The Implicit Calculus

A New Foundation for Generic Programming

Bruno C. d. S. Oliveira
National University of Singapore
bruno@ropas.snu.ac.kr

Tom Schrijvers
Universiteit Gent
tom.schrijvers@ugent.be

Wontae Choi Wonchan Lee
Kwangkeun Yi
Seoul National University
{wtchoi,wclee,kwang}@ropas.snu.ac.kr

Abstract

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*. This turns 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 ($\lambda_{\Rightarrow}$), 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

Keywords Implicit parameters, type classes, C++ concepts, generic programming, Haskell, Scala.

1. Introduction

Generic programming (GP) [23] 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).

PLDI'12, June 11–16, Beijing, China.

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:

$$sort\,[\alpha] : (\alpha \to \alpha \to Bool) \to List\ \alpha \to List\ \alpha$$

with 3 parameters: the type of the elements in the list (α); the comparison operator; and the list to be compared. 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 greatly simplifies such calls by making the type argument and the comparison operator implicit.

$$isort : \forall\alpha.(\alpha \to \alpha \to Bool) \Rightarrow List\ \alpha \to List\ \alpha$$

The function $isort$ declares that the comparison function is implicit by using $\Rightarrow$ instead of $\to$. It is used as:

$$\textbf{implicit}\ \{cmpInt : Int \to Int \to Bool\}\ \textbf{in}\ (isort\,[2,1,3], isort\,[5,9,3])$$

The two calls of $isort$ each take only one explicit argument: the list to be sorted. Both the concrete type of the elements (Int) and the comparison operator ($cmpInt$) are *implicitly* instantiated.

The element type is automatically inferred from the type of the list. 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. Thus, in the subexpression ($isort\,[2,1,3], isort\,[5,9,3]$), $cmpInt$ is in the local scope and available for resolution.

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 [23]. These ideas were used in C++ libraries such as the Standard Template Library [24] and Boost [1]. In the FP community, Haskell *type classes* [42] have proven to be an excellent mechanism for GP, although their original design did not have that purpose. As years passed the FP community created its own forms of GP [14, 10, 21].

Garcia et al.'s [9] 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* [4, 3], 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* [23, 11, 35].

Several researchers started working on various approaches to concepts (see Siek's work [34] for a historical overview). Some researchers focused on integrating concepts into C++ [7, 11], while others focused on developing new languages with GP in mind. The work on System F^G [35, 36] 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[1].

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* [26]. Yet, ultimately, they proved to be quite flexible and in some ways superior to type classes. In fact Scala turns out to have very good support for generic programming [28, 29].

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 with type classes gave the FP community insights about the limitations of type classes. Some of these limitations were addressed by concept proposals. Other limitations were solved by implicits. However, as far as we know, no existing language or language proposal overcomes all limitations. We discuss these limitations next.

Global scoping: In Haskell, rules[2] are global and there can be only a single rule for any given type [18, 2, 6, 8]. Locally scoped rules are not available. Several researchers have already proposed to fix this issue: with named rules [18] or locally scoped ones [2, 6, 8]. 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 [13]. Yet, the need for first-class type classes is real in practice. For example, Lämmel and Peyton Jones [21] desire the following type class for their GP approach:

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

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 modeled with types, it is possible to abstract over concepts. Oliveira and Gibbons [28] 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 [12], 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:

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

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 $\lambda_{\Rightarrow}$, 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 $\lambda_{\Rightarrow}$ 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 $\lambda_{\Rightarrow}$ by combining type-directed resolution with conventional type-inference. We illustrate this on a simple, but quite expressive 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, and like System F^G, 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 [12] have discussed higher-order rules informally and several other researchers noted their usefulness [40, 31, 28], 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 $\lambda_{\Rightarrow}$ 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 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* $\lambda_{\Rightarrow}$ provides a simple, expressive and general formal model for implicits. Despite its expressiveness, the calculus is minimal and provides an ideal setting for the formal study of implicits and GP.
- Of particular interest is our resolution mechanism, which is significantly more expressive than existing mechanisms in the

[1] In the context of C++ rules correspond to *models* or *concept_maps*.

[2] In the context of Haskell rules correspond to *type-class instances*.

literature. It is based on a simple (logic-programming style) query language, works for any type, and it supports partial resolution as well as higher-order rules.

- The calculus has a polymorphic type system and an elaboration semantics to System F. This also provides an effective implementation of our calculus. The elaboration semantics is proved to be type-preserving, ensuring the soundness of the calculus.
- We present a small, but realistic source language, built on top of $\lambda_{\Rightarrow}$ via a type-directed encoding. This language features implicit instantiation and a simple type of interface, which can be used to model simple forms of concepts. This source language also supports higher-order rules.
- Finally, both $\lambda_{\Rightarrow}$ and the source language have been implemented and the source code for their implementation is available at `http://ropas.snu.ac.kr/~bruno/implicit`.

Organization Section 2 presents an informal overview of our calculus. Section 3 shows a polymorphic type system that statically excludes ill-behaved programs. Section 4 shows the elaboration semantics of our calculus into System F and correctness results. Section 5 presents the source language and its encoding into $\lambda_{\Rightarrow}$. Section 6 discusses comparisons and related work. Section 7 concludes. The companion technical report [30] provides additional technical material and proofs.

2. Overview of the Implicit Calculus $\lambda_{\Rightarrow}$

Our calculus $\lambda_{\Rightarrow}$ 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:

$$types \cong propositions \cong rules$$

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 [19]. Resolution [20] 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 $\lambda_{\Rightarrow}$ 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 $\lambda_{\Rightarrow}$ 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: $\lambda_{\Rightarrow}$ 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 computes an $Int \times Bool$ pair from implicit Int and $Bool$ values:

$$(|(?Int + 1, \neg\ ?Bool) : \{Int, Bool\} \Rightarrow Int \times Bool|)$$

The rule abstraction syntax resembles a type-annotated expression: the expression $(?Int + 1, \neg\ ?Bool)$ to the left of the colon is the *rule body*, and to the right is the *rule type* $\{Int, Bool\} \Rightarrow Int \times Bool$. A rule abstraction abstracts over a set of implicit values (here $\{Int, Bool\}$), or, more generally, over rules to build values.

