Implicit Parameters: Dynamic Scoping with Static Types

Jeffrey R. Lewis* Mark B. Shields* Erik Meijer† John Launchbury*

*Oregon Graduate Institute of Science & Technology
†University of Utrecht

Abstract

This paper introduces a language feature, called *implicit parameters*, that provides dynamically scoped variables within a statically-typed Hindley-Milner framework. Implicit parameters are lexically distinct from regular identifiers, and are bound by a special `with` construct whose scope is dynamic, rather than static as with `let`. Implicit parameters are treated by the type system as parameters that are not explicitly declared, but are inferred from their use.

We present implicit parameters within a small call-by-name λ-calculus. We give a type system, a type inference algorithm, and several semantics. We also explore implicit parameters in the wider settings of call-by-need languages with overloading, and call-by-value languages with effects. As a witness to the former, we have implemented implicit parameters as an extension of Haskell within the Hugs interpreter, which we use to present several motivating examples.

1 A Scenario: Pretty Printing

You have just finished writing the perfect pretty printer. It takes as input a document to be laid out, and produces a string.

```
pretty :: Doc -> String
```

You have done the hard part—your code is lovely, concise and modular, and your pretty printer produces output that is somehow even prettier than anything you would bother to do by hand. You're thinking: JFP: Functional Pearl.

But, there are just a few fussy details left.

For example, you were not focusing on the unimportant details, so you hard-coded the width of the display to be `78` characters. The annoying thing is that the check to see if you have exceeded the display width is buried deep within the code.

```
... if i >= 78 then ...
```

It is on line 478 of one thousand lines of code, and it is 5 levels deep in the recursion. You have basically two choices. You can define a global named `width`, and use it on line 478, or you can add an extra parameter to nearly every function in the pretty printer and percolate `width` up through all the levels of recursion. Neither choice is very satisfactory.

All this fuss is especially annoying because the change that you wish to make is conceptually rather small, yet implementing it will require a significant change to the program. What you would really like to do is get the best of both—make the definition parameterized, but not have to thread the additional parameter through all that code. What you would like to use is an *implicit parameter*.

With the system proposed in this paper, you only need to change line 478, the place where the display width is checked (and perhaps a handful of type signatures—this is discussed in Section 5.4). The rest of the pretty printer will remain completely unaffected. The idea is to introduce a parameter to the program whose presence is *inferred*, rather than the programmer having to spell it out everywhere.

To introduce an implicit parameter, we change line 478 as follows:

```
... if i >= ?width then ...
```

The `?` is an annotation on an identifier that indicates an implicit parameter. After this small change, when we ask what the type of `pretty` is again, the answer is now:

```
pretty :: (?width :: Int) => Doc -> String
```

This means that `pretty` is a function from `Doc` to `String` with an *implicit* parameter named `width`, of type `Int`. All we had to do was *use* the implicit parameter, and its presence was inferred.

The most striking difference between implicit and regular explicit parameters is that once an implicit parameter is introduced, it is propagated automatically. In other words, when a function with implicit parameters is called, its implicit parameters are inherited by the caller. If we examine the definition of `pretty`, we find that it is defined in terms of a function `worker`, which is itself implicitly parameterized by `?width`.

```
pretty d = worker d []
worker :: (?width :: Int) =>
          Doc -> [Doc] -> String
```

Without lifting a finger, as we saw by type of `pretty`, the `width` parameter is propagated to become a parameter of `pretty` as well.

If an implicit parameter is used twice in the same context, then the two uses will be merged. Thus, if we used `pretty` twice, to get something twice as pretty, we would still only have one `width` parameter:

```
twice_as_pretty d = pretty d ++ pretty d
twice_as_pretty :: (?width :: Int) =>
                   Doc -> String
```

Implicit parameters are bound using the `with` construct. We can express the original behavior of `pretty`, with the fixed width of `78`, as:

```
pretty with ?width = 78 :: Doc -> String
```

Of course, we did not need to extend the language just to set the display width to `78` in the end. The point is that the *user* is in control of the display width. Maybe their display is only 40 characters wide, or maybe they need, at one point, to halve the display width:

```
less_pretty = pretty with ?width = ?width / 2
less_pretty :: (?width :: Int) => Doc -> String
```

Notice that this means that `with` bindings are not recursive, and thus the implicit parameters can be easily re-bound.

The merging of multiple uses of an implicit parameter in the same scope is not always what you want, but it is easy to work around by renaming. For example, consider laying out two documents side by side, with different display widths.

```
beside x y =
  let lhs = pretty d1 with ?width = ?xwidth
      rhs = pretty d2 with ?width = ?ywidth
  in
      zipConcat (fill ?xwidth (lines lhs))
                (lines rhs)
beside :: (?xwidth :: Int, ?ywidth :: Int) =>
          Doc -> Doc -> String
```

1.1 The rest of the paper

In Section 2, we introduce a type system for implicit parameters, followed in Section 3 by two semantics for implicit parameters. In Section 4 we offer several illuminating examples. Section 5 discusses some of the issues associated with adding implicit parameters to a full language. This is followed in Section 6 by related work, and finally, we close in Section 7 with future directions.

