(\texttt{Alg-IAbs}) \quad \frac{\Delta, \rho_1 \rightsquigarrow x \vdash_{alg} \rho_2 \rightsquigarrow E \qquad x\ \mathit{fresh}}{\Delta \vdash_{alg} \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda(x : |\rho_1|).E}$$

$$(\texttt{Alg-Simp}) \quad \frac{\Delta \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}; E \qquad \Delta \vdash_{alg} \rho_i \rightsquigarrow E_i \quad (\forall \rho_i \in \bar{\rho})}{\Delta \vdash_{alg} \tau \rightsquigarrow E[\bar{\omega}/\bar{E}]}$$

$$\boxed{\Delta \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}; E}$$

$$(\texttt{M1-Head}) \quad \frac{\rho; \epsilon; \epsilon; \epsilon; x \vdash_{match} \tau \hookrightarrow \bar{\rho}; \bar{\omega}'; E'}{\Delta, \rho \rightsquigarrow x \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}'; E'}$$

$$(\texttt{M1-Tail}) \quad \frac{\begin{array}{c}\rho; \epsilon; \epsilon; \epsilon; x \not\vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E' \\ \Delta \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}''; E''\end{array}}{\Delta, \rho \rightsquigarrow x \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}''; E''}$$

$$\boxed{\rho; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E'}$$

$$(\texttt{MTC-TAbs}) \quad \frac{\rho; \bar{\rho}; \bar{\alpha}, \alpha; \bar{\omega}; E\,\alpha \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E'}{\forall\alpha.\rho; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E'}$$

$$(\texttt{MTC-IAbs}) \quad \frac{\rho_2; \bar{\rho}, \rho_1; \bar{\alpha}; \bar{\omega}, \omega; E\,\omega \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E' \qquad \omega\ \mathit{fresh}}{\rho_1 \Rightarrow \rho_2; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E'}$$

$$(\texttt{MTC-Simp}) \quad \frac{\theta = \boldsymbol{mgu}_{\bar{\alpha}}(\tau, \tau')}{\tau'; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}\theta; \bar{\omega}; E|\theta|}$$

Fig. 4. Resolution Algorithm

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

4. TYPE-DIRECTED TRANSLATION TO SYSTEM F

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

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

$$\texttt{(UInst)}\ \frac{\alpha \in \bar{\alpha}}{[\alpha/\rho] = mgu_{\bar{\alpha}}(\rho, \alpha)} \qquad \texttt{(UVar)}\ \frac{}{\emptyset = mgu_{\bar{\alpha}}(\beta, \beta)}$$

$$\texttt{(UFun)}\ \frac{\theta_1 = mgu_{\bar{\alpha}}(\rho_{1,1}, \rho_{2,1}) \qquad \theta_2 = mgu_{\bar{\alpha}}(\rho_{1,2}, \rho_{2,2}\theta_1)}{\theta_2 \cdot \theta_1 = mgu_{\bar{\alpha}}(\rho_{1,1} \rightarrow \rho_{1,2}, \rho_{2,1} \rightarrow \rho_{2,2})}$$

$$\texttt{(URul)}\ \frac{\theta_1 = mgu_{\bar{\alpha}}(\rho_{1,1}, \rho_{2,1}) \qquad \theta_2 = mgu_{\bar{\alpha}}(\rho_{1,2}, \rho_{2,2}\theta_1)}{\theta_2 \cdot \theta_1 = mgu_{\bar{\alpha}}(\rho_{1,1} \Rightarrow \rho_{1,2}, \rho_{2,1} \Rightarrow \rho_{2,2})}$$

$$\texttt{(UAbs)}\ \frac{\theta = mgu_{\bar{\alpha}}(\rho_1, \rho_2) \qquad \beta \notin ftv(\theta)}{\theta = mgu_{\bar{\alpha}}(\forall\beta.\rho_1, \forall\beta.\rho_2)}$$

Fig. 5. Most General Matching Unifier

$$\boxed{\vdash_{term} \rho} \qquad \texttt{(TermSimp)}\ \frac{}{\vdash_{term} \tau} \qquad \texttt{(TermForall)}\ \frac{\vdash_{term} \rho}{\vdash_{term} \forall\alpha.\rho}$$

$$\texttt{(TermRule)}\ \frac{\begin{array}{c}\vdash_{term} \rho_1 \qquad \vdash_{term} \rho_2 \\ \rho_1 \lhd \tau_1 \qquad \rho_2 \lhd \tau_2 \qquad \|\tau_1\| < \|\tau_2\| \\ \forall\alpha \in ftv(\rho_1) \cup ftv(\rho_2) : \quad occ_\alpha(\tau_1) \leqslant occ_\alpha(\tau_2)\end{array}}{\vdash_{term} \rho_1 \Rightarrow \rho_2}$$

$$\begin{array}{rcl}
occ_\alpha(Int) & = & 0 \\
occ_\alpha(\beta) & = & \begin{cases} 1 & (\alpha = \beta) \\ 0 & (\alpha \neq \beta) \end{cases} \\
occ_\alpha(\rho_1 \rightarrow \rho_2) & = & occ_\alpha(\rho_1) + occ_\alpha(\rho_2) \\
occ_\alpha(\rho_1 \Rightarrow \rho_2) & = & occ_\alpha(\rho_1) + occ_\alpha(\rho_2) \\
occ_\alpha(\forall\beta.\rho) & = & occ_\alpha(\rho)
\end{array}$$

$$\begin{array}{rcl}
\|Int\| & = & 1 \\
\|\alpha\| & = & 1 \\
\|\rho_1 \rightarrow \rho_2\| & = & 1 + \|\rho_1\| + \|\rho_2\| \\
\|\rho_1 \Rightarrow \rho_2\| & = & 1 + \|\rho_1\| + \|\rho_2\| \\
\|\forall\alpha.\rho\| & = & \|\rho\|
\end{array}$$

Fig. 6. Termination Condition

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

The translation follows the type system presented in Section 3. The additional machinery that is necessary (on top of the type system) corresponds to the grayed parts of Figures 1, 2 and 3.

4.1. Type-Directed Translation

Figure 1 presents the translation rules that convert $\lambda_?$ expressions into ones of System F. The gray parts of the figure essentially extend the type system with the necessary information for the translation.

The syntax of System F is as follows:

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

With respect to the type system the type environments Γ remain the same. However, implicit environments Δ need to be extended with evidence information for the translation, thus becoming *translation* environments:

$$\text{Translation Environments} \quad \Delta \ ::= \ \epsilon \mid \Delta, \rho \rightsquigarrow E$$

The main translation judgment, which adapts the typing judgment, is

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

This judgment states that the translation of $\lambda_?$ expression e with type ρ is System F expression E, with respect to type environment Γ and translation environment Δ. The translation environment Δ relates each rule type in the earlier implicit environment to a System F variable x; this variable serves as explicit value-level evidence for the implicit rule. Lookup in the translation environment is defined similarly to lookup in the implicit environment, except that the lookup now returns a pair of a rule type and an evidence variable.

The function $|\cdot|$ takes $\lambda_?$ types ρ to System F types T:

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

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

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

$$\Delta \vdash_r \rho \rightsquigarrow E$$

which is shown, in deterministic form, in Figure 3. The translation of resolution basically extends the type-checking process with simultaenously building System F evidence terms. The mechanism that builds evidence dualizes the process of peeling off abstractions and universal quantifiers. The rule (`R-IAbs`) wraps a lambda binder with a fresh variable x around a System F expression E, which is generated from the resolution for the head of the rule (ρ_2). The rule (`R-TAbs`) wraps a type lambda binder around the System F expression resulting from the resolution of ρ. For simple types (`R-Simp`), evidence E_1 is retrieved from matching τ and then used in the resolution process of the simple type τ with rule $\vdash_\downarrow$:

$$\Delta; \rho \leadsto E_1 \vdash_\downarrow \tau \leadsto E_2$$

The rules (`I-IAbs`) and (`I-TAbs`) are the most interesting in the translation. In rule (`I-IAbs`) we need evidence for ρ_2 to build evidence for τ, but all we have is evidence E_1 for $\rho_1 \Rightarrow \rho_2$. However, we can construct evidence for ρ_2, by generating evidence E_2 for ρ_1 and then simply applying E_1 to E_2. Rule (`I-TAbs`) is similar: we need evidence for ρ, but all we have is evidence E_1 for $\forall\alpha.\rho$. To create evidence for ρ we can substitute α by some type ρ' in ρ and generate a term which is the type application of E_1 to $|\rho'|$.

Finally, matching ($\Delta\langle\tau\rangle$) also needs to be extended with evidence generation, but this extension is straightforward.