Hence, when a value of type $Int \times Bool$ is needed (expressed by the query $?(Int \times Bool)$), the above rule can be used, provided that an integer and a boolean value are available in the implicit environment. In such an environment, the rule returns a pair of the incremented Int value and negated $Bool$ value.

The implicit environment is extended through rule application (analogous to extending the environment with function applications). Rule application is expressed as, for example:

$$(|(?Int + 1, \neg\ ?Bool) : \{Int, Bool\} \Rightarrow Int \times Bool|)\ \mathbf{with}\ \{1, True\}$$

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

$$\mathbf{implicit}\ \{1, True\}\ \mathbf{in}\ (?Int + 1, \neg\ ?Bool)$$

which returns $(2, False)$.

Higher-order rules: $\lambda_{\Rightarrow}$ supports higher-order rules. For example, the rule

$$(|?(Int \times Int) : \{Int, \{Int\} \Rightarrow Int \times Int\} \Rightarrow Int \times Int|),$$

when applied, will compute an integer pair given an integer and a rule to compute an integer pair from an integer. Hence, the following rule application returns $(3, 4)$:

$$\mathbf{implicit}\ \{3, (|(?Int, ?Int + 1) : \{Int\} \Rightarrow Int \times Int|)\}\ \mathbf{in}\ ?(Int \times Int)$$

Recursive resolution: Note that resolving the query $?(Int \times Int)$ involves applying multiple rules. The current environment does not contain the required integer pair. It does however contain the integer 3 and a rule $(|(?Int, ?Int + 1) : \{Int\} \Rightarrow Int \times Int|)$ 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: $\lambda_{\Rightarrow}$ allows polymorphic rules. For example, the rule

$$(|(?\alpha, ?\alpha) : \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha|)$$

can be instantiated to multiple rules of monomorphic types

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

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

$$\mathbf{implicit}\ \{3, True, (|(?\alpha, ?\alpha) : \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha|)\}\ \mathbf{in}\ (?(Int \times Int), ?(Bool \times Bool))$$

Polymorphic rules can also be used to resolve polymorphic queries:

$$\mathbf{implicit}\ \{(|(?\alpha, ?\alpha) : \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha|)\}\ \mathbf{in}\ ?(\forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha)$$

Combining higher-order and polymorphic rules: The rule

$$(|(?((Int \times Int) \times (Int \times Int))) : \{Int, \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha\} \Rightarrow (Int \times Int) \times (Int \times Int)|)$$

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

$$\forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha$$

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

$$\mathbf{implicit}\ \{3, (|(?\alpha, ?\alpha) : \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha|)\}\ \mathbf{in}\ ?((Int \times Int) \times (Int \times Int))$$

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

$$\begin{array}{l}\textbf{implicit } \{1\} \textbf{ in} \\ \quad \textbf{implicit } \{ \mathit{True}, (\!| \textbf{ if } ?\mathit{Bool} \textbf{ then } 2 : \{\mathit{Bool}\} \Rightarrow \mathit{Int} |\!) \} \\ \quad \textbf{in } ?\mathit{Int}\end{array}$$

The query $?Int$ is not resolved with the integer value 1. Instead the rule that returns an integer from a boolean is applied to the boolean $True$, because those two rules can provide an integer value and they are 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 $\lambda_{\Rightarrow}$ through nested scoping. The nearest matching rule takes priority over other matching rules. For example consider the following program:

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

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

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

Then the lexically nearest match is $\lambda x.x{:}\forall\alpha.\alpha \to \alpha$ and evaluating this program results in 1.

3. The $\lambda_{\Rightarrow}$ Calculus

This section formalizes the syntax and type system of $\lambda_{\Rightarrow}$.

3.1 Syntax

This is the syntax of the calculus:

$$\begin{array}{lrcl}\text{(Simple) Types} & \tau & ::= & \alpha \mid \mathit{Int} \mid \tau_1 \to \tau_2 \mid \rho \\ \text{Rule Types} & \rho & ::= & \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau \\ \text{Expressions} & e & ::= & n \mid x \mid \lambda x : \tau.e \mid e_1\ e_2 \\ & & \mid & ?\rho \mid (\!| e : \rho |\!) \mid e[\vec{\tau}] \mid e \textbf{ with } \overline{e : \rho}\end{array}$$

Types τ are either type variables α, the integer type Int, function types $\tau_1 \to \tau_2$ or rule types ρ. A *rule type* $\rho = \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau$ is a type scheme with universally quantified variables $\vec{\alpha}$ and an (implicit) *context* $\bar{\rho}$. This *context* summarizes the assumed implicit environment. Note that we use $\vec{o}$ to denote an ordered sequence $o_1, \ldots, o_n$ of entities and $\bar{o}$ to denote a set $\{o_1, \ldots, o_n\}$. Such ordered sequences and sets can be empty, and we often omit empty universal quantifiers and empty contexts from a rule type. The base case of rule types is when $\bar{\rho}$ is the empty set (that is $\forall\vec{\alpha}.\{\} \Rightarrow \tau$ or, more compactly, $\forall\vec{\alpha}.\tau$).

Expressions include integer constants n and the three basic typed λ-calculus expressions (variables, lambda binders and applications). A *query* $?\rho$ queries the implicit environment for a value of type ρ. A *rule abstraction* $(\!| e : \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau |\!)$ builds a rule whose type is $\forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau$ and whose body is e.

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.

Note that, unlike System F, our calculus does not have a separate Λ binder for type variables. Instead rule abstractions play a dual role in the binding structure: 1) the universal quantification of type variables (which binds types), and 2) the context (which binds a rule set). Therefore, a Λ binder can be encoded using a rule with an empty context:

$$\Lambda\,\vec{\alpha}.(e : \tau) \stackrel{\text{def}}{=} (\!| e : \forall\vec{\alpha}.\tau |\!)$$

The design choice of making rules double binders is due to our interpretation of rules as logic programming rules[3]. After all, in the matching process of resolution, a rule is applied as a unit. Hence, separating rules into more primitive binders (à la System F's type and value binders) would only complicate the definition of resolution unnecessarily. However, elimination can be modularized into two constructs: *type application* $e[\bar{\tau}]$ and *rule application* $e \textbf{ with } \overline{e : \rho}$.