2 Types for Implicit Parameters

We now formalize implicit parameters by presenting a type system and inference algorithm for a small language.

2.1 Syntax and Types

Figure 1 presents the syntax and types of λ^{IP}, a call-by-name λ-calculus with let-bound polymorphism, implicit variables

λ-vars	x, y, z		
`let`-vars	p, q		
Implicit vars	$?x, ?y, ?z$		
Terms	t, u, v	$::=$	$x \mid p \mid ?x$
		$\mid$	$\lambda x.\ t \mid t\ \ u$
		$\mid$	$\texttt{let}\ p = u\ \texttt{in}\ t$
		$\mid$	$t\ \texttt{with}\ ?x = u$
Type vars	α, β		
Types	τ, υ	$::=$	α
		$\mid$	$\upsilon \to \tau$
Schemes	σ, φ	$::=$	$\forall\overline{\alpha}.\ C \Rightarrow \tau$
Contexts	C, D	$::=$	$?x_1{:}\,\tau_1, \ldots, ?x_n{:}\,\tau_n$
			$?x_1, \ldots, ?x_n$ distinct
Type contexts	Γ	$::=$	$x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n$
			$x_1, \ldots, x_n$ distinct

$$\boxed{C;\Gamma \vdash t{:}\,\upsilon}$$

$$\text{MVAR}\quad \frac{x{:}\,\upsilon \in \Gamma}{C;\Gamma \vdash x{:}\,\upsilon}$$

$$\text{PVAR}\quad \frac{p{:}\,\forall\overline{\alpha}.D \Rightarrow \upsilon \in \Gamma \qquad D[\overline{\alpha} \mapsto \overline{\tau}] \subseteq C}{C;\Gamma \vdash p{:}\,\upsilon[\overline{\alpha} \mapsto \overline{\tau}]}$$

$$\text{IVAR}\quad \frac{?x{:}\,\tau \in C}{C;\Gamma \vdash ?x{:}\,\tau}$$

$$\text{APP}\quad \frac{C;\Gamma \vdash t{:}\,\upsilon \to \tau \qquad C;\Gamma \vdash u{:}\,\upsilon}{C;\Gamma \vdash (t\ \ u){:}\,\tau}$$

$$\text{ABS}\quad \frac{C;\Gamma, x{:}\,\upsilon \vdash t{:}\,\tau}{C;\Gamma \vdash (\lambda x.\ t){:}\,\upsilon \to \tau}$$

$$\text{LET}\quad \frac{D;\Gamma \vdash u{:}\,\upsilon \qquad \sigma = gen(D,\Gamma,\upsilon) \qquad C;\Gamma, p{:}\,\sigma \vdash t{:}\,\tau}{C;\Gamma \vdash (\texttt{let}\ p = u\ \texttt{in}\ t){:}\,\tau}$$

$$\text{WITH}\quad \frac{C \backslash ?x, ?x{:}\,\upsilon;\Gamma \vdash t{:}\,\tau \qquad C;\Gamma \vdash u{:}\,\upsilon}{C;\Gamma \vdash (t\ \texttt{with}\ ?x = u){:}\,\tau}$$

Figure 1: Well-typed λ^{IP} terms.

$?x$, and `with` bindings. Syntactically, `with` associates to the left.

The type system is an extension of a standard Hindley-Milner type system. What distinguishes it is primarily the presence of a new context C for implicit parameters. The C context keeps track of the implicit parameters which are used in a given term. In addition, type schemes, used to describe the types of let-bound variables, have been extended to include implicit parameter contexts. This implies that the notion of "generalization", which in traditional Hindley-Milner determines which type variables to quantify over, now also includes the abstraction of implicit parameters. We write $gen(C, \Gamma, \tau)$ to denote the principal type scheme

associated with τ in the context of C and Γ.

$$gen(C, \Gamma, \tau) = \forall \overline{\alpha}.\, C \Rightarrow \tau$$
$$\text{where } \overline{\alpha} = (tvars(C) \cup tvars(\tau)) \setminus tvars(\Gamma)$$

As an aid to presenting our axiomatic semantics in Section 3.2, we choose to make a lexical distinction between `let`-bound variables and λ-bound variables. We use p and q for the former, and x, y, and z for the latter. Such a distinction is unnecessary in practice.

Unlike `let`-bound variables, the bindings in implicit parameter contexts are monomorphic. With `let`, we get to see both what the variable is bound to, and everywhere it is used; hence, we can generalize. Since this is a luxury that we are not afforded with dynamic scoping, we must restrict implicit parameters to be monomorphic. From the type system's perspective, implicit parameters are thus very much like lambda-bound variables, whose binding sites just happen to be far removed from their usages.

The IVAR and WITH rules are the only ones that explicitly add and remove elements from the implicit parameter context. The notation $C \backslash ?x$ denotes C with any binding for $?x$ removed.

You may note that while IVAR corresponds to MVAR and WITH corresponds to a LET writ backwards, there's no corresponding APP or ABS rules for implicit parameters. This, in combination with the fact that $\Rightarrow$ types only appear in type schemes, makes it clear that functions can't take implicitly parameterized arguments. This is for the same reasons that lambda-bound variables are not generalized: we would have to either adandon type inference, or abandon type safety. In section 6.1, we will discuss how this avoids the worst sort of bugs that arise with dynamic scoping in Lisp.