THEOREM 4.1 (TYPE-PRESERVING TRANSLATION). *Let e be a $\lambda_?$ expression, ρ be a type and E be a System F expression. If $\epsilon \mid \epsilon \vdash e : \rho \leadsto E$, then $\epsilon \vdash E : |\rho|$.*

PROOF. (Sketch) We first prove[3] the more general lemma "if $\Gamma \mid \Delta \vdash e : \rho \leadsto E$, then $|\Gamma|, |\Delta| \vdash E : |\rho|$" by induction on the derivation of translation. Then, the theorem trivially follows. □

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

LEMMA 4.2 (TYPE-PRESERVING RESOLUTION). *Let Δ be an implicit environment, ρ be a type and E be a System F expression. If $\Delta \vdash_r \rho \leadsto E$, then $|\Delta| \vdash E : |\rho|$.*

Moreover, we can express two key properties of Figure 3's definition of resolution in terms of the generated evidence.

LEMMA 4.3 (DETERMINACY). *The generated evidence of resolution is uniquely determined.*

$$\forall \Delta, \rho, E_1, E_2 : \qquad \Delta \vdash_r \rho \leadsto E_1 \;\wedge\; \Delta \vdash_r \rho \leadsto E_2 \quad \Rightarrow \quad E_1 = E_2$$

LEMMA 4.4 (SOUNDNESS). *Figure 3's definition of resolution (here denoted $\vdash_r^3$) is sound (but incomplete) with respect to Figure 2's definition (here denoted $\vdash_r^2$).*

$$\forall \Delta, \rho, E : \qquad \Delta \vdash_r^3 \rho \leadsto E \quad \Rightarrow \quad \Delta \vdash_r^2 \rho \leadsto E$$

4.2. Evidence Generation in the Algorithm

The evidence generation in Figure 4 is largely similar to that in the deterministic specification of resolution in Figure 3.

The main difference, and complication, is due to the fact that the evidence for type instantiation and recursive resolution is needed before these operations actually take place, as the algorithm has to postpone them. For this reason, the algorithm first produces placeholders that are later substituted for the actual evidence.

[3]in the technical report

The central relation is $\rho; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E'$. It captures the matching instantiation of context type ρ against simple type τ. The input evidence for ρ is E, and the output evidence for the instantiation is E'. The accumulating parameters $\bar{\alpha}$ and $\bar{\rho}$ denote that the instantiation of type variables $\bar{\alpha}$ and the recursive resolution of $\bar{\rho}$ have been postponed. We use the $\bar{\alpha}$ themselves as convenient placeholders for the instantiating types, and we use the synthetic $\bar{\omega}$ as placeholders for the evidence of the $\bar{\rho}$. The rules (MTC-Abs) and (MTC-Abs) introduce these two kinds of placeholders in the evidence. The former kind, $\bar{\alpha}$, are substituted in rule (MTC-Simp) where the actual type instantiatons $\bar{\theta}$ are computed. The latter kind, $\bar{\omega}$, are substituted later in rule (Alg-Simp) where the recursive resolutions take place.
Now we can state the correctness of the algorithm.

THEOREM 4.5 (PARTIAL CORRECTNESS). *Let Δ be an implicit environment, ρ be a type and E be a System F expression. Assume that $\epsilon \vdash_{unamb} \rho$ and also $\forall \rho_i \in \Delta : \epsilon \vdash_{unamb} \rho_i$. Then $\Delta \vdash_r \rho \rightsquigarrow E$ if and only if $\Delta \vdash_{alg} \rho \rightsquigarrow E$, provided that the algorithm terminates.*

4.3. Dynamic Semantics

Finally, we define the dynamic semantics of $\lambda_?$ as the composition of the type-directed translation and System F's dynamic semantics. Following Siek's notation [Siek and Lumsdaine 2005a], this dynamic semantics is:

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

with $\rightarrow^*$ the reflexive, transitive closure of System F's standard single-step call-by-value reduction relation (see [Pierce 2002, Chapter 23]).
Now we can state the conventional type safety theorem for $\lambda_?$:

THEOREM 4.6 (TYPE SAFETY). *If $\epsilon \mid \epsilon \vdash e : \rho$, then $eval(e) = V$ for some System F value V.*

The proof follows trivially from Theorem 4.1.

5. SOURCE LANGUAGES AND IMPLICIT INSTANTIATION

Languages like Haskell and Scala provide a lot more programmer convenience than $\lambda_?$ (which is a low level core language) because of higher-level GP constructs, interfaces and implicit instantiation. This section illustrates how to build a simple source language on top of $\lambda_?$ to add the expected convenience. We should note that unlike Haskell this language supports local and nested scoping, and unlike both Haskell and Scala it supports higher-order rules. We present the type-directed translation from the source to $\lambda_?$.

5.1. Type-directed Translation to $\lambda_?$

The full syntax of the source language is presented in Figure 8. Its use is illustrated in the program of Figure 7, which comprises an encoding of Haskell's equality type class Eq. The example shows that the source language features a simple type of interface $I\ \bar{T}$ (basically records), which are used to encode simple forms of type classes. Note that we follow Haskell's conventions for records: field names u are unique and they are modelled as regular functions taking a record as the first argument. So a field u with type T in an interface declaration $I\ \bar{\alpha}$ actually has type $\forall \bar{\alpha}.\epsilon \Rightarrow I\ \bar{\alpha} \rightarrow T$. The language also has other conventional programming constructs (such as let expressions, lambdas and primitive types).

$$
\begin{array}{l}
\textbf{interface}\ \mathit{Eq}\ \alpha = \{\mathit{eq} : \alpha \to \alpha \to \mathit{Bool}\} \\
\textbf{let}\ (\equiv) : \forall \alpha.\ \{\mathit{Eq}\ \alpha\} \Rightarrow \alpha \to \alpha \to \mathit{Bool} = \mathit{eq}\ ?\ \textbf{in} \\
\textbf{let}\ \mathit{eqInt_1} : \mathit{Eq}\ \mathit{Int} = \mathit{Eq}\ \{\mathit{eq} = \mathit{primEqInt}\}\ \textbf{in} \\
\textbf{let}\ \mathit{eqInt_2} : \mathit{Eq}\ \mathit{Int} = \mathit{Eq}\ \{\mathit{eq} = \lambda x\ y.\mathit{isEven}\ x \wedge \mathit{isEven}\ y\}\ \textbf{in} \\
\textbf{let}\ \mathit{eqBool} : \mathit{Eq}\ \mathit{Bool} = \mathit{Eq}\ \{\mathit{eq} = \mathit{primEqBool}\}\ \textbf{in} \\
\textbf{let}\ \mathit{eqPair} : \forall \alpha\ \beta.\ \{\mathit{Eq}\ \alpha, \mathit{Eq}\ \beta\} \Rightarrow \mathit{Eq}\ (\alpha, \beta) = \\
\qquad \mathit{Eq}\ \{\mathit{eq} = \lambda x\ y.\mathit{fst}\ x \equiv \mathit{fst}\ y \wedge \mathit{snd}\ x \equiv \mathit{snd}\ y\}\ \textbf{in} \\
\textbf{let}\ p_1 : (\mathit{Int}, \mathit{Bool}) = (4, \mathit{True})\ \textbf{in} \\
\textbf{let}\ p_2 : (\mathit{Int}, \mathit{Bool}) = (8, \mathit{True})\ \textbf{in} \\
\textbf{implicit}\ \{\mathit{eqInt_1}, \mathit{eqBool}, \mathit{eqPair}\}\ \textbf{in} \\
(p_1 \equiv p_2, \textbf{implicit}\ \{\mathit{eqInt_2}\}\ \textbf{in}\ p_1 \equiv p_2)
\end{array}
$$

Fig. 7. Encoding the Equality Type Class

Interface Declarations

$\textbf{interface}\ I\ \bar{\alpha}\ =\ \overline{u : T}$

Types

$$
\begin{array}{lll}
T & ::= \alpha & \text{Type Variables} \\
 & \mid I\ \bar{T} & \text{Interface Type} \\
 & \mid T \to T & \text{Function} \\
\sigma & ::= \forall \overline{\alpha}.\ \overline{\sigma} \Rightarrow T & \text{Rule Type}
\end{array}
$$

Expressions

$$
\begin{array}{lll}
E & ::= x & \text{Lambda Variable} \\
 & \mid \lambda x.E & \text{Abstraction} \\
 & \mid E_1\ E_2 & \text{Application} \\
 & \mid u & \text{Let Variable} \\
 & \mid \textbf{let}\ u : \sigma\ =\ E_1\ \textbf{in}\ E_2 & \text{Let} \\
 & \mid \textbf{implicit}\ \overline{u}\ \textbf{in}\ E_2 & \text{Implicit Scoping} \\
 & \mid\ ? & \text{Implicit Lookup} \\
 & \mid I\ \overline{u = E} & \text{Interface Implementation}
\end{array}
$$

Fig. 8. Syntax of Source Language

Unlike the core language, we strongly differentiate between simple types T and type schemes σ in order to facilitate type inference. The source language also distinguishes simply typed variables x from let-bound variables u with polymorphic type σ.