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

$$\textbf{implicit } \overline{e : \rho} \textbf{ in } e_1 : \tau \stackrel{\text{def}}{=} (\!| e_1 : \bar{\rho} \Rightarrow \tau |\!) \textbf{ with } \overline{e : \rho}$$

For readability purposes, when we use **implicit** we omit the type annotation τ. As we shall see in Section 5 this annotation can be automatically inferred.

For brevity and simplicity reasons, we have kept $\lambda_{\Rightarrow}$ small. In examples we may use additional syntax such as built-in integer operators and boolean literals and types.

3.2 Type System

Figure 1 presents the static type system of $\lambda_{\Rightarrow}$. The typing judgment $\Gamma \mid \Delta \vdash e : \tau$ means that expression e has type τ under type environment Γ and implicit environment Δ. The auxiliary resolution judgment $\Delta \vdash_r \rho$ expresses that type ρ is resolvable with respect to Δ. Here, Γ is the conventional type environment that captures type variables; Δ is the *implicit environment*, defined as a stack of contexts. Figure 1 also presents lookup in the implicit environment ($\Delta\langle\tau\rangle$) and in contexts ($\bar{\rho}\langle\tau\rangle$).

We will not discuss the first four rules ((`TyInt`), (`TyVar`), (`TyAbs`) and (`TyApp`)) because they are entirely standard. For now we also ignore the gray-shaded conditions in the other rules; they are explained in Section 3.3.

Rule (`TyRule`) checks a rule abstraction $(\!| e : \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau |\!)$ by checking whether the rule's body e actually has the type τ under the assumed implicit type context $\bar{\rho}$. Rule (`TyInst`) instantiates a rule type's type variables $\vec{\alpha}$ with the given types $\vec{\tau}$, and rule (`TyRApp`) instantiates the type context $\bar{\rho}$ with expressions of the required rule types $\overline{e : \rho}$. Finally, rule (`TyQuery`) delegates queries directly to the resolution rule (`TyRes`).

Resolution Principle The underlying principle of resolution in $\lambda_{\Rightarrow}$ originates from resolution in logic. Following the Curry-Howard correspondence, we assign to each type a corresponding logical interpretation with the $(\cdot)^\dagger$ function:

Definition 3.1 (Logical Interpretation)**.**

$$\begin{array}{rcl}\alpha^\dagger & = & \alpha^\dagger \\ \mathit{Int}^\dagger & = & \mathit{Int}^\dagger \\ (\tau_1 \to \tau_2)^\dagger & = & \tau_1^\dagger \to^\dagger \tau_2^\dagger \\ (\forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau)^\dagger & = & \forall\vec{\alpha}^\dagger.\ \bigwedge_{\rho \in \bar{\rho}} \rho^\dagger \Rightarrow \tau^\dagger\end{array}$$

Here, type variables α map to propositional variables $\alpha^\dagger$ and the primitive type Int maps to the propositional constant $Int^\dagger$. Unlike Curry-Howard, we do not map function types to logical implications; we deliberately restrict our implicational reasoning to rule types. So, instead we also map the function arrow to an uninterpreted higher-order predicate $\to^\dagger$. Finally, as already indicated, we map rule types to logical implications.

[3] In Prolog these are not separated either.

Type Environments $\Gamma ::= \cdot \mid \Gamma; x : \tau$
Implicit Environments $\Delta ::= \cdot \mid \Delta; \bar{\rho}$

$\boxed{\Gamma \mid \Delta \vdash e : \tau}$

(TyInt) $\Gamma \mid \Delta \vdash n : Int$

(TyVar) $$\frac{(x : \tau) \in \Gamma}{\Gamma \mid \Delta \vdash x : \tau}$$

(TyAbs) $$\frac{\Gamma; x : \tau_1 \mid \Delta \vdash e : \tau_2}{\Gamma \mid \Delta \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

(TyApp) $$\frac{\Gamma \mid \Delta \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \mid \Delta \vdash e_2 : \tau_2}{\Gamma \mid \Delta \vdash e_1\, e_2 : \tau_1}$$

(TyRule) $$\frac{\rho = \forall \vec{\alpha}.\bar{\rho} \Rightarrow \tau \quad \mathsf{unambiguous}(\rho) \qquad \Gamma \mid \Delta; \bar{\rho} \vdash e : \tau \quad \vec{\alpha} \cap ftv(\Gamma, \Delta) = \emptyset}{\Gamma \mid \Delta \vdash (\!|e : \rho|\!) : \rho}$$

(TyInst) $$\frac{\Gamma \mid \Delta \vdash e : \forall \vec{\alpha}.\bar{\rho} \Rightarrow \tau}{\Gamma \mid \Delta \vdash e[\vec{\tau}] : [\vec{\alpha} \mapsto \vec{\tau}](\bar{\rho} \Rightarrow \tau)}$$

(TyRApp) $$\frac{\Gamma \mid \Delta \vdash e : \bar{\rho} \Rightarrow \tau \qquad \Gamma \mid \Delta \vdash e_i : \rho_i \quad (\forall e_i : \rho_i \in \overline{e : \rho})}{\Gamma \mid \Delta \vdash (e \ \mathbf{with}\ \overline{e : \rho}) : \tau}$$

(TyQuery) $$\frac{\Delta \vdash_r \rho \quad \mathsf{unambiguous}(\rho)}{\Gamma \mid \Delta \vdash ?\rho : \rho}$$

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

(TyRes) $$\frac{\Delta\langle\tau\rangle = \bar{\rho}' \Rightarrow \tau \qquad \Delta \vdash_r \rho_i \quad (\forall \rho_i \in \bar{\rho}' - \bar{\rho})}{\Delta \vdash_r \forall \vec{\alpha}.\bar{\rho} \Rightarrow \tau}$$

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

$$\frac{\bar{\rho}\langle\tau\rangle = \rho \quad \mathsf{no\_overlap}(\bar{\rho}, \tau)}{(\Delta; \bar{\rho})\langle\tau\rangle = \rho}$$

$$\frac{\bar{\rho}\langle\tau\rangle = \bot \quad \Delta\langle\tau\rangle = \rho}{(\Delta; \bar{\rho})\langle\tau\rangle = \rho}$$

$\boxed{\bar{\rho}\langle\tau\rangle = \rho}$

$$\frac{\rho \in \bar{\rho} \qquad \rho = \forall \vec{\alpha}'.\bar{\rho}' \Rightarrow \tau' \qquad \theta\tau' = \tau}{\bar{\rho}\langle\tau\rangle = \theta\bar{\rho}' \Rightarrow \tau}$$

Figure 1. Type System

Resolution in $\lambda_{\Rightarrow}$ then corresponds to checking entailment of the logical interpretation. We postulate this property as a theorem that constrains the design of resolution.