Of the standard rules, PVAR and LET are the only ones that are modified, outside of simply adding the implicit parameter context. The only real change to the PVAR rule is to insist that the (instantiated) implicit parameters of the variable must be included in the implicit parameter context of the judgement.

The point to note about the LET rule is that the dynamic context D, used in the judgement on u, is completely independent of the dynamic context C used in the consequent. The independence of D and C assures us that all implicit parameters arising in u end up being associated with x.

2.2 Principal Types

We inherit from Hindley-Milner the problem that typings for terms are not unique. Fortunately, our extension preserves the property that terms have principal types, i.e. a unique type that best represents the type for that term.

In Hindley-Milner, a polymorphic term may be given an arbitrarily more specific type than its most general typing. With implicit parameters, we similarly allow a term to be given more implicit parameters than it actually uses. For example, two valid typings of the constant 1 would be `Int` and $(?x{:}\texttt{Bool}) \Rightarrow \texttt{Int}$.

We extend the notion of "more general" type in Hindley-Milner to mean, in addition, "with fewer implicit parameters". That is, $C \Rightarrow \tau$ is more general than $D \Rightarrow \upsilon$, written $C \Rightarrow \tau \preceq D \Rightarrow \upsilon$, if $\exists \theta.\ \theta\, C \subseteq D \wedge \theta\tau \preceq \upsilon$.

$$\boxed{\theta;\, C; \Gamma \ \vdash\ t{:}\,\tau}$$

MVAR
$$\frac{x{:}\,\upsilon \in \Gamma}{\texttt{id};\, \cdot; \Gamma \ \vdash\ x{:}\,\upsilon}$$

PVAR
$$\frac{p{:}\,\forall \overline{\alpha}.\, C \Rightarrow \upsilon \in \Gamma \qquad \overline{\beta} \text{ new}}{\texttt{id};\, C[\overline{\alpha} \mapsto \overline{\beta}]; \Gamma \ \vdash\ p{:}\,\upsilon[\overline{\alpha} \mapsto \overline{\beta}]}$$

IVAR
$$\frac{\alpha \text{ new}}{\texttt{id};\, ?x{:}\,\alpha; \Gamma \ \vdash\ ?x{:}\,\alpha}$$

APP
$$\frac{\begin{array}{c}\theta_1;\, C_1; \Gamma \ \vdash\ t{:}\,\upsilon_1 \to \tau \\ \theta_2;\, C_2; \Gamma \ \vdash\ u{:}\,\upsilon_2 \qquad (C, \theta_3) = mgu(C_1, C_2) \\ \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3 \sqcup mgu(\upsilon_1, \upsilon_2)\end{array}}{\theta;\, \theta\, C; \Gamma \ \vdash\ (t\ \ u){:}\,\theta\tau}$$

ABS
$$\frac{\theta;\, C; \Gamma, x{:}\,\alpha \ \vdash\ t{:}\,\tau \qquad \alpha \text{ new}}{\theta;\, C; \Gamma \ \vdash\ (\lambda x.\ \ t){:}\,\theta\alpha \to \tau}$$

LET
$$\frac{\begin{array}{c}\theta_1;\, D; \Gamma \ \vdash\ u{:}\,\upsilon \qquad \sigma = gen(D, \theta_1\Gamma, \upsilon) \\ \theta_2;\, C; \Gamma, p{:}\,\sigma \ \vdash\ t{:}\,\tau \qquad \theta = \theta_1 \sqcup \theta_2\end{array}}{\theta;\, \theta\, C; \Gamma \ \vdash\ (\texttt{let}\ \ p = u\ \ \texttt{in}\ \ t){:}\,\theta\tau}$$

WITH
$$\frac{\begin{array}{c}\theta_1;\, C_1; \Gamma \ \vdash\ t{:}\,\tau \qquad \alpha \text{ new} \\ \text{if } ?x{:}\,\upsilon' \in C_1 \text{ then } \upsilon_1 = \upsilon' \text{ else } \upsilon_1 = \alpha \\ \theta_2;\, C_2; \Gamma \ \vdash\ u{:}\,\upsilon_2 \\ (C, \theta_3) = mgu(C_1 \backslash ?x,\, C_2) \\ \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3 \sqcup mgu(\upsilon_1, \upsilon_2)\end{array}}{\theta;\, \theta\, C; \Gamma \ \vdash\ (t\ \ \texttt{with}\ \ ?x = u){:}\,\theta\tau}$$

Figure 2: Type Inference for λ^{IP} terms.

2.3 Type Inference

A type inference algorithm for implicit parameters is given in Figure 2. It is presented in the deductive style of Remy [15]. An invocation is written as follows, where a down-arrow indicates an input to the algorithm, and an up-arrow indicates an output.

$$\overset{\uparrow}{\theta};\ \overset{\uparrow}{C};\ \overset{\downarrow}{\Gamma} \vdash \overset{\downarrow}{t}{:}\overset{\uparrow}{\tau}$$

The result of the algorithm is the principal implicitly-parameterized type $C \Rightarrow \tau$ of t.