Figure 9 presents the type-directed translation $G \vdash E : T \rightsquigarrow e$ of source language expressions E of type T to core expressions e, with respect to type environment G. The type environment collects both simple and polymorphic variable typings. The connection between source types T and σ on the one hand and core types τ and ρ on the other hand is captured in the auxiliary function $[\![\cdot]\!]$. We should also remark that while the type system in Figure 9 (the non-grey parts) is sufficient to allow the type-directed translation, it is incomplete as it does not check whether queries can be resolved or not. Performing such checks at the level of the source language is possible, but requires repeating some of the infrastructure in Figures 1 and 3. For simplicity reasons, and since we are mainly interested in the type-directed translation, we have avoided that extra machinery here. For the translation of records, we assume that $\lambda_?$ is extended likewise with records. Like in $\lambda_?$, we also assume the existence of primitive types like integers, booleans and pairs for the sake of examples.

$$\text{Type Environments } G ::= \epsilon \mid G, u : \sigma \mid G, x : T$$

$$\boxed{G \vdash E : T \leadsto e}$$

(Ty-Var)
$$\frac{G(x) = T}{G \vdash x : T \leadsto x}$$

(Ty-Abs)
$$\frac{G, x : T_1 \vdash E \leadsto e}{G \vdash \lambda x.E : T_1 \to T_2 \leadsto \lambda x : \llbracket T_1 \rrbracket . e}$$

(Ty-App)
$$\frac{\begin{array}{c} G \vdash E_1 : T_1 \to T_2 \leadsto e_1 \\ G \vdash E_2 : T_1 \leadsto e_2 \end{array}}{G \vdash E_1\ E_2 : T_2 \leadsto e_1\, e_2}$$

(TyLVar)
$$\frac{\begin{array}{c} G(u) = \forall \overline{\alpha}.\ \overline{\sigma} \Rightarrow T_2 \\ \theta = [\overline{\alpha} \mapsto \overline{T}] \quad T_1 = \theta T_2 \\ q_i =\ ?\llbracket \theta \sigma_i \rrbracket \quad (\forall \sigma_i \in \overline{\sigma}) \end{array}}{G \vdash u : T_1 \leadsto u\ \overline{\llbracket T \rrbracket}\ \overline{\textbf{with}\ q}}$$

(TyLet)
$$\frac{\begin{array}{c} \sigma_1 = \forall \overline{\alpha}.\overline{\sigma_2} \Rightarrow T_1 \\ G \vdash E_1 : T_1 \leadsto e_1 \\ G, u : \sigma_1 \vdash E_2 : T_2 \leadsto e_2 \end{array}}{G \vdash\ \textbf{let}\ u : \sigma_1 = E_1\ \textbf{in}\ E_2 : T_2 \leadsto (\lambda u : \llbracket \sigma_1 \rrbracket . e_2)\ (\overline{\Lambda \alpha.}\ \overline{\lambda_? \sigma_2 .} e_1)}$$

(TyImp)
$$\frac{\begin{array}{c} G \vdash E : T \leadsto e \\ G(u_i) = \sigma_i \quad (\forall u_i \in \overline{u}) \end{array}}{G \vdash\ \textbf{implicit}\ \overline{u}\ \textbf{in}\ E : T \leadsto (\overline{\lambda_? \llbracket \sigma \rrbracket .} e)\ \overline{\textbf{with}\ u}}$$

(TyIVar)
$$G \vdash ? : T \leadsto\ ?\llbracket T \rrbracket$$

(TyRec)
$$\frac{\forall i : \begin{cases} G(u_i) = \forall \bar{\alpha}.\epsilon \Rightarrow I\ \bar{\alpha} \to T_i \\ G \vdash E_i : \theta T_i \leadsto e \quad \theta = [\bar{\alpha} \mapsto \bar{T}] \end{cases}}{G \vdash I\ \overline{u = E} : I\ \bar{T} \leadsto I\ \overline{u = e}}$$

$$\begin{array}{rcl} \llbracket \alpha \rrbracket & = & \alpha \\ \llbracket T_1 \to T_2 \rrbracket & = & \llbracket T_1 \rrbracket \to \llbracket T_2 \rrbracket \\ \llbracket I\ \bar{T} \rrbracket & = & I\ \llbracket \bar{T} \rrbracket \\ \llbracket \forall \overline{\alpha}.\overline{\sigma} \Rightarrow T \rrbracket & = & \forall \llbracket \bar{\alpha} \rrbracket . \llbracket \overline{\sigma} \rrbracket \Rightarrow \llbracket T \rrbracket \end{array}$$

Fig. 9. Type-directed Encoding of Source Language in $\lambda_?$

let *and* **let***-bound variables.* The rule (`TyLet`) in Figure 9 shows the type-directed translation for let expressions. This translation binds the variable u using a regular lambda abstraction in an expression e_2, which is the result of the translation of the body of the let construct (E_2). This lambda abstraction is then applied to an expression which has type $\forall\overline{\alpha}.\overline{\sigma_2} \Rightarrow T_1$. When both $\overline{\alpha}$ and $\overline{\sigma_2}$ are empty that expression is simply an expression e_1, resulting from the translation of the expression E_1. Otherwise, for each α in $\overline{\alpha}$ and for each σ in $\overline{\sigma_2}$, corresponding type and rule binders are created around the expression e_1.

The source language provides convenience to the user by inferring type arguments and implicit values automatically. This inference happens in rule (`TyLVar`), i.e., the use of let-bound variables. That rule recovers the type scheme of variable u from the environment G. Then it instantiates the type scheme and fires the necessary queries to resolve the context.

Queries. The source language also includes a query operator (?). Unlike $\lambda_?$ this query operator does not explicitly state the type; that information is provided implicitly through type inference. For example, instead of using $p_1 \equiv p_2$ in Figure 7, we could have directly used the field eq as follows:

$$eq\ ?\ p_1\ p_2$$

When used in this way, the query acts like a placeholder for a value. The type of the placeholder value can be determined using type-inference. Once the type system knows the type of the placeholder it automatically synthesizes a value of the of the right type from the implicit context.

The translation of source language queries, given by the rule (`TyIVar`), is trivial. To simplify type-inference, the query is limited to types, and does not support partial resolution.

Implicit scoping. The **implicit** construct is the core scoping construct of the source language. It is first used in our example to make *eqInt₁*, *eqBool* and *eqPair* available for resolution at the expression

$$(p_1 \equiv p_2, \textbf{implicit}\ \ \{\mathit{eqInt_2}\}\ \textbf{in}\ p_1 \equiv p_2)$$

Within this expression there is a second occurrence of **implicit**, which introduces an overlapping rule ($\mathit{eqInt_2}$) that takes priority over $\mathit{eqInt_1}$ for the subexpression $p_1 \equiv p_2$.

The translation rule (`TyImp`) of **implicit** into $\lambda_?$ also exploits type-information to avoid redundant type annotations. It is not necessary to annotate the let-bound variables used in the rule set $\overline{u}$ since that information is recovered from the environment G.

Higher-order rules and implicit instantiation for any type. The following example illustrates higher-order rules and implicit instantiation working for any type in the source language.

$$
\begin{array}{l}
\textbf{let}\ \mathit{show} : \forall\alpha.\ \{\alpha \to \mathit{String}\} \Rightarrow \alpha \to \mathit{String} =\ ?\ \ \textbf{in} \\
\textbf{let}\ \mathit{showInt} : \mathit{Int} \to \mathit{String} = \ldots\ \textbf{in} \\
\textbf{let}\ \mathit{comma} : \forall\alpha.\ \{\alpha \to \mathit{String}\} \Rightarrow [\alpha] \to \mathit{String} = \ldots\ \textbf{in} \\
\textbf{let}\ \mathit{space} : \forall\alpha.\ \{\alpha \to \mathit{String}\} \Rightarrow [\alpha] \to \mathit{String} = \ldots\ \textbf{in} \\
\textbf{let}\ o :\ \{\mathit{Int} \to \mathit{String},\ \{\mathit{Int} \to \mathit{String}\} \Rightarrow [\mathit{Int}] \to \mathit{String}\} \\
\qquad\quad \Rightarrow \mathit{String} = \mathit{show}\,[1,2,3]\ \textbf{in} \\
\textbf{implicit}\ \mathit{showInt}\ \textbf{in} \\
(\textbf{implicit}\ \mathit{comma}\ \textbf{in}\ o, \textbf{implicit}\ \mathit{space}\ \textbf{in}\ o)
\end{array}
$$

For brevity, we have omitted the implementations of *showInt*, *comma* and *space*; but *showInt* renders an *Int* as a *String* in the conventional way, while *comma* and

$space$ provide two ways for rendering lists. Evaluation of the expression yields `("1,2,3","1 2 3")`. Thanks to the implicit rule parameters, the contexts of the two calls to o control how the lists are rendered.

This example differs from that in Figure 7 in that instead of using a *nominal* interface type like Eq, it uses standard functions to model a simple concept for pretty printing values. The use of functions as implicit values is similar to *structural* matching of concepts, since only the type of the function matters for resolution.

5.2. Extensions

The goal of our work is to present a minimal and general framework for implicits. As such we have avoided making assumptions about extensions that would be useful for some languages, but not others.

In this section we briefly discuss some extensions that would be useful in the context of particular languages and the implications that they would have in our framework.