Theorem 3.1 (Resolution Specification)**.**

If $\Delta \vdash_r \rho$*, then* $\Delta^\dagger \models \rho^\dagger$.

Resolution for Simple Types The step from the logical interpretation to the (TyRes) rule in Figure 1 is non-trivial. So, let us first look at a simpler incarnation. What does resolution look like for simple types τ like Int?

(SimpleRes) $$\frac{\Delta\langle\tau\rangle = \bar{\rho}' \Rightarrow \tau \qquad \Delta \vdash_r \rho_i \quad (\forall \rho_i \in \bar{\rho}')}{\Delta \vdash_r \tau}$$

First, it looks up a *matching* rule type in the implicit environment by means of the lookup function $\Delta\langle\tau\rangle$ defined in Fig. 1. This partial function respects the nested scopes: it first looks in the topmost context of the implicit environment, and, only if it does not find a matching rule, does it descend. Within an environment context, the lookup function looks for a rule type whose right-hand side τ' can be instantiated to the queried τ using a matching unifier θ. This rule type is then returned in instantiated form.

The matching expresses that the looked-up rule produces a value of the required type. To do so, the looked-up rule may itself require other implicit values. This requirement is captured in the context $\bar{\rho}'$, which must be resolved recursively. Hence, the resolution rule is itself a recursive rule. When the context $\bar{\rho}'$ of the looked-up rule is empty, a base case of the recursion has been reached.

Example Consider this query for a tuple of integers:

$$Int; \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha \vdash_r Int \times Int$$

Lookup yields the second rule, which produces a tuple, instantiated to $\{Int\} \Rightarrow Int \times Int$ with matching substitution $\theta = [\alpha \mapsto Int]$. In order to produce a tuple, the rule requires a value of the component type. Hence, resolution proceeds by recursively querying for Int. Now lookup yields the first rule, which produces an integer, with empty matching substitution and no further requirements.

Resolution for Rule Types So far, so good. Apart from allowing any types, recursive querying for simple types is quite similar to recursive type class resolution, and $\lambda_{\Rightarrow}$ carefully captures the expected behavior. However, what is distinctly novel in $\lambda_{\Rightarrow}$, is that it also provides *resolution of rule types*, which requires a markedly different treatment.

(RuleRes) $$\frac{\Delta\langle\tau\rangle = \bar{\rho} \Rightarrow \tau}{\Delta \vdash_r \forall \vec{\alpha}.\bar{\rho} \Rightarrow \tau}$$

Here we retrieve a whole rule from the environment, including its context. Resolution again performs a lookup based on a matching right-hand side τ, but subsequently also matches the context with the one that is queried. No recursive resolution takes place.

Example Consider a variant of the above query:

$$Int; \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha \vdash_r \{Int\} \Rightarrow Int \times Int$$

Again lookup yields the second rule, instantiated to $\{Int\} \Rightarrow Int \times Int$. The context $\{Int\}$ of this rule matches the context of the queried rule. Hence, the query is resolved without recursive resolution.

Unified Resolution The feat that our actual resolution rule (TyRes) accomplishes is to unify these seemingly disparate forms of resolution into one single inference rule. In fact, both (SimpleRes) and (RuleRes) are special cases of (TyRes), which provides some additional expressiveness in the form of *partial resolution* (explained below).

The first hurdle for (TyRes) is that types τ and rule types ρ are different syntactic categories. Judging from its definition, (TyRes) only covers rule types. How do we get it to treat simple types then? Just promote the simple type τ to its corresponding rule type $\forall.\{\} \Rightarrow \tau$ and (TyRes) will do what we expect for simple types, including recursive resolution. At the same time, it still matches proper rule types exactly, without recursion, when that is appropriate.

Choosing the right treatment for the context is the second hurdle. This part is managed by recursively resolving $\bar{\rho}' - \bar{\rho}$. In the case of promoted simple types, $\bar{\rho}$ is empty, and the whole of $\bar{\rho}'$ is recursively solved; which is exactly what we want. In the case $\bar{\rho}'$ matches $\bar{\rho}$, no recursive resolution takes place. Again this perfectly corresponds to what we have set out above for proper rule types. However, there is a third case, where $\bar{\rho}' - \bar{\rho}$ is a non-empty proper

subset of $\bar{\rho}'$. We call this situation, where part of the retrieved rule's context is recursively resolved and part is not, *partial resolution*.

Example Here is another query variant:

$$Bool; \forall\alpha.\{Bool, \alpha\} \Rightarrow \alpha \times \alpha \vdash_r \{Int\} \Rightarrow Int \times Int$$

The first lookup yields the second rule, instantiated to $\{Bool, Int\} \Rightarrow Int \times Int$, which almost matches the queried rule type. Only $Bool$ in the context is unwelcome, so it is eliminated through a recursive resolution step. Fortunately, the first rule in the environment is available for that.

3.3 Additional Type System Conditions

The gray-shaded conditions in the type system are to check lookup errors (no_overlap) and ambiguous instantiations (unambiguous).

Avoiding Lookup Errors To prevent lookup failures, we have to check for two situations:

- A lookup has no matching rule in the environment.
- A lookup has multiple matching rules which have different rule types but can yield values of the same type (overlapping rules).

The former condition is directly captured in the definition of lookup among a set of rule types. The latter condition is captured in the no_overlap property, which is defined as:

$$\begin{array}{l}\mathsf{no\_overlap}(\{\rho_1, \ldots, \rho_n\}, \tau) \stackrel{\mathsf{def}}{=} \\ \quad \forall i, j. \quad \rho_i = \forall\vec{\alpha_i}.\bar{\rho_i} \Rightarrow \tau_i \quad \wedge \quad \exists\theta_i.\theta_i\tau_i = \tau \\ \qquad \wedge \quad \rho_j = \forall\vec{\alpha_j}.\bar{\rho_j} \Rightarrow \tau_j \quad \wedge \quad \exists\theta_j.\theta_j\tau_j = \tau \\ \qquad \Longrightarrow \quad i = j\end{array}$$

Avoiding Ambiguous Instantiations We avoid ambiguous instantiations in the same way as Haskell does: all quantified type variables ($\vec{\alpha}$) in a rule type ($\forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau$) must occur in τ. We use the unambiguous condition to check in (TyRule) and (TyQuery):

$$\begin{array}{l}\mathsf{unambiguous}(\forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau) = \vec{\alpha} \subseteq ftv(\tau) \\ \qquad\qquad \wedge \forall\rho_i \in \bar{\rho}.\mathsf{unambiguous}(\rho_i).\end{array}$$

If there is a quantified type variable not in type τ, the type may yield ambiguous instantiations (e.g. $\forall\alpha.\{\alpha\} \Rightarrow Int$).