We write $mgu(\tau_1, \tau_2)$ to denote the most general idempotent unifier of types τ_1 and τ_2. On implicit parameter contexts, we write $mgu(C_1, C_2)$ to denote the pair (θ, C), where C is the smallest context containing both $\theta\, C_1$ and $\theta\, C_2$, and θ is defined as follows.

$$\theta = \bigsqcup \{ mgu(\tau_1, \tau_2) \ \mid\ ?x{:}\,\tau_1 \in C_1 \ \wedge\ ?x{:}\,\tau_2 \in C_2 \}$$

As usual, substitutions form a semi-lattice on $\preceq$, where $\theta_1 \preceq \theta_2$ if $\exists \theta'.\ \ \theta'\theta_1 = \theta_2$, and we write $\theta_1 \sqcup \theta_2$ to denote the least upper bound of two substitutions in this lattice. It is understood that a type inference derivation is impossible should any of these operations be undefined.

The relationship between the type system and the inference algorithm is expressed by the following two theorems.

Theorem 1 *Soundness*

$$\theta; C; \Gamma \vdash t : \tau \;\Rightarrow\; C; \theta\Gamma \vdash t : \tau$$

Theorem 2 *Completeness*

$$\begin{array}{rl} C; \theta\Gamma \vdash t : \tau \;\Rightarrow\; & \exists \theta', C', \tau', \theta''. \\ & \theta'; C'; \Gamma \vdash t : \tau' \;\wedge \\ & C \supseteq \theta'' C' \;\wedge\; \tau = \theta''\tau' \\ & \theta\Gamma = \theta''\theta'\Gamma \end{array}$$

Proof proceeds by induction on the structure of derivations, coupled with the usual tedious reasoning with type substitutions.

Note that Theorem 2 implies that $C' \Rightarrow \tau' \preceq C \Rightarrow \tau$. Since this holds for all possible types τ of t, our algorithm yields t's principal type.

3 Semantics

In this section we develop an axiomatic semantics for λ^{IP} which is suitable for program transformations. Rather than continue by developing this into an operational semantics, we instead present a type-directed translation of well-typed λ^{IP} terms into a call-by-name λ-calculus with let-bound polymorphism. This translation preserves equality, and forms the basis of our implementation of implicit parameters in Hugs.

3.1 Intuition

Before launching into a detailed axiomatic semantics, let's first test our intuition against some simple examples. To make things interesting we'll assume λ^{IP} has been extended with integers and addition in the obvious way.

First, let's explore the interaction of `with` and `let`. Consider

$$\begin{array}{l} (\texttt{let}\ \ p = ?y + 2\ \ \texttt{in}\ \ p + (p\ \ \texttt{with}\ \ ?y = 1)) \\ \quad \texttt{with}\ \ ?y = 2 \end{array}$$

Does this equal 7 or 8? Recall that the LET rule in Figure 1 requires that p be generalized over its implicit parameter $?y$. This means each occurrence of p should be evaluated with its own local environment. Hence we may rewrite this example to

$$\begin{array}{l} \texttt{let}\ \ p = ?y + 2 \\ \texttt{in}\ \ (p + (p\ \ \texttt{with}\ \ ?y = 1))\ \ \texttt{with}\ \ ?y = 2 \end{array}$$

which may be further simplified to

$$\begin{array}{l} \texttt{let}\ \ p = ?y + 2 \\ \texttt{in}\ \ (p\ \ \texttt{with}\ \ ?y = 2) + (p\ \ \texttt{with}\ \ ?y = 1) \end{array}$$

Now expand the definition of p

$$(?y + 2\ \ \texttt{with}\ \ ?y = 2) + (?y + 2\ \ \texttt{with}\ \ ?y = 1)$$

the we see the correct result is 7.

Abstracting from the particulars we see that

$$\begin{array}{c} (\texttt{let}\ \ y = v\ \ \texttt{in}\ \ t)\ \ \texttt{with}\ \ ?x = u \\ = \\ \texttt{let}\ \ y = v\ \ \texttt{in}\ \ (t\ \ \texttt{with}\ \ ?x = u) \end{array}$$

Note in particular that

$$\begin{array}{c} (\texttt{let}\ \ y = v\ \ \texttt{in}\ \ t)\ \ \texttt{with}\ \ ?x = u \\ \neq \\ \texttt{let}\ \ y = (v\ \ \texttt{with}\ \ ?x = u)\ \ \texttt{in}\ \ (t\ \ \texttt{with}\ \ ?x = u) \end{array}$$

as otherwise our implicit parameters would simply be static variables!