Full-blown Concepts. The most noticeable feature that was not discussed is a full-blown notion of concepts. One reason not to commit to a particular notion of concepts is that there is no general agreement on what the right notion of concepts is. For example, following Haskell type classes, the C++0x concept proposal [Gregor et al. 2006] is based on a *nominal* approach with *explicit* concept refinement, while Stroustrup favors a *structural* approach with *implicit* concept refinement because that would be more familiar to C++ programmers [Stroustrup 2009]. Moreover, various other proposals for GP mechanisms have their own notion of interface: Scala uses standard OO hierarchies; Dreyer et al. use ML-modules [Dreyer et al. 2007]; and in dependently typed systems (dependent) record types are used [Sozeau and Oury 2008; Devriese and Piessens 2011].

An advantage of $\lambda_?$ is that no particular notion of interface is imposed on source language designers. Instead, language designers are free to use the one they prefer. In our source language, for simplicity, we opted to add a very simple (and limited) type of interface. But existing language designs [Oliveira et al. 2010; Dreyer et al. 2007; Sozeau and Oury 2008; Devriese and Piessens 2011] offer evidence that more sophisticated types of interfaces, including some form of refinement or associated types, can be built on top of $\lambda_?$.

Type Constructor Polymorphism and Higher-order Rules. Type constructor polymorphism is an advanced, but highly powerful GP feature available in Haskell and Scala, among others. It allows abstracting container types like $List$ and $Tree$ with a type variable f; and applying the abstracted container type to different element types, e.g., $f\ Int$ and $f\ Bool$.

This type constructor polymorphism leads to a need for higher-order rules: rules for containers of elements that depend on rules for the elements. The instance for showing values of type $Perfect\ f\ \alpha$ in Section 1, is a typical example of this need.

Extending $\lambda_?$ with type constructor polymorphism is not hard. Basically, we need to add a kind system and move from a System F-like language to a System F_ω-like language.

Subtyping. Languages like Scala or C++ have subtyping. Subtyping would require significant adaptations to $\lambda_?$. Essentially, instead of targetting System F, we would have to target a version of System F with subtyping. In addition, the notion of matching in the lookup function $\Delta\langle\tau\rangle$ would have to be adjusted. While subtyping is a useful feature, some language designs do not support it because it makes the system more complex and interferes with type-inference.

Type-inference. Languages without subtyping (like Haskell or ML) make it easier to support better type-inference. Since we do not use subtyping, it is possible to improve support for type-inference in our source language. In particular, we currently require a type annotation for let expressions, but it should be possible to make that annotation optional, by building on existing work for the GHC Haskell compiler [Schrijvers et al. 2009; Vytiniotis et al. 2011].

6. RELATED WORK

This section discusses related work.

The goal of our work is to formalize a core language ($\lambda_?$) with the essential features of a type-directed implicit parameter passing mechanism equipped with recursive resolution. There have been several other proposals for core calculi which contain some form of implicit parameter passing and/or recursive resolution. However none of these allows for both implicit parameter passing for any types of values and recursive resolution. Furthermore, although there have been some informal proposals for higher-order rules, the implicit calculus is the first system providing a full formalization of this feature.

6.1. Type Classes

Several core calculi and refinements have been proposed in the context of type-classes. As already discussed in detail in Section 1, there are a number of design choices that (Haskell-style) type classes take that are different from the implicit calculus. Most prominently, type classes make a strong differentiation between types and type classes, and they use global scoping instead of local scoping for the rule environment. These design choices can be traced back to Wadler and Blott's [1989] original paper on type classes. In that paper the authors argue that the inspiration for type classes came from Standard ML's *eqtype variables*. Eqtype variables were used to provide overloaded equality, by allowing type variables which range only over types that admit equality. Wadler and Blott generalized that idea by allowing arbitrary predicates over types to be defined as type classes. This lead to type classes being viewed as *predicates over types* rather than types, and languages with types classes making a syntactic distinction between type classes and types. Implementations of type classes (that is, type class instances) are implicitly passed, whereas regular values implementing types are explicitly passed.

The reason for global scoping is also motivated by Wadler and Blott. They wanted to extend Hindley-Milner type-inference [Hindley 1969; Milner 1978; Damas and Milner 1982] and discovered that local instances resulted in the loss of principal types. In both the implicit calculus and our source language there are sufficient type annotations that the problem does not arise. However, the problem would indeed arise in the source language if there were no top-level type annotations for a program. For example, if we could write:

```
implicit eqInt : Eq Int in
  implicit eqChar : Eq Char in
    eq
```

(assuming the existence of values *eqInt* and *eqChar*) then, without further annotations, the meaning of this program would be ambiguous. In a language that strives for Hindley-Milner type-inference and principal types, this kind of ambiguity would be viewed as a serious problem. However, there are many languages with type-class like mechanisms (including Scala, Coq, Agda and Isabelle) that have more modest goals in terms of type-inference. In these languages there are usually enough type annotations that such ambiguity is avoided, and there is indeed added expressive power because type annotations drive the resolution process.

There are three possible behaviors here, which can all be accounted for by our source language given a suitable type annotation for the program:

— With the type annotation $Int \to Int \to Bool$ resolution picks the $eqInt$ instance.
— With the type annotation $Char \to Char \to Bool$ would pick $eqChar$ instead.
— With the type annotation $\forall a.Eq\ a \Rightarrow a \to a \to Bool$, a third implicit rule of type $Eq\ a$ would have to be available in the implicit environment and resolution would pick that rule instead.

In other words, type annotations allow the user to control the resolution process according to the intended semantics.

There is a wide range of work that builds on the original type classes proposal [Wadler and Blott 1989]. Jones's work on *qualified types* [Jones 1995] provides a particularly elegant framework that captures type classes and other forms of predicates on types. Like type classes, qualified types too make a strong distinction between types and predicates over types, and scoping is global. There have been some proposals for addressing the limitations that arise from global scoping [Kahl and Scheffczyk 2001; Dijkstra and Swierstra 2005]. However in those designs, type classes are still second-class and resolution only works for type classes. The GHC Haskell compiler supports overlapping instances [Jones et al. 1997], that live in the same global scope. This allows some relief for the lack of local scoping. A lot of recent work on type classes is focused on increasingly more powerful "type class" interfaces. *Functional dependencies* [Jones 2000], *associated types* [Chakravarty et al. 2005b; Chakravarty et al. 2005a] and *type families* [Schrijvers et al. 2008] are all examples of this trend. This line of work is orthogonal to our own.

Our calculus has a different approach to overlapping compared to *instance chains* [Morris and Jones 2010]. With instance chains the programmer imposes an order on a set of overlapping type class instances. All instance chains for a type class have global scope and are expected not to overlap. This makes the scope of overlapping closed within a chain. In our calculus, we make our local scope closed, thus overlap only happens within one nested scope. More recently, there has been a proposal for *closed type families with overlapping equations* [Eisenberg et al. 2014]. This proposal allows the declaration of a type family and a (closed) set of instances. After this declaration no more instances can be added. In contrast our notion of scoping is closed at a particular resolution point, but the scopes can still be extended in other resolution points.

6.2. Local Scoping

Implicit parameters [Lewis et al. 2000] are a proposal for a name-based implicit parameter passing mechanism with local scoping. Lewis et al. formalized a small core language with the mechanism and there is also a GHC Haskell implementation. Implicit parameters allow *named* arguments to be passed implicitly, and these arguments can be of any type. Using implicit parameters[4] we could for example, write the following program:

$$\begin{array}{l}\mathbf{data}\ EqD\ a = EqD\ \ \{eq' :: a \to a \to Bool\} \\ eq :: (?eqD :: EqD\ a) \Rightarrow a \to a \to Bool \\ eq = eq'\ (?eqD)\end{array}$$

[4]Here we use the implementation available in the GHC Haskell compiler. The reader should note that the syntax used in GHC differs from the syntax used in the original paper [Lewis et al. 2000].

$$
\begin{array}{ll}
\mathit{eqInt} \; :: \mathit{EqD}\ \mathit{Int} & \text{-- Definition omitted} \\
\mathit{eqMaybe} :: (?\mathit{eqD} :: \mathit{EqD}\ a) \Rightarrow \mathit{EqD}\ (\mathit{Maybe}\ a) & \text{-- Definition omitted} \\
p_1 = \mathbf{let}\ ?\mathit{eqD} = \mathit{eqInt}\ \mathbf{in}\ \mathit{eq}\ 3\ 4 &
\end{array}
$$

Here the intention is to model something similar to Haskell's Eq type class or the code in Figure 7. The definitions eqInt and $\mathit{eqMaybe}$ play the role of the rules. The implicit parameters are named, so that later they can be resolved from a local scope that binds named arguments. This resolution process is illustrated in program p_1: the eqInt dictionary is brought into the local scope (bound to the variable eqD) and used in the expression $\mathit{eq}\ 3\ 4$. In this case, the use of implicit parameters does not look too different from the implicit calculus. However, implicit parameters do not support recursive resolution, so for most use-cases of type-classes they require composing rules manually, instead of relying on the recursive resolution mechanism to do this automatically. For example, the program:

$$
\begin{array}{l}
p_2 = \mathbf{let}\ ?\mathit{eqD} = (\mathbf{let}\ ?\mathit{eqD} = (\mathbf{let}\ ?\mathit{eqD} = \mathit{eqInt}\ \mathbf{in}\ \mathit{eqMaybe})\ \mathbf{in}\ \mathit{eqMaybe}) \\
\qquad \mathbf{in}\ \mathit{eq}\ (\mathit{Just}\ (\mathit{Just}\ 3))\ (\mathit{Just}\ (\mathit{Just}\ 4))
\end{array}
$$

illustrates a situation where we would like to compare two expressions $(\mathit{Just}\ (\mathit{Just}\ 3))$ and $(\mathit{Just}\ (\mathit{Just}\ 4))$. This can be done by using the rule $\mathit{eqMaybe}$ twice and using the rule eqInt once. While in the implicit calculus recursive resolution would automatically compose these rules, with implicit parameters the rules have to be manually composed.