4. Type-Directed Translation to System F

In this section we define the dynamic semantics of $\lambda_{\Rightarrow}$ in terms of System F's dynamic semantics, by means of a type directed translation. This translation turns implicit contexts into explicit parameters and statically resolves all queries, much like Wadler and Blott's dictionary passing translation for type classes [42]. The advantage of this approach is that we simultaneously provide a meaning to well-typed $\lambda_{\Rightarrow}$ programs and an effective implementation that resolves all queries statically.

4.1 Type-Directed Translation

Figure 2 presents the translation rules that convert $\lambda_{\Rightarrow}$ expressions into ones of System F extended with the integer and unit types. This figure essentially extends Figure 1 with the necessary information for the translation, but for readability we have omitted the earlier gray-shaded conditions.

The syntax of System F is as follows:

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

The main translation judgment is

$$\Gamma \mid \Delta \vdash e : \tau \rightsquigarrow E,$$

which states that the translation of $\lambda_{\Rightarrow}$ expression e with type τ is System F expression E, with respect to type environment Γ and

$$\begin{array}{lrcl}\text{Type Environments} & \Gamma & ::= & \cdot \mid \Gamma; x : \tau \\ \text{Translation Environments} & \Delta & ::= & \cdot \mid \Delta; \overline{\rho : x}\end{array}$$

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

(TrInt) $$\Gamma \mid \Delta \vdash n : Int \rightsquigarrow n$$

(TrVar) $$\frac{(x : \tau) \in \Gamma}{\Gamma \mid \Delta \vdash x : \tau \rightsquigarrow x}$$

(TrAbs) $$\frac{\Gamma; x : \tau_1 \mid \Delta \vdash e : \tau_2 \rightsquigarrow E}{\Gamma \mid \Delta \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : |\tau_1|.E}$$

(TrApp) $$\frac{\Gamma \mid \Delta \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow E_1 \quad \Gamma \mid \Delta \vdash e_2 : \tau_2 \rightsquigarrow E_2}{\Gamma \mid \Delta \vdash e_1\ e_2 : \tau_1 \rightsquigarrow E_1\ E_2}$$

(TrQuery) $$\frac{\Delta \vdash_r \rho \rightsquigarrow E}{\Gamma \mid \Delta \vdash ?\rho : \rho \rightsquigarrow E}$$

(TrRule) $$\frac{\rho = \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau \quad \vec{\alpha} \cap ftv(\Gamma, \Delta) = \emptyset \quad \Gamma \mid \Delta; \overline{\rho : x} \vdash e : \tau \rightsquigarrow E \quad \bar{x}\ \text{fresh}}{\Gamma \mid \Delta \vdash (|e : \rho|) : \rho \rightsquigarrow \Lambda\vec{\alpha}.\lambda(\vec{x} : |\vec{\rho}|).E}$$

(TrInst) $$\frac{\Gamma \mid \Delta \vdash e : \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau \rightsquigarrow E}{\Gamma \mid \Delta \vdash e[\vec{\tau}] : [\vec{\alpha} \mapsto \vec{\tau}](\bar{\rho} \Rightarrow \tau) \rightsquigarrow E\ |\vec{\tau}|}$$

(TrRApp) $$\frac{\Gamma \mid \Delta \vdash e : \bar{\rho} \Rightarrow \tau \rightsquigarrow E \quad \Gamma \mid \Delta \vdash e_i : \rho_i \rightsquigarrow E_i \quad (\forall e_i : \rho_i \in \overline{e : \rho})}{\Gamma \mid \Delta \vdash (e\ \mathbf{with}\ \overline{e : \rho}) : \tau \rightsquigarrow E\ \vec{E}}$$

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

(TrRes) $$\frac{\Delta(\tau) = \bar{\rho}' \Rightarrow \tau : E \quad \bar{x}\ \text{fresh} \quad \forall\rho_i \in \bar{\rho}' : \begin{cases} \Delta \vdash_r \rho_i \rightsquigarrow E_i & , \rho_i \notin \bar{\rho} \\ E_i = x_i & , \rho_i \in \bar{\rho} \end{cases}}{\Delta \vdash_r \forall\vec{\alpha}.\bar{\rho} \Rightarrow \tau \rightsquigarrow \Lambda\vec{\alpha}.\lambda(\vec{x} : |\vec{\rho}|).(E\ \vec{E})}$$

$\boxed{\Delta\langle\tau\rangle = \rho : E}$ $$\frac{\overline{\rho : x}\langle\tau\rangle = \rho : E}{(\Delta; \overline{\rho : x})\langle\tau\rangle = \rho : E}$$

$$\frac{\overline{\rho : x}\langle\tau\rangle = \bot \quad \Delta\langle\tau\rangle = \rho}{(\Delta; \overline{\rho : x})\langle\tau\rangle = \rho}$$

$\boxed{\overline{\rho : x}\langle\tau\rangle = \rho : E}$ $$\frac{(\rho : x) \in \overline{\rho : x} \quad \rho = \forall\vec{\alpha}'.\bar{\rho}' \Rightarrow \tau' \quad \theta\tau' = \tau \quad \theta = [\vec{\alpha}' \mapsto \vec{\tau}]}{\overline{\rho : x}\langle\tau\rangle = \theta\bar{\rho}' \Rightarrow \tau : x\ |\vec{\tau}|}$$

$$\begin{array}{rcl}|\alpha| & = & \alpha \\ |Int| & = & Int \\ |\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| \\ |\forall\vec{\alpha}.\{\rho_1, \cdots, \rho_n\} \Rightarrow \tau| & = & \forall\vec{\alpha}.|\rho_1| \to \cdots \to |\rho_n| \to |\tau| \\ |\Gamma| & = & \{(x : |\tau|) \mid (x : \tau) \in \Gamma\} \\ |\Delta| & = & \{(x : |\rho|) \mid (\rho : x) \in \Delta\}\end{array}$$

Figure 2. Type-directed Translation to System F

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 value-level explicit evidence for the implicit rule. Lookup in the translation environment is defined similarly to lookup in the type environment, except that the lookup now returns a pair of a rule type and an evidence variable.