What about application and implicit variables? Consider

$$\begin{array}{l} ((\lambda x.\ \texttt{let}\ \ p = ?y + x \\ \qquad \texttt{in}\ \ (p + x\ \ \texttt{with}\ \ ?y = 1))\ \ (?y + 2)) \\ \qquad \texttt{with}\ \ ?y = 2 \end{array}$$

Does this equal 9 or 7? Again, consulting Figure 1, we see the ABS rule requires x to be a monotype. Thus, when substituting the argument $?y+2$ for the variable x, we must take care to ensure $?y + 2$ uses the binding $?y = 2$ rather than $?y = 1$. If we don't substitute for $?y$ immediately, we must *reroute* this correct binding via a fresh implicit variable $?z$. Doing so yields

$$\begin{array}{l} (\texttt{let}\ \ p = ?y + (?y + 2\ \ \texttt{with}\ \ ?y = ?z) \\ \ \texttt{in}\ \ (p + (?y + 2\ \ \texttt{with}\ \ ?y = ?z)\ \ \texttt{with}\ \ ?y = 1)) \\ \quad \texttt{with}\ \ ?z = ?y\ \ \texttt{with}\ \ ?y = 2 \end{array}$$

Now we can propagate the renaming of $?y$ to $?z$

$$\begin{array}{l} (\texttt{let}\ \ p = ?y + ?z + 2) \\ \ \texttt{in}\ \ p + ?z + 2\ \ \texttt{with}\ \ ?y = 1)) \\ \quad \texttt{with}\ \ ?z = ?y\ \ \texttt{with}\ \ ?y = 2 \end{array}$$

propagate the binding $?y = 2$ and hence $?z = 2$

$$\begin{array}{l} (\texttt{let}\ \ p = ?y + ?z + 2) \\ \ \texttt{in}\ \ p + 2 + 2\ \ \texttt{with}\ \ ?y = 1\ \ \texttt{with}\ \ ?z = 2)) \end{array}$$

expand p

$$?y + ?z + 2 + 2 + 2\ \ \texttt{with}\ \ ?y = 1\ \ \texttt{with}\ \ ?z = 2$$

and the correct answer is 9.

3.2 Axiomatic Semantics

Figure 3 presents the β-, η- and α-rules for well-typed λ^{IP} terms. The β-rules are defined in terms of three substitution operators, one for each binding form, and hence variable class.

First, consider β-reduction for λ-abstractions. We write $t[x \mapsto u]$ to denote substituting u for x in t. Since x is a λ-bound variable, it must be monomorphically typed. Thus, though u may contain implicit parameters, they should be provided by the surrounding context, and not from within t. That is, just as we must take care to avoid *static* name capture by *renaming* bound variables, we must also avoid *dynamic* name capture by *rebinding* implicit variables.

This rebinding is most obvious in the definition of $(t\ \ \texttt{with}\ \ ?y = v)[x \mapsto u]$, where we have taken care to ensure that if u has an implicit parameter $?y$, its binding will bypass the binding of $?y$ to v via the fresh implicit variable $?z$.

The case for $(\texttt{let}\ \ q = v\ \ \texttt{in}\ \ t)[x \mapsto u]$ is similar, though this time we must bypass *all* the implicit parameters of u,

β rules

$$(\lambda x.\ t)\ u \ = \ t\,[\,x \mapsto u\,]$$
$$\texttt{let}\ p = u\ \texttt{in}\ t \ = \ t\,[\,p \mapsto u\,]$$
$$t\ \texttt{with}\ ?x = u \ = \ t\,[\,?x \mapsto u\,]$$

η rules

$$\lambda x.\ t\ x \ = \ t$$
$$\texttt{let}\ p = t\ \texttt{in}\ p \ = \ t$$
$$?x\ \texttt{with}\ ?x = t \ = \ t$$

α rules

$$\lambda x.\ t \ = \ \lambda y.\ t\,[\,x \mapsto y\,]$$
$$\texttt{let}\ p = u\ \texttt{in}\ t \ = \ \texttt{let}\ q = u\ \texttt{in}\ t\,[\,p \mapsto q\,]$$

(plus congruent closure)

Substitution for λ-vars

$$x\,[\,x \mapsto u\,] \ = \ u$$
$$y\,[\,x \mapsto u\,] \ = \ y$$
$$p\,[\,x \mapsto u\,] \ = \ p$$
$$?x\,[\,x \mapsto u\,] \ = \ ?x$$
$$(\lambda y.\ t)\,[\,x \mapsto u\,] \ = \ \lambda y.\ t\,[\,x \mapsto u\,] \quad \text{where } y \notin \mathit{fvars}(u)$$
$$(t\ v)\,[\,x \mapsto u\,] \ = \ (t\,[\,x \mapsto u\,])\ (v\,[\,x \mapsto u\,])$$
$$(\texttt{let}\ q = v\ \texttt{in}\ t)\,[\,x \mapsto u\,] \ = \ \texttt{let}\ q = v\,[\,x \mapsto u\ \overline{\texttt{with}\ ?y = ?z}\,]\ \texttt{in}\ (t\,[\,x \mapsto u\,]\ \overline{\texttt{with}\ ?z = ?y})$$
where $q \notin \mathit{fvars}(u)$, $C; \Gamma \vdash u : v$, $\overline{?y} = \mathit{vars}(C)$, $\overline{?z}$ fresh

$$(t\ \texttt{with}\ ?y = v)\,[\,x \mapsto u\,] \ = \ t\,[\,x \mapsto u\ \texttt{with}\ ?y = ?z\,]\ \texttt{with}\ ?y = v\,[\,x \mapsto u\,]\ \texttt{with}\ ?z = ?y \quad \text{where } ?z \text{ fresh}$$