System F^G. System F^G [Siek and Lumsdaine 2005b] also offers an implicit parameter passing mechanism with local scoping, which is used for concept-based generic programming. Instead of a name-based approach, a type-directed approach is used for passing implicit parameters. This is closer to the implicit calculus. Program p_1 could be modelled as follows in System F^G:

$$
\begin{array}{l}
\mathbf{concept}\ \mathit{Eq}\langle t\rangle\ \{ \\
\quad \mathit{eq} : \mathbf{fn}\ (t, t) \rightarrow \mathit{Bool}; \\
\}\ \mathbf{in} \\
\mathbf{let}\ p_1 = \mathbf{model}\ \mathit{Eq}\langle \mathit{int}\rangle\ \{\ldots\}\ \mathbf{in}\ \mathit{eq}\,[\mathit{int}]\ 3\ 4
\end{array}
$$

The **concept** declaration provides the interface for $\mathit{Eq}\langle t\rangle$ concepts, whereas the **model** declaration provides the corresponding implementation of the concept. A difference to the implicit calculus is that declaring a model automatically adds that model to the implicit environment. In program p_1 we must both provide the model and add it to the implicit environment in a single step. In the implicit calculus, these two aspects are decoupled. A more important difference to the implicit calculus is that, like type classes, there is a strong differentiation between types and concepts in System F^G: concepts cannot be used as types; and types cannot be used as concepts. As a consequence, models implementing concepts can only be passed as implicit parameters, and regular values can only be passed as explicit parameters.

In contrast to $\lambda_?$, System F^G has both a notion of concepts and implicit instantiation of concepts[5] built-in. This has the advantage that language designers can just reuse that infrastructure, instead of having to implement it (as we did in Section 5). The language G [Siek and Lumsdaine 2011] is based on System F^G and it makes good use of these built-in mechanisms. However, System F^G also imposes important design

[5]Note that instantiation of type variables is still explicit.

```
trait $A$ {
  implicit def $id[a] : a \Rightarrow a = x \Rightarrow x$
  def $?[a]$ (implicit $x : a) = x$
}
object $B$ extends $A$ {
  implicit def $succ : Int \Rightarrow Int = x \Rightarrow x + 1$
  val $v_1 = (?[Int \Rightarrow Int]).apply\ (3) // evaluates\ to\ 4$
  val $v_2 = (?[Char \Rightarrow Char]).apply\ (\texttt{'a'}) // evaluates\ to$ 'a'
}
```

Fig. 10. Nested Scoping with Overlapping Rules in Scala

choices, such the use of a notion of concepts that is built-in to the calculus. In contrast $\lambda_?$ offers a freedom of choice (see also the discussion in Section 5.2).

Finally System F^G only formalizes a very simple type of resolution, which does not support recursive resolution. To create program p_2 a model:

model $Eq\langle Maybe\,[Maybe\,[Int]]\rangle\ \{\ldots\}$

that manually composes rules must first be created in System F^G.

Modular type classes [Dreyer et al. 2007] are a language design that uses ML-modules to model type classes. The main novelty of this design is that, in addition to explicit instantiation of modules, implicit instantiation is also supported. In this design local scoping is allowed. However, unlike $\lambda_?$ (and also System F^G and implicit parameters) the local scopes cannot be nested. Furthermore, implicit instantiation is limited to modules (that is other regular values cannot be implicitly passed and automatically resolved).

6.3. Scala Implicits

The main inspiration for our work comes from Scala implicits [Oliveira et al. 2010; Odersky 2010]. Like our work Scala implicits allow implicit parameters of any types of values and recursive resolution is supported. Prior to our work, there was no small core calculus or any other form of formalization for this style of implicit parameters. The main objective of our work was the formalize the essence of the ideas behind Scala implicits. What we promote on this work is that the key idea of implicits is a type-directed implicit parameter passing mechanism that works for all types of values and supports local scoping with recursive resolution. However, we should note that our goal is not to have a faithful formalization of Scala implicits, since many other orthogonal aspects of the mechanism are tailored for the particularities of the Scala language.

Therefore there are noteworthy differences between $\lambda_?$ and Scala implicits. In contrast to $\lambda_?$, Scala has subtyping. We do not think that subtyping is essential, and it complicates the formalization: as discussed in Section 5.2 subtyping would require considerable adaptations to our calculus. Therefore we have omitted subtyping here. Although Scala also provides local and nested scoping, nested scoping can only happen through subclassing and the rules for resolution in the presence of overlapping instances are quite ad-hoc. Figure 10 illustrates the idea of nested scoping in Scala. Note that Scala implicits do not natively support the query expressions (?), but we can easily encode this functionality. In Scala each trait/class declaration introduces a scope and trait/class extension allows extending that scope. Thus in trait A an implicit rule for type $\forall a.a \rightarrow a$ is introduced. In object B the scope of A is extended and a new, overlapping, rule ($succ$) of type $Int \to Int$ is added[6]. Like the implicit calculus the later rule takes priority when a query of type $Int \to Int$ is required. However, in Scala the rules of scoping are more complicated than in the implicit calculus. If more than one implicit value has the right type, there must be a single most specific one according to an ordering. Informally this ordering states that a rule A is more specific than a rule B if the relative weight of A over B is greater than the relative weight of B over A. The relative weight is a score between 0 and 2, where A gets a point over B for being as specific as B, and another if it is defined in a class (or in its companion object) which is derived from the class that defines B, or whose companion object defines B. Roughly, a method is as specific as a member that is applicable to the same arguments, a polymorphic method is compared to another member after stripping its type parameters, and a non-method member is as specific as a method that takes arguments or type parameters. In other words Scala's scoring system attempts to account for both nested scoping through subclassing and the most specific type, whereas in the implicit calculus only the lexical scope is considered. Finally, Scala has no (first-class) rule abstractions and consequently no support for higher-order rules. Rather, implicit arguments can only be used in definitions.

6.4. Type Classes, Theorem Proving and Dependent Types

A number of dependently typed languages also include several mechanisms inspired by type classes. Although such mechanisms have been implemented and they are actively used, there is little work on formalization.

Isabelle Type Classes. The first type-class mechanism in a theorem prover was in Isabelle [Haftmann. and Wenzel 2006]. The mechanism was largely influenced by Haskell type classes and shares many of the same design choices. The introduction of *axiomatic type classes* [Wenzel and Mnchen 2000] showed how theorem proving can benefit from type classes to model not only the operations in type classes, but also the corresponding algebraic laws.

Coq's Canonical Structures And Type Classes. The Coq theorem prover has two mechanisms that allow modelling type-class like structures: *canonical structures* [Gonthier et al. 2011] and *type classes* [Sozeau and Oury 2008]. The two mechanisms have quite a bit of overlap in terms of functionality. In both mechanisms the idea is to use dependent records to model type-class-like structures, and pass instances of such records implicitly. Both mechanisms support recursive resolution to automatically build suitable records and they follow Haskell type classes model of global scoping. Furthermore, because Coq is dependently typed an additional feature of the two mechanisms is that they can also model *value classes* [Gonthier et al. 2011] (that is classes parametrized by values, rather than by types). This functionality is not available in the implicit calculus, due to the lack of dependent types. Another difference is that recursive resolution is allowed to backtrack in canonical structures and type classes, whereas the implicit calculus forbids this. The reason for forbidding backtracking in the implicit calculus (and also Haskell type classes) is justified by the use of the mechanism for programming purposes, and the need for users to easily predict which instances are used. In a theorem proving context, backtracking makes more sense since, due to *proof irrelevance*, which instances get picked in a proof is not so important, as long as the proof is completed.

A key difference to our work is that both canonical structures and Coq's type classes focus on the implementation of a concrete mechanism, whereas we focus on the formal-

[6]Note that introducing this rule directly in A would result in an ambiguity problem, since two overlapping rules are not allowed within the same trait/class.

ization of a general mechanism. Neither canonical structures nor Coq's type classes have been formally specified. It could be that a generalization of the implicit calculus with dependent types (and allowing backtracking) would be able to provide a suitable specification for these mechanisms. Generalizing the implicit calculus with Coq style dependent types poses considerable challenges, because computation can happen during type-checking.