Figure 2 also defines the type translation function $|\cdot|$ from $\lambda_{\Rightarrow}$ types τ to System F types T. In order to obtain a unique translation of types, we assume that the types in a context are lexicographically ordered.

Variables, lambda abstractions and applications are translated straightforwardly. Queries are translated by rule (`TrQuery`) using the auxiliary resolution judgment $\vdash_r$, defined by rule (`TrRes`). Note that rule (`TrRes`) performs the same process that rule (`TyRes`) performs in the type system except that it additionally collects evidence variables.

Rule (`TrRule`) translates rule abstractions to explicit type and value abstractions in System F, and rule (`TrInst`) translates instantiation to type application. Finally, rule (`TrRApp`) translates rule application to application in System F.

Example We have that:

$$\cdot \mid \cdot \vdash (\!|(?\alpha, ?\alpha) : \forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha|\!) \rightsquigarrow \Lambda\alpha.\lambda(x:\alpha).(x,x)$$

and also:

$$(Int : x_1), (\forall\alpha.\{\alpha\} \Rightarrow \alpha \times \alpha : x_2) \vdash_r Int \times Int \rightsquigarrow x_2\ Int\ x_1$$

For brevity, Figure 2 omits the case where the context of a rule type is empty. To properly handle empty contexts, the translation of rule type should include $|\{\} \Rightarrow \tau| = () \to |\tau|$ and the translation rules (`TrRule`), (`TrRApp`) and (`TrRes`) should be extended in the obvious way.

Theorem 4.1 (Type-preserving translation)**.** *Let e be a $\lambda_{\Rightarrow}$ expression, τ be a type and E be a System F expression. If $\cdot \mid \cdot \vdash e : \tau \rightsquigarrow E$, then $\cdot \vdash E : |\tau|$.*

Proof. (Sketch) We first prove[4] the more general lemma "if $\Gamma \mid \Delta \vdash e : \tau \rightsquigarrow E$, then $|\Gamma|, |\Delta| \vdash E : |\tau|$" by induction on the derivation of translation. Then, the theorem trivially follows. □

4.2 Dynamic Semantics

Finally, we define the dynamic semantics of $\lambda_{\Rightarrow}$ as the composition of the type-directed translation and System F's dynamic semantics. Following Siek's notation [35], this dynamic semantics is:

$$eval(e) = V \qquad \textit{where } \cdot \mid \cdot \vdash e : \tau \rightsquigarrow E \textit{ and } E \to^* V$$

with $\to^*$ the reflexive, transitive closure of System F's standard single-step call-by-value reduction relation.

Now we can state the conventional type safety theorem for $\lambda_{\Rightarrow}$:

Theorem 4.2 (Type Safety)**.** *If $\cdot \mid \cdot \vdash e : \tau$, 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 $\lambda_{\Rightarrow}$ (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 $\lambda_{\Rightarrow}$ 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 $\lambda_{\Rightarrow}$.

[4] in the technical report

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

Figure 3. Encoding the Equality Type Class

Interface Declarations

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

Types

T	::=	α	Type Variables
	$\mid$	Int	Integer Type
	$\mid$	$I\ \vec{T}$	Interface Type
	$\mid$	$T \to T$	Function
σ	::=	$\forall\overline{\alpha}.\ \overline{\sigma} \Rightarrow T$	Rule Type

Expressions

E	::=	n	Integer Literal
	$\mid$	x	Lambda Variable
	$\mid$	$\lambda x.E$	Abstraction
	$\mid$	$E_1\ E_2$	Application
	$\mid$	u	Let Variable
	$\mid$	$\textbf{let}\ u : \sigma\ =\ E_1\ \textbf{in}\ E_2$	Let
	$\mid$	$\textbf{implicit}\ \overline{u}\ \textbf{in}\ E_2$	Implicit Scoping
	$\mid$	$?$	Implicit Lookup
	$\mid$	$I\ \overline{u = E}$	Interface Implementation

Figure 4. Syntax of Source Language

5.1 Type-directed Translation to $\lambda_{\Rightarrow}$

The full syntax of the source language is presented in Figure 4. Its use is illustrated in the program of Figure 3, which comprises an encoding of Haskell's equality type class `Eq`. The example shows that the source language features a simple type of interface $I\ \vec{T}$ (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 modeled as regular functions taking a record as the first argument. So a field u with type T in an interface declaration $I\ \vec{\alpha}$ actually has type $\forall\bar{\alpha}.\{\} \Rightarrow I\ \vec{\alpha} \to T$. There are also other conventional programming constructs (such as **let** expressions, lambdas and primitive types).

Unlike the core language, we strongly differentiate between simple types T and type schemes σ in order to facilitate type inference. Moreover, as the source language provides implicit rather than explicit type instantiation, the order of type variables in a quantifier is no longer relevant. Hence, they are represented by a set $(\forall\bar{\alpha})$. We also distinguish simply typed variables x from let-bound variables u with polymorphic type σ.

Figure 5 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 simply and polymorphic variable typings. The connection between source types T and σ on the one hand and core types τ

Type Environments $G \quad ::= \quad \cdot \mid G, u : \sigma \mid G, x : T$

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

(TyIntL) $$G \vdash n : Int \rightsquigarrow n$$

(TyVar) $$\frac{G(x) = T}{G \vdash x : T \rightsquigarrow x}$$

(TyAbs) $$\frac{G, x : T_1 \vdash E \rightsquigarrow e}{G \vdash \lambda x.E : T_1 \to T_2 \rightsquigarrow \lambda x : [\![T_1]\!].e}$$

(TyApp) $$\frac{G \vdash E_1 : T_1 \to T_2 \rightsquigarrow e_1 \quad G \vdash E_2 : T_1 \rightsquigarrow e_2}{G \vdash E_1\ E_2 : T_2 \rightsquigarrow e_1\ e_2}$$