Substitution for let-vars

$$(\texttt{let}\ q = v\ \texttt{in}\ t)\,[\,p \mapsto u\,] \ = \ \texttt{let}\ q = v\,[\,p \mapsto u\,]\ \texttt{in}\ t\,[\,p \mapsto u\,] \quad \text{where } q \notin \mathit{fvars}(u)$$
$$(t\ \texttt{with}\ ?y = v)\,[\,p \mapsto u\,] \ = \ t\,[\,p \mapsto u\,]\ \texttt{with}\ ?y = v\,[\,p \mapsto u\,]$$
(remaining cases as for λ-vars)

Substitution for implicit vars

$$x\,[\,?x \mapsto u\,] \ = \ x$$
$$p\,[\,?x \mapsto u\,] \ = \ p\ \texttt{with}\ ?x = u$$
$$?x\,[\,?x \mapsto u\,] \ = \ u$$
$$?y\,[\,?x \mapsto u\,] \ = \ ?y$$
$$(\lambda y.\ t)\,[\,?x \mapsto u\,] \ = \ \lambda y.\ t\,[\,?x \mapsto u\,] \quad \text{where } y \notin \mathit{fvars}(u)$$
$$(t\ v)\,[\,?x \mapsto u\,] \ = \ (t\,[\,?x \mapsto u\,])\ (v\,[\,?x \mapsto u\,])$$
$$(\texttt{let}\ q = v\ \texttt{in}\ t)\,[\,?x \mapsto u\,] \ = \ \texttt{let}\ q = v\ \texttt{in}\ t\,[\,?x \mapsto u\,] \quad \text{where } q \notin \mathit{fvars}(u)$$
$$(t\ \texttt{with}\ ?y = v)\,[\,?x \mapsto u\,] \ = \ t\,[\,?x \mapsto u\,]\ \texttt{with}\ ?y = v\,[\,?x \mapsto u\,]$$
$$(t\ \texttt{with}\ ?x = v)\,[\,?x \mapsto u\,] \ = \ t\ \texttt{with}\ ?x = v\,[\,?x \mapsto u\,]$$

Figure 3: Axiomatic semantics for λ^{IP}.

lest they become captured by q. Notice that this rule *will change the type of* q! In particular, q will now depend on the fresh implicit variables $\overline{?z}$, which are bound by the `with` surrounding t. However, the overall term's type remains unchanged.

Perhaps surprisingly, the rules for β-reduction of a `let` term perform no such rebinding. This becomes obvious once we recall that a let-bound term is always generalized on all of its implicit parameters. Hence $t\,[\,p \mapsto u\,]$ is essentially λ-calculus substitution.

Finally, the rules for β-reduction of `with` terms illustrate how a dynamic environment is propagated into sub-terms. Since implicit variables are lexically distinct from static variables, there is never a danger of a name capture, and so $(\lambda y.\ t)\,[\,?x \mapsto u\,]$ need not rename y. Furthermore, since implicit parameters are immutable, the dynamic environment need not be threaded state-like through the program. Thus in $(t\ \ v)\,[\,?x \mapsto u\,]$, we simply propagate u into both t and v.

The most interesting rule comes with `let`. In $(\texttt{let}\ \ q = v\ \ \texttt{in}\ \ t)\,[\,?x \mapsto u\,]$, v has been generalized. Hence, if v depends on the implicit variable $?x$, the implicit binding must be resolved for each occurrence of q in t, and may be unrelated to the binding of $?x$ to u.

3.3 Translation Semantics

We also present a type-directed translation of λ^{IP} terms into the familiar call-by-name λ-calculus with let-bound polymorphism and tuples. As well as providing another semantics, this translation provides a convenient mechanism for adding implicit parameters to an existing language.

Our translation borrows the technique of *dictionary passing* [16], used to give a semantics for overloading in Haskell. In short, we encode $C \Rightarrow \tau$ as $C \to \tau$, and treat implicit parameter contexts as tuples of explicit parameters.

Figure 4 presents the translation, which is by induction over a type derivation using the rules of Figure 1. The tuples d which show up in the VAR and LET rules arise from making the implicit parameters explicit. Note that in the target language, $?x$ is just an ordinary variable—the ? is retained to ensure introduced identifiers are not confused with those already in the source program.

Since a given term may be well-typed by more than one type derivation, we must address the question of whether the translation is coherent. For example, the term

$$\texttt{let}\ p = 1\ \texttt{in}\ p + 2$$

could be translated to

$$\texttt{let}\ p = 1\ \texttt{in}\ p + 2$$

should we choose the principle type for p, or to

$$\texttt{let}\ p = \lambda ?y.\ 1\ \texttt{in}\ p\ ?y + 2$$

should we choose a more specific type for p. However, since the only thing that arises is additional unused parameters such as $?y$, it is not difficult to see that there's no loss of coherence.