Instance arguments [Devriese and Piessens 2011] are an Agda extension that is closely related to implicits. Like the implicit calculus, instance arguments use a special arrow for introducing implicit arguments. However, unlike most other mechanisms, implicit rules are not declared explicitly. Instead rules are drawn directly from the type-environment, and any previously defined declaration can be used as a rule. Furthermore resolution is limited in its expressive power, to avoid introducing a different computational model in Agda. This design differs significantly from $\lambda_?$, where resolution is very expressive and the scoping mechanisms allow explicit rule declarations.

6.5. Other Related Work

Resolution with Higher-order Rules. Resolution in $\lambda_?$ is significantly more expressive than in other systems. Notably resolution supports higher-order rule types and queries, as well as queries for polymorphic types. A closely related design sketch is that of higher-order predicates by Hinze and Peyton Jones [2001]; no other IP system has adopted a similar extension.

As Hinze and Peyton Jones show, higher-order predicates are specially important when dealing with types that involve type constructor polymorphism. In order to simplify presentation, our formalization of the implicit calculus does not include type constructor polymorphism.

Our work is the first to study the meta-theory of higher-order rules as part of a language. Hinze and Peyton Jones only list a system of inference rules, but do not study any of its properties.

Type Classes and Logic Programming. The connection between Haskell type classes and Prolog is folklore. Neubauer et. al. [2002] also explore the connection with *Functional Logic Programming* and consider different evaluation strategies to deal with overlapping rules. With *Constraint Handling Rules*, Stuckey and Sulzmann [2002] use *Constraint Logic Programming* to implement type classes.

7. CONCLUSION

Our main contribution is the development of the implicit calculus $\lambda_?$. This calculus isolates and formalizes the key ideas of Scala implicits and provides a simple model for language designers interested in developing similar mechanisms for their own languages. In addition, $\lambda_?$ supports higher-order rules and partial resolution, which add considerable expressiveness to the calculus.

Implicits provide an interesting alternative to conventional GP mechanisms like type classes or concepts. By decoupling resolution from a particular type of interfaces, implicits make resolution more powerful and general. Furthermore, this decoupling has other benefits too. For example, by modeling concept interfaces as conventional types, those interfaces can be abstracted as any other types, avoiding the issue of second class interfaces that arise with type classes or concepts.

Ultimately, all the expressiveness offered by $\lambda_?$ offers a wide-range of possibilities for new generic programming applications.

APPENDIX

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We are grateful to Ben Delaware, Derek Dreyer, Jeremy Gibbons, Scott Kilpatrick, eta Ziliani, the members of ROPAS and the anonymous reviewers for their comments and suggestions. This work was partially supported by Korea Ministry of Education, Science and Technology/Korea Science and Enginering Foundation's ERC grant R11-2008-007-01002-0, Brain Korea 21, Mid-career Research Program 2010-0022061, and by Singapore Ministry of Education research grant MOE2010-T2-2-073.

REFERENCES

Boost 2010. The Boost C++ libraries. http://www.boost.org/.

CAMARÃO, C. AND FIGUEIREDO, L. 1999. Type inference for overloading without restrictions, declarations or annotations. In *FLOPS*.

CHAKRAVARTY, M., KELLER, G., AND JONES, S. L. P. 2005a. Associated type synonyms. In *ICFP*.

CHAKRAVARTY, M., KELLER, G., JONES, S. L. P., AND MARLOW, S. 2005b. Associated types with class. In *POPL*.

DAMAS, L. AND MILNER, R. 1982. Principal type-schemes for functional programs. In *POPL*. 207–212.

DEVRIESE, D. AND PIESSENS, F. 2011. On the bright side of type classes: Instance arguments in agda. In *ICFP*.

DIJKSTRA, A. AND SWIERSTRA, S. D. 2005. Making implicit parameters explicit. Tech. rep., Utrecht University.

DOS REIS, G. AND STROUSTRUP, B. 2006. Specifying C++ concepts. In *POPL '06*. 295–308.

DREYER, D., HARPER, R., CHAKRAVARTY, M., AND KELLER, G. 2007. Modular type classes. In *POPL*.

EISENBERG, R. A., VYTINIOTIS, D., PEYTON JONES, S., AND WEIRICH, S. 2014. Closed type families with overlapping equations. In *POPL 2014: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, CA, USA.

GARCIA, R., JARVI, J., LUMSDAINE, A., SIEK, J., AND WILLCOCK, J. 2003. A comparative study of language support for generic programming. In *OOPSLA*.

GIBBONS, J. 2003. Patterns in datatype-generic programming. In *The Fun of Programming, Cornerstones in Computing*. Palgrave.

GONTHIER, G., ZILIANI, B., NANEVSKI, A., AND DREYER, D. 2011. How to make ad hoc proof automation less ad hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. 163–175.

GREGOR, D., JÄRVI, J., SIEK, J. G., STROUSTRUP, B., REIS, G. D., AND LUMSDAINE, A. 2006. Concepts: linguistic support for generic programming in c++. In *OOPSLA*.

HAFTMANN., F. AND WENZEL, M. 2006. Constructive type classes in Isabelle. In *TYPES*.

HINDLEY, J. R. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146*, 29–60.

HINZE, R. AND JONES, S. L. P. 2001. Derivable type classes. *Electronic Notes in Theoretical Computer Science 41,* 1, 5 – 35.

HUGHES, J. 1999. Restricted data types in Haskell. In *Haskell*.

JANSSON, P. AND JEURING, J. 1996. Polytypic programming. In *AFP*. Springer-Verlag.

JONES, M. P. 1995. Simplifying and improving qualified types. In *FPCA*.

JONES, M. P. 2000. Type classes with functional dependencies. In *ESOP*.

JONES, S. L. P., JONES, M. P., AND MEIJER, E. 1997. Type classes: exploring the design space. In *Haskell Workshop*.

KAHL, W. AND SCHEFFCZYK, J. 2001. Named instances for Haskell type classes. In *Haskell Workshop*.

KOWALSKI, R. 1974. Predicate logic as a programming language. In *Proceedings of IFIP Congress*.

KOWALSKI, R., DONALD, AND KUEHNER. 1971. Linear resolution with selection function. *Artificial Intelligence 2*.

LÄMMEL, R. AND JONES, S. L. P. 2005. Scrap your boilerplate with class: extensible generic functions. In *ICFP*.

LEWIS, J., LAUNCHBURY, J., MEIJER, E., AND SHIELDS, M. 2000. Implicit parameters: dynamic scoping with static types. In *POPL*.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17,* 3, 348–375.

MORRIS, J. G. AND JONES, M. P. 2010. Instance chains: type class programming without overlapping instances. In *ICFP*.

MUSSER, D. AND STEPANOV, A. 1988. Generic programming. In *Symbolic and algebraic computation: ISSAC 88*. Springer, 13–25.

MUSSER, D. R. AND SAINI, A. 1995. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc.

NEUBAUER, M., THIEMANN, P., GASBICHLER, M., AND SPERBER, M. 2002. Functional logic overloading. In *POPL*.

ODERSKY, M. 2006. Poor man's type classes. `http://lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf`.

ODERSKY, M. 2010. The Scala language specification, version 2.8.

OLIVEIRA, B. C. D. S. AND GIBBONS, J. 2010. Scala for generic programmers. *Journal of Functional Programming 20*.

OLIVEIRA, B. C. D. S., MOORS, A., AND ODERSKY, M. 2010. Type classes as objects and implicits. In *OOPSLA*.

OLIVEIRA, B. C. D. S., SCHRIJVERS, T., CHOI, W., LEE, W., AND YI, K. 2012. Extended report: The implicit calculus. `http://arxiv.org/abs/1203.4499`.

PIERCE, B. C. 2002. *Types and programming languages*. MIT Press, Cambridge, MA, USA.

RODRIGUEZ, A., JEURING, J., JANSSON, P., GERDES, A., KISELYOV, O., AND OLIVEIRA, B. C. D. S. 2008. Comparing libraries for generic programming in haskell. In *Haskell*.

SCHRIJVERS, T., JONES, S. L. P., CHAKRAVARTY, M., AND SULZMANN, M. 2008. Type checking with open type functions. In *ICFP*.

SCHRIJVERS, T., JONES, S. L. P., SULZMANN, M., AND VYTINIOTIS, D. 2009. Complete and decidable type inference for GADTs. In *ICFP*.

SIEK, J. 2011. The C++0x Concepts Effort. `http://ecee.colorado.edu/~siek/concepts_effort.pdf`.

SIEK, J. G. AND LUMSDAINE, A. 2005a. Essential language support for generic programming. In *PLDI*.

SIEK, J. G. AND LUMSDAINE, A. 2005b. Essential language support for generic programming. In *PLDI*.

SIEK, J. G. AND LUMSDAINE, A. 2011. A language for generic programming in the large. *Science of Computer Programming 76(5)*.

SOZEAU, M. AND OURY, N. 2008. First-class type classes. In *TPHOLs*.

STROUSTRUP, B. 2009. Simplifying the use of concepts. Tech. rep., Technical Report N2906, ISO/IEC JTC 1 SC22 WG21.

STUCKEY, P. J. AND SULZMANN, M. 2002. A theory of overloading. In *ICFP*.