(TyLVar) $$\frac{G(u) = \forall \overline{\alpha}.\ \overline{\sigma} \Rightarrow T' \quad \theta = [\overline{\alpha} \mapsto \overline{T}] \quad T = \theta T' \quad q_i = (?[\![\theta\sigma_i]\!]) : [\![\theta\sigma_i]\!] \quad (\forall \sigma_i \in \overline{\sigma})}{G \vdash u : T \rightsquigarrow u[[\![\vec{T}]\!]]\ \mathbf{with}\ \overline{q}}$$

(TyLet) $$\frac{\sigma = \forall \overline{\alpha}.\overline{\sigma} \Rightarrow T_1 \quad G \vdash E_1 : T_1 \rightsquigarrow e_1 \quad G, u : \sigma \vdash E_2 : T_2 \rightsquigarrow e_2}{G \vdash\ \mathbf{let}\ u : \sigma = E_1\ \mathbf{in}\ E_2 : T_2 \rightsquigarrow (\lambda u : [\![\sigma]\!].e_2)\ (\!|e_1 : [\![\sigma]\!]|\!)}$$

(TyImp) $$\frac{G \vdash E : T \rightsquigarrow e \quad G(u_i) = \sigma_i \quad q_i = u_i : [\![\sigma_i]\!] \quad (\forall u_i \in \overline{u})}{G \vdash\ \mathbf{implicit}\ \overline{u}\ \mathbf{in}\ E : T \rightsquigarrow (\!|e : [\![\overline{\sigma}]\!] \Rightarrow [\![T]\!]|\!)\ \mathbf{with}\ \overline{q}}$$

(TyIVar) $$G \vdash ? : T \rightsquigarrow\ ?(\{\} \Rightarrow [\![T]\!])\ \mathbf{with}\ \{\}$$

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

$$\begin{array}{rcl} [\![\alpha]\!] & = & \alpha \\ [\![Int]\!] & = & Int \\ [\![T_1 \to T_2]\!] & = & [\![T_1]\!] \to [\![T_2]\!] \\ [\![I\ \vec{T}]\!] & = & I\ [\![\vec{T}]\!] \\ [\![\forall \overline{\alpha}.\overline{\sigma} \Rightarrow T]\!] & = & \forall [\![\vec{\alpha}]\!].[\![\overline{\sigma}]\!] \Rightarrow [\![T]\!] \end{array}$$

Figure 5. Type-directed Encoding of Source Language in $\lambda_{\Rightarrow}$

and ρ on the other hand is captured in the auxiliary function $[\![\cdot]\!]$. Note that this function imposes a canonical ordering $\vec{\alpha}$ on the set of quantifier variables $\bar{\alpha}$ (based on their precedence in the left-to-right prefix traversal of the quantified type term). For the translation of records, we assume that $\lambda_{\Rightarrow}$ is extended likewise with records.

let* and let-*bound variables The rule (TyLet) in Figure 5 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). Then it applies that abstraction to a rule whose rule type is just the corresponding (translated) type of the definition (σ_1), and whose body is the translation of 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 $\lambda_{\Rightarrow}$ 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 5, 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 Coq placeholder (_), which similarly instructs Coq to automatically infer a value.

The translation of source language queries, given by the rule (TyIVar), is fairly straightforward. To simplify type-inference, the query is limited to types, and does not support partial resolution (although other designs with more powerful queries are possible). In the translated code the query is combined with a rule instantiation and application in order to eliminate the empty rule set.

Implicit scoping The **implicit** construct, which has been already informally introduced in Section 1, is the core scoping construct of the source language. It is used in our example to first introduce definitions in the implicit environment ($eqInt_1$, $eqBool$ and $eqPair$) available at the expression

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

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

The translation rule (TyImp) of **implicit** into $\lambda_{\Rightarrow}$ also exploits type-information to avoid redundant type annotations. For example, it is not necessary to annotate the **let**-bound variables used in the rule set $\overline{u}$ because that information can be 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.

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

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 3 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 leads to a programming style akin 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 [11] 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 [38]. Moreover, various other proposals for GP mechanisms have their own notion of interface: Scala uses standard OO hierarchies; Dreyer et al. use ML-modules [8]; and in dependently typed systems (dependent) record types are used [37, 5].

An advantage of $\lambda_{\Rightarrow}$ 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 [29, 8, 37, 5] offer evidence that more sophisticated types of interfaces, including some form of refinement or associated types, can be built on top of $\lambda_{\Rightarrow}$.

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 polymorhism 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 $\lambda_{\Rightarrow}$ 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 $\lambda_{\Rightarrow}$. 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, as well as the *no_overlap* condition. 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 [33, 41].

6. Related Work

Throughout the paper we have already discussed a lot of related work. In what follows, we offer a more detailed technical comparison of $\lambda_{\Rightarrow}$ versus System F^G and Scala implicits, which are the closest to our work. Then we discuss the relation with other work in the literature.

System F^G Generally speaking our calculus is more primitive and general than System F^G. In contrast to $\lambda_{\Rightarrow}$, System F^G has both a notion of concepts and implicit instantiation of concepts[5]. This has the advantage that language designers can just reuse that infrastructure, instead of having to implement it. The language G [36] is based on System F^G and it makes good use of these built-in mechanisms. However, System F^G also imposes important design choices. Firstly it forces the language designer to use the notion of concepts that is built-in to System F^G. In contrast $\lambda_{\Rightarrow}$ offers a freedom of choice (see also the discussion in Section 5.2). Secondly, fixing implicit instantiation in the core prevents useful alternatives. For example, Scala and several other systems do provide implicit instantiation by default, but also offer the option of explicit instantiation, which is useful to resolve ambiguities [29, 18, 6, 8]. This cannot be modeled on top of System F^G, because explicit instantiation is not available. In contrast, by taking explicit instantiation (rule application) as a core feature, $\lambda_{\Rightarrow}$ can serve as a target for languages that offer both styles of instantiation.

There are also important differences in terms of scoping and resolution of rules. System F^G only formalizes a very simple type of resolution, which does not support recursive resolution. Furthermore, scoping is less fine-grained than in $\lambda_{\Rightarrow}$. For example, System F^G requires a built-in construct for **model** expressions, but in $\lambda_{\Rightarrow}$ **implicit** (which plays a similar role) is just syntactic sugar on top of more primitive constructs.