Our axiomatic semantics is sound with respect to our translation semantics. We write $[\![t]\!]$ to denote the t' such that

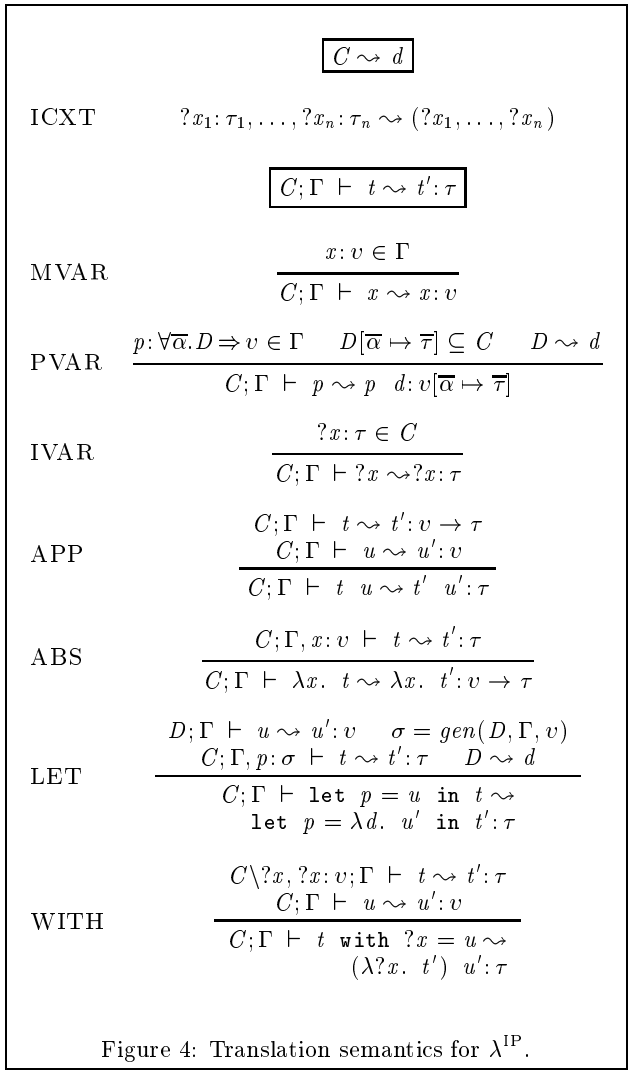

Figure 4: Translation semantics for λ^{IP}.

$C \mid \Gamma \vdash t \rightsquigarrow t' : \tau$, where C, Γ and τ are implied by context, and write $t'\,[\,u'/x\,]$ to denote ordinary λ-calculus substitution.

Lemma 3

$$\begin{array}{rcl} t\,[\,x \mapsto u\,] = v & \Rightarrow & [\![t]\!]\,[\,[\![u]\!]/x\,] = [\![v]\!] \\ t\,[\,p \mapsto u\,] = v & \Rightarrow & [\![t]\!]\,[\,\lambda d.[\![u]\!]/p\,] = [\![v]\!] \\ \multicolumn{3}{c}{\text{where } D; \Gamma \vdash u : v \text{ and } D \rightsquigarrow d} \\ t\,[\,?x \mapsto u\,] = v & \Rightarrow & [\![t]\!]\,[\,[\![u]\!]/?x\,] = [\![v]\!] \end{array}$$

Theorem 4

$$t = u \Rightarrow [\![t]\!] = [\![u]\!]$$

All these proofs proceed by straightforward induction on the structure of t.

4 Examples

So far we have presented the implicit parameter system as an extension of a simple Hindley-Milner typed lambda calculus. However, in practice it also integrates nicely with full languages, particularly Haskell. We have demonstrated this by extending the Hugs interpreter to include implicit parameters. As suggested in Section 3.3, the implementation leverages off of the existing type class mechanism, with implicit parameters as a new kind of type predicate. The resulting system is available in the distribution of Hugs 98 [8].

The chief advantage of having a real implementation is the development of real examples. These enabled us to explore whether the system we were exploring was merely a curiosity, or one which has real practical potential.

The rest of this section contains a variety of illustrative examples.

4.1 Auxiliary parameters in recursive definitions

In recursive function definitions, there are often parameters that don't change between recursive calls. Out of convenience, or as a small nod to efficiency, these definitions are often factored into a worker-wrapper arrangement, where the worker (the recursive definition) is written as a local function that is not explicitly parameterized over any of these auxiliary parameters. For example:

```
append :: [a] -> [a] -> [a]
append xs ys = prepend xs
  where prepend (x:xs) = x : prepend xs
        prepend [] = ys
```

Unfortunately, this has the side effect of hiding the function that is doing all the work (`prepend`) from the rest of the program. Sometimes this is quite unnecessary, and can be a considerable inconvenience. For example, we cannot access `prepend` outside of the body of `append` either to examine its type, or test it directly. Even worse, because standard Haskell lacks the ability to express scoped type variables, we cannot even give a type signature for `prepend`!

Using implicit parameters, we can factor the definition in the same way, but `prepend` need not be hidden inside the definition of `append` anymore.

```
append :: [a] -> [a] -> [a]
append xs ys = prepend xs with ?ys = ys

prepend :: (?ys :: [a]) => [a] -> [a]
prepend (x:xs) = x : prepend xs
prepend [] = ?ys
```

Now we can give a proper type to `prepend`, and reuse and test it in isolation from `append`. This is obviously a trivial example, but the principle scales naturally particularly when writing monadic code, where a common pattern is to generate state components, and then pass the references to recursive worker code. Being able to pass the state context implicitly simplifies the body of the code.