SULZMANN, M., DUCK, G., JONES, S. L. P., AND STUCKEY, P. J. 2007. Understanding functional dependencies via Constraint Handling Rules. *Journal of Functional Programming 17*.

TRIFONOV, V. 2003. Simulating quantified class constraints. In *Haskell*.

VYTINIOTIS, D., JONES, S. L. P., SCHRIJVERS, T., AND SULZMANN, M. 2011. OUTSIDEIN(x): Modular type inference with local assumptions. *Journal of Functional Programming 21,* 4–5, 333–412.

WADLER, P. L. AND BLOTT, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *POPL*.

WENZEL, M. AND MNCHEN, T. 2000. Using axiomatic type classes in isabelle.

A. PROOFS

Throughout the proofs we refer to the type system rules of System F listed in Figure 11.

A.1. Type Preservation

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

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

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

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

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

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

Fig. 11. System F Type System

LEMMA A.1. *If*

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

then

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

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

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

It follows from (Ty-Var) that

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

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

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

Thus we have by (F-Var) that

$$|\Gamma|, |\Delta| \vdash x : |\tau|$$

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

It follows from (Ty-Abs) that

$$\Gamma; x : \rho_1 | \Delta \vdash e : \rho_2 \rightsquigarrow E$$

and by the indution hypothesis that

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

As all variables are renamed unique, it is easy to verify that this also holds:

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

Hence, by (`F-Abs`) we have

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

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

By the induction hypothesis, we have:

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

and

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

Then it follows by (`F-App`) that

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

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

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

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

As $\alpha \notin \Delta$, it is easy to see that the above is equivalent to

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

Thus, based on (`F-TAbs`) we have

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

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

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

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

By (`Ty-TApp`) and the induction hypothesis, it follows that

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

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

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

It follows from (`F-TApp`) that

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

which is easily seen to be equivalent to

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

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

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

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

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

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

Thus, based on (`F-Abs`) we have

$$|\Gamma|, |\Delta| \vdash \lambda x : |\rho_1|.E : |\rho_1| \rightarrow |\rho_2|$$

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

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

(`Ty-IApp`) $\Gamma|\Delta \vdash e_1$ **with** $e_2 : \rho_1 \rightsquigarrow E_1\, E_2$.

From (`Ty-IApp`) and the induction hypothesis we have:

$$|\Gamma|, |\Delta| \vdash E_1 : |\rho_2 \Rightarrow \rho_1|$$

and

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

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

$$|\Gamma|, |\Delta| \vdash E_1 : |\rho_2| \rightarrow |\rho_1|$$

Hence, based on (`F-App`) we know

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

(`Ty-Query`) $\Gamma|\Delta \vdash ?\rho : \rho \rightsquigarrow E$.

From (`Ty-Query`) we have

$$\Delta \vdash_r \rho \rightsquigarrow E$$

Based on Lemma A.2 we then know

$$|\Delta| \vdash E : |\rho|$$

Hence, because all variables are unique

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

□

LEMMA A.2. *If*

$$\Delta \vdash_r \rho \rightsquigarrow E$$

then

$$|\Delta| \vdash E : |\rho|$$

PROOF.
By induction on the derivation.

(R-TAbs) $\Delta \vdash_r \forall \alpha.\rho \rightsquigarrow \Lambda \alpha.E$.

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

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

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

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

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

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

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

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

(R-TApp). $\Delta \vdash_r \rho[\rho'/\alpha] \rightsquigarrow E\,|\rho'|$

From rule (`R-TApp`) and the induction hypothesis, we have

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

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

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

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

$$|\Delta| \vdash E\,|\rho'| : |\rho|[|\rho'|/\alpha]$$

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

$$|\Delta| \vdash E\,|\rho'| : |\rho[\rho'/\alpha]|$$

(R-IVar). $\Delta \vdash_r \rho \rightsquigarrow x$

From rule (`R-IVar`) we have

$$(\rho \rightsquigarrow x) \in \Delta$$

Hence, based on the definition of $|\cdot|$, we have

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

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

$$|\Delta| \vdash x : |\rho|$$

(R-IAbs). $\Delta \vdash_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$

From rule (`R-IAbs`) and the induction hypothesis, we have

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

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

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

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

$$|\Delta| \vdash \lambda x : |\rho_1|.E : |\rho_1| \rightarrow |\rho_2|$$

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

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

(R-IApp). $\Delta \vdash_r \rho_2 \rightsquigarrow E_2\, E_1$

From rule (`R-IAbs`) and the induction hypothesis, we have

$$|\Delta| \vdash E_1 : |\rho_1|$$

and

$$|\Delta| \vdash E_2 : |\rho_1 \Rightarrow \rho_2|$$

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

$$|\Delta| \vdash E_2 : |\rho_1| \rightarrow |\rho_2|$$

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

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

□

A.2. Soundness of Deterministic Resolution

Lemma A.5 states that deterministic resolution is sound with respect to non-deterministic resolution. Its proof relies on Lemma A.3, which states that $\Delta\langle\tau\rangle$ returns an element in the environment.

The proof of Lemma A.5 also proceeds by mutual induction with the proof of Lemma A.4. The latter lemma states that the auxiliary $\vdash_\downarrow$ relation is sound with respect to non-deterministic resolution.

LEMMA A.3. *If*

$$\Delta\langle\tau\rangle = \rho \rightsquigarrow x$$

then

$$(\rho \rightsquigarrow x) \in \Delta$$

The proof is trivial.

LEMMA A.4. *If*

$$\Delta; \rho \rightsquigarrow E_1 \vdash_\downarrow \tau \rightsquigarrow E_2$$

and

$$\Delta \vdash_r^2 \rho \rightsquigarrow E_1$$

then

$$\Delta \vdash_r^2 \tau \rightsquigarrow E_2$$

PROOF. The proof proceeds by induction on the derivation of the $\vdash_\downarrow$ assumption.

(I-Simp). $\Delta; \tau \rightsquigarrow E \vdash_\downarrow \tau \rightsquigarrow E$
The conclusion follows trivially.
(I-IAbs). $\Delta; \rho_1 \Rightarrow \rho_2 \rightsquigarrow E_1 \vdash_\downarrow \tau \rightsquigarrow E_3$

From the (`I-IAbs`) rule and the next lemma, it follows that

$$\Delta \vdash_r^2 \rho_1 \rightsquigarrow E_2$$

Given the above, as well as the assumption $\Delta \vdash^2_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow E_1$, it follows from rule (`R-IAbs`) that

$$\Delta \vdash^2_r \rho_2 \rightsquigarrow E_1\, E_2$$

From the (`I-IAbs`) rule and the induction hypothesis, we then have

$$\Delta \vdash^2_r \tau \rightsquigarrow E_3$$

(`I-TAbs`). $\Delta; \forall\alpha.\rho \rightsquigarrow E_1 \vdash_\downarrow \tau \rightsquigarrow E_2$

From the assumption $\Delta \vdash^2_r \forall\alpha.\rho \rightsquigarrow E_1$ and rule (`R-TApp`) it follows that

$$\Delta \vdash^2_r \rho[\rho'/\alpha] \rightsquigarrow E_1\, |\rho'|$$

From that, the (`I-TAbs`) rule and the induction hypothesis, we then have

$$\Delta \vdash^2_r \tau \rightsquigarrow E_2$$

□

LEMMA A.5. *If*

$$\Delta \vdash^3_r \rho \rightsquigarrow E$$

then

$$\Delta \vdash^2_r \rho \rightsquigarrow E$$

PROOF. The proof proceeds by induction on the derivation.

(`R-IAbs`)^3. $\Delta \vdash^3_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$
From rule (`R-IAbs`)^3 and the induction hypothesis, we have

$$\Delta, x : \rho_1 \vdash^2_r \rho_2 \rightsquigarrow E$$

Hence, from rule (`R-IAbs`)^2 it follows that

$$\Delta \vdash^2_r \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda x : |\rho_1|.E$$

(`R-TAbs`)^3. $\Delta \vdash^3_r \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E$
From rule (`R-TAbs`)^3 and the induction hypothesis, we have

$$\Delta \vdash^2_r \rho \rightsquigarrow E$$

Hence, from rule (`R-TAbs`)^2 it follows that

$$\Delta \vdash^2_r \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E$$

(`R-Simp`)^3. $\Delta \vdash^3_r \tau \rightsquigarrow E$

From the (`R-Simp`) rule and the lemma, it follows that

$$(\rho \rightsquigarrow x) \in \Delta$$

or, following rule (`R-IVar`), that

$$\Delta \vdash^2_r \rho \rightsquigarrow x$$

From the (`R-Simp`) rule and the other lemma, we also have that

$$\Delta \vdash^2_r \rho \rightsquigarrow x \quad \Rightarrow \quad \Delta \vdash^2_r \tau \rightsquigarrow E$$

Combining both observations, yields the desired result

$$\Delta \vdash^2_r \tau \rightsquigarrow E$$

□

A.3. Correctness of the Resolution Algorithm

Lemma A.8 states that the resolution algorithm $\vdash_{alg}$ is sound with respect to the deterministic resolution specification $\vdash_r$.