Scala Implicits Scala implicits are integrated in a full-blown language, but they have only been informally described in the literature [29, 27]. Our calculus aims at providing a formal model of implicits, but there are some noteworthy differences between $\lambda_{\Rightarrow}$ and Scala implicits. In contrast to $\lambda_{\Rightarrow}$, Scala has subtyping. As discussed in Section 5.2 subtyping would require some adaptations to our calculus. In Scala, nested scoping can only happen through subclassing and the rules for resolution in the presence of overlapping instances are quite ad-hoc. Furthermore, Scala has no (first-class) rule abstractions. Rather, implicit arguments can only be used in definitions. In contrast $\lambda_{\Rightarrow}$ provides a more general and disciplined account of scoping for rules.

Type Classes Obviously, the original work on type classes [42] and the framework of *qualified types* [15] around it has greatly influenced our own work, as well as that of System F^G and Scala.

There is a lot of work on Haskell type classes in the literature. Notably, there have been some proposals for addressing the limitations that arise from global scoping [18, 6]. However in those designs, type classes are still second-class and resolution only works for type classes. The GHC Haskell compiler supports overlapping instances [17], 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* [16], *associated types* [4, 3] and *type families* [32] are all examples of this trend. This line of work is orthogonal to our work.

Other Languages and Systems *Modular type classes* [8] 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 contrast to $\lambda_{\Rightarrow}$, implicit instantiation is limited to modules and, although local scoping is allowed, it cannot be nested.

Instance arguments [5] are an Agda extension that is closely related to implicits. However, unlike most GP mechanisms, implicit rules are not declared explicitly. Furthermore resolution is limited in its expressive power, to avoid introducing a different computational model in Agda. This design differs significantly from $\lambda_{\Rightarrow}$, where resolution is very expressive and the scoping mechanisms allow explicit rule declarations.

Implicit parameters [22] are a Haskell extension that allows *named* arguments to be passed implicitly. Implicit parameters are resolved by name, not by type and there is no recursive resolution.

GP and Logic Programming The connection between Haskell type classes and Prolog is folklore. Neubauer et. al. [25] also

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

explore the connection with *Functional Logic Programming* and consider different evaluation strategies to deal with overlapping rules. With *Constraint Handling Rules*, Stuckey and Sulzmann [39] use *Constraint Logic Programming* to implement type classes.

7. Conclusion

Our main contribution is the development of the implicit calculus $\lambda_{\Rightarrow}$. 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, $\lambda_{\Rightarrow}$ 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 $\lambda_{\Rightarrow}$ offers a wide-range of possibilities for new generic programming applications.

Acknowledgements We are grateful to Ben Delaware, Derek Dreyer, Jeremy Gibbons, Scott Kilpatrick, Phil Wadler, Beta Ziliani, the member of ROPAS and the anonymous reviewers for their comments and suggestions. This work was partially supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant R11-2008-007-01002-0 and the Mid-career Researcher Program (2010-0022061) through NRF grant funded by the MEST. This work was also partially supported by Singapore Ministry of Education research grant MOE2010-T2-2-073.

References

[1] The Boost C++ libraries. http://www.boost.org/, 2010.

[2] C. Camarão and L. Figueiredo. Type inference for overloading without restrictions, declarations or annotations. In *FLOPS*, 1999.

[3] M. Chakravarty, G. Keller, and S. L. Peyton Jones. Associated type synonyms. In *ICFP*, 2005.

[4] M. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. Associated types with class. In *POPL*, 2005.

[5] D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in agda. In *ICFP*, 2011.

[6] A. Dijkstra and S. D. Swierstra. Making implicit parameters explicit. Technical report, Utrecht University, 2005.

[7] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *POPL '06*, pages 295–308, 2006.

[8] D. Dreyer, R. Harper, M. Chakravarty, and G. Keller. Modular type classes. In *POPL*, 2007.

[9] R. Garcia, J. Jarvi, A. Lumsdaine, Jeremy Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA*, 2003.

[10] J. Gibbons. Patterns in datatype-generic programming. In *The Fun of Programming, Cornerstones in Computing*. Palgrave, 2003.

[11] D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in c++. In *OOPSLA*, 2006.

[12] R. Hinze and S. L. Peyton Jones. Derivable type classes. *Electronic Notes in Theoretical Computer Science*, 41(1):5 – 35, 2001.

[13] J. Hughes. Restricted data types in Haskell. In *Haskell*, 1999.

[14] P. Jansson and J. Jeuring. Polytypic programming. In *AFP*. Springer-Verlag, 1996.

[15] M. P. Jones. Simplifying and improving qualified types. In *FPCA*, 1995.

[16] M. P. Jones. Type classes with functional dependencies. In *ESOP*, 2000.

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

[18] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In *Haskell Workshop*, 2001.

[19] R. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP Congress*, 1974.

[20] R. Kowalski, Donald, and Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2, 1971.

[21] R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP*, 2005.

[22] J. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: dynamic scoping with static types. In *POPL*, 2000.

[23] D. Musser and A. Stepanov. Generic programming. In *Symbolic and algebraic computation: ISSAC 88*, pages 13–25. Springer, 1988.

[24] D. R. Musser and A. Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., 1995.

[25] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. Functional logic overloading. In *POPL*, 2002.

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

[27] M. Odersky. The Scala language specification, version 2.8, 2010.

[28] B. C. d. S. Oliveira and J. Gibbons. Scala for generic programmers. *Journal of Functional Programming*, 20, 2010.

[29] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.

[30] B. C. d. S. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. Extended report: The implicit calculus. `http://arxiv.org/abs/1203.4499`, 2012.

[31] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Haskell*, 2008.

[32] T. Schrijvers, S. L. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP*, 2008.

[33] T. Schrijvers, S. L. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP*, 2009.

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

[35] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *PLDI*, 2005.

[36] J. G. Siek and A. Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5), 2011.

[37] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, 2008.

[38] B. Stroustrup. Simplifying the use of concepts. Technical report, Technical Report N2906, ISO/IEC JTC 1 SC22 WG21, 2009.

[39] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *ICFP*, 2002.

[40] V. Trifonov. Simulating quantified class constraints. In *Haskell*, 2003.

[41] D. Vytiniotis, S. L. Peyton Jones, T. Schrijvers, and M. Sulzmann. OUTSIDEIN(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4–5):333–412, 2011.

[42] P. L. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, 1989.