Its proof relies on Lemma A.7, which states that the auxiliary relation $\vdash_{match1st}$ is sound, whose proof in turn relies on Lemma A.6 which states that the auxiliary relation $\vdash_{match}$ is sound.

The completeness proof proceeds in a similar fashion.

LEMMA A.6. *If*

$$\rho; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}'; E'$$

then there exist $\bar{\rho}''$ with

$$\bar{\rho}\theta \subseteq \bar{\rho}'$$

where $\theta = [\bar{\rho}''/\bar{\alpha}]$,
and

$$\bar{\omega} \subseteq \bar{\omega}'$$

such that forall Δ and $\bar{E}''$:
if

$$\Delta \vdash_r \rho_i' \rightsquigarrow E_i'' \qquad (\forall \rho_i' \in \bar{\rho}')$$

then

$$\rho\theta \lhd \tau$$

and

$$\Delta; \rho\theta \rightsquigarrow E|\theta|\eta \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

where $\eta = [\bar{E}''/\bar{\omega}']$

PROOF. The proof proceeds by induction on the derivation.

(MTC-Simp). $\tau'; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}\theta; \bar{\omega}; E|\theta|$
Obviously, we have that

$$\bar{\omega} \subseteq \bar{\omega}$$

and

$$\bar{\rho}\theta \subseteq \bar{\rho}\theta$$

From rule (`MTC-Simp`) we have

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

This means

$$\tau'\theta = \tau$$

Hence, from rule (`M-Simp`) it follows that

$$\tau'\theta \lhd \tau$$

Also, from rule (`I-Simp`) it follows that

$$\Delta; \tau \rightsquigarrow E|\theta|\eta \vdash_{\downarrow} \tau \rightsquigarrow E|\theta|\eta$$

or, equivalently,

$$\Delta; \tau'\theta \rightsquigarrow E|\theta|\eta \vdash_{\downarrow} \tau \rightsquigarrow E|\theta|\eta$$

(MTC-IAbs). $\rho_1 \Rightarrow \rho_2; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}; E'$

From the rule (`MTC-IAbs`) and the induction hypothesis, it follows that

$$\bar{\omega}, \omega \subseteq \bar{\omega'}$$

Hence,

$$\bar{\omega} \subseteq \bar{\omega'}$$

Similarly, it follows that

$$(\bar{\rho}, \rho_1)\theta \subseteq \bar{\rho}'$$

Hence,

$$\bar{\rho}\theta \subseteq \bar{\rho}'$$

Also, from the rule (`MTC-IAbs`) and the induction hypothesis, it follows that

$$\rho_2\theta \lhd \tau$$

and, by rule (`M-IAbs`), we hence have

$$\rho_1\theta \Rightarrow \rho_2\theta \lhd \tau$$

or, more succinctly,

$$(\rho_1 \Rightarrow \rho_2)\theta \lhd \tau$$

Finally, from the rule (`MTC-IAbs`) and the induction hypothesis, it follows that

$$\Delta; \rho_2\theta \rightsquigarrow (E\,\omega)|\theta|\eta \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

or

$$\Delta; \rho_2\theta \rightsquigarrow (E\theta\eta)\,(\omega\eta) \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

Using rule (`I-IAbs`) we may then conclude

$$\Delta; \rho_1\theta \Rightarrow \rho_2\theta \rightsquigarrow E\theta\eta \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

or, equivalently,

$$\Delta; (\rho_1 \Rightarrow \rho_2)\theta \rightsquigarrow E\theta\eta \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

(MTC-TAbs). $\forall\alpha.\rho; \bar{\rho}; \bar{\alpha}; \bar{\omega}; E \vdash_{match} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}; E'$

From the rule (`MTC-TAbs`) and the induction hypothesis, it follows that

$$\bar{\omega} \subseteq \bar{\omega'}$$

Similarly, it follows that

$$\bar{\rho}\theta \subseteq \bar{\rho}'$$

Also it follows that

$$\rho\theta \lhd \tau$$

or, equivalently,

$$\rho[\bar{\alpha}\theta/\bar{\alpha}][\alpha\theta/\alpha] \lhd \tau$$

Hence, following rule (`MTC-TAbs`), we have that

$$\forall\alpha.(\rho[\bar{\alpha}\theta/\bar{\alpha}]) \lhd \tau$$

or, equivalently,

$$(\forall \alpha.\rho)[\bar{\alpha}\theta/\bar{\alpha}] \lhd \tau$$

Finally, from the rule (MTC-TAbs) and the induction hypothesis, it follows that

$$\Delta; \rho\theta \rightsquigarrow (E\,\alpha)|\theta|\eta \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

or, equivalently,

$$\Delta; \rho\theta \rightsquigarrow (E|\theta|[\alpha|\theta|/\alpha]\eta)\,(\alpha|\theta|) \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

Hence, following rule (I-TAbs), we get

$$\Delta; (\forall \alpha.\rho)\theta \rightsquigarrow E|\theta|\eta \vdash_{\downarrow} \tau \rightsquigarrow E'\eta$$

□

LEMMA A.7. *If*

$$\Delta \vdash_{match1st} \tau \hookrightarrow \bar{\rho}'; \bar{\omega}; E$$

then there exist ρ *and* x *such that*

$$\Delta\langle\tau\rangle = \rho \rightsquigarrow x$$

and for all $\Delta' \supseteq \Delta$ *and for all* $\bar{E}'$ *such that*

$$\Delta' \vdash_r \rho'_i \rightsquigarrow E'_i \qquad (\forall \rho'_i \in \bar{\rho}')$$

we have that

$$\Delta'; \rho \rightsquigarrow x \vdash_{\downarrow} \tau \rightsquigarrow E\eta$$

where $\eta = [\bar{E}'/\bar{\omega}]$.

PROOF. The proof proceeds by induction on the derivation.

(M1-Head). $\Delta, \rho \rightsquigarrow x \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}; E$
From rule (M1-Head) and the previous lemma, we then have that

$$\rho \lhd \tau$$

Using rule (L-Head) we can conclude that

$$(\Delta, \rho \rightsquigarrow x)\langle\tau\rangle = \rho \rightsquigarrow x$$

Similarly, we have that

$$\Delta'; \rho \rightsquigarrow x\eta \vdash_{\downarrow} \tau \rightsquigarrow E\eta$$

which simplifies to

$$\Delta'; \rho \rightsquigarrow x \vdash_{\downarrow} \tau \rightsquigarrow E\eta$$

(M1-Tail). $\Delta, \rho \rightsquigarrow x \vdash_{match1st} \tau \hookrightarrow \bar{\rho}; \bar{\omega}; E$
From rule (M1-Tail) and the induction hypothesis, we then have that there exist ρ' and x' such that

$$\Delta\langle\tau\rangle = \rho' \rightsquigarrow x'$$

Also from rule (M1-Tail) and a previous lemma, we have that

$$\rho \not\lhd \tau$$

Hence, using rule (`L-Tail`) we can conclude that

$$(\Delta, \rho \rightsquigarrow x)\langle\tau\rangle = \rho' \rightsquigarrow x'$$

From rule (`M1-Tail`) and the induction hypothesis, we also have that

$$\Delta'; \rho' \rightsquigarrow x' \vdash_{\downarrow} \tau \rightsquigarrow E\eta$$

for $\Delta' \supseteq (\Delta, \rho \rightsquigarrow x)$.

□

LEMMA A.8. *If*

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

then

$$\Delta \vdash_{r} \rho \rightsquigarrow E$$

PROOF. The proof proceeds by induction on the derivation.

(`Alg-TAbs`). $\Delta \vdash_{alg} \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E$
From rule (`Alg-TAbs`) and the induction hypothesis, it follows that

$$\Delta \vdash_{r} \rho \rightsquigarrow E$$

Using rule (`R-TAbs`), we then get

$$\Delta \vdash_{r} \forall\alpha.\rho \rightsquigarrow \Lambda\alpha.E$$

(`Alg-IAbs`). $\Delta \vdash_{alg} \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda(x : |\rho_1|).E$
From rule (`Alg-IAbs`) and the induction hypothesis, it follows that

$$\Delta, \rho_1 \rightsquigarrow x \vdash_{r} \rho_2 \rightsquigarrow E$$

Using rule (`R-IAbs`), we then get

$$\Delta \vdash_{r} \rho_1 \Rightarrow \rho_2 \rightsquigarrow \lambda(x : |\rho_1|).E$$

(`Alg-Simp`). $\Delta \vdash_{alg} \tau \rightsquigarrow E[\bar{\omega}/\bar{E}]$
From rule (`Alg-Simp`) and the previous lemma it follows that

$$\Delta\langle\tau\rangle = \rho \rightsquigarrow x$$

and

$$\Delta; \rho \rightsquigarrow x \vdash_{\downarrow} \tau \rightsquigarrow E[\bar{\omega}/\bar{E}]$$

Hence, using rule (`R-Simp`), we conclude

$$\Delta \vdash_{r} \tau \rightsquigarrow E[\bar{\omega}/\bar{E}]$$

□