To make this concrete, consider the following depth-first traversal routine [9]. In this case, we actually pass the *procedures* for accessing the state, rather than the array itself.

```
data Rose a = Node a [Rose a]

dfs :: Graph -> [Vertex] -> [Rose Vertex]
dfs g vs = runST (
```

```
  do {arr <- newSTArray (bounds ?g) False;
      dfsLoop vs
      with ?g = g
           ?marked = readSTArray arr
           ?mark = \v -> writeSTArray arr v True })

dfsLoop [] = return []
dfsLoop (v:vs)
 = do {b <- ?marked v;
       if b then dfsLoop vs else
   do {?mark v;
       ps <- dfsLoop (children ?g v);
       qs <- dfsLoop vs;
       return ((Node v ps) : qs)
   } }
```

The auxilliary function `dfsLoop` looks as if it were defined in a local definition, but it is a top level function and typable in its own right:

```
dfsLoop :: (?g :: Graph,
            ?marked :: Vertex -> ST s Bool,
            ?mark :: Vertex -> ST s ())
      =>
           [Vertex] -> ST s [Rose Vertex]
```

4.2 Environments

When writing shell scripts, many of the parameters to the script are passed in the environment. This is not simply because of the paucity of shell scripting languages, but rather that the environment variables form a moderately stable context for the execution of the script. It is similar with GUI code. The large graphics context (`gc`) contains all the relevant windowing information, including the default font, color, and size, for example. Much of the time, the `gc` remains unchanged, until, for example, some text needs to be written in a different color. The `gc` color is changed, the text is written, and the color is changed back.

Typical shell environments are represented as lists of pairs. We can implement shell environments using implicit parameters as follows:

```
type Environment = [(String,String)]

getEnv :: (?env :: Environment) =>
          String -> String
getEnv var = case lookup ?env var of
               Nothing -> ""
               Just val -> val
```

Although shells also typically provide a way to change the environment by side effect, the far more common idiom is to make changes to the environment that only scope over sub-processes, but do not propogate forward. This idiom can be naturally mimicked using implicit parameters. Consider a script that called a program which needed a different environment (the search path to be ordered differently, core size changed, and so on).

```
setEnv :: (?env :: Environment) =>
          String -> String -> Environment
setEnv v w = update ?env
  where
      update [] = [(v,w)]
      update ((a,b):abs)
       = if a==v then (a,w):abs
         else (a,b):update abs
```

This might be used as in the following example:

```
foo x path
  = (getEnv "PATH",
     baz x with ?env=setEnv "PATH" path,
     bar x)
```

The first and third components of the tuple have access to the current value of `PATH`, but the call to `baz` is in the context of `PATH` being bound to the contents of `path`.

Numerical methods provide another example where environments are useful. Here the environment is likely to contain parameters to control factors such as desired accuracy (ϵ), what response to use to ill-conditioned problems, and so on.

4.3 File IO

When doing file I/O in Haskell, the programmer is forced to carry about file handles. This adds quite a bit of clutter. Using implicit parameters, we can model functionally the nice situation in C, where there's a notion of standard input and output streams, and a given stream can easily be redirected to another stream.

The Haskell IO library provides primitives like `getLine` and `putStr` that follow this convention, but provide no easy way to redirect. Using implicit parameters, we could redefine `getLine` and `putStr` as follows.

```
getLine :: (?stdIn :: FileHandle) => IO String
putStr :: (?stdOut :: FileHandle) =>
          String -> IO ()
```

Using these, we define a simple session:

```
session :: (?stdIn  :: FileHandle,
            ?stdOut :: FileHandle) => IO ()
session =
  do  putStr "What is your name?\n"
      s <- getLine
      putStr ("Hello, " ++ s ++ "!\n")
```

If we postulate a mechanism that binds `stdIn` and `stdOut` at the top-level to their respective defaults (a top-level `with` declaration, for example) then, by default, `getLine` and `putStr` would behave exactly as in Haskell. However, by using `with`, the programmer can easily redirect `stdOut` elsewhere, without having to change the `session` code at all:

```
do  h <- openFile "foo"
    session with ?stdOut = h
```

4.4 Linking Haskell and Java via JNI

The Java Native Interface (JNI) allows a two-way integration between Java and native code, programs written in other languages such as C or Haskell [10]. On the native side, the reflection of a Java method has two extra parameters, the `JNIEnv` pointer and the `jobject` pointer. The `JNIEnv`

pointer is a handle to a virtual method table through which native methods can access parameters and objects in Java. The `jobject` pointer is the `this` variable in Java.

Consider the following simple class that has a native method that is supposed to display a prompt and return the user's response:

```
class HaskellPrompt {
  String prompt;
  native String getLine();
}
```

This method can be implemented in Haskell using the function `getLine :: JNIEnv -> Jobject -> IO JString` that as explained above gets two additional arguments, the environment pointer of type `JNIEnv` and the this pointer of type `Jobject` to the `HaskellPrompt` instance on which the `getLine` method is called.

In order to display the prompt we have to fetch its content from the `prompt` field of the the object, and marshal its value to a proper Haskell string. We then display it, read the user's response, and unmarshal it back into a Java string that is returned as the result of calling `getLine`.

The actual details in interacting with Java via JNI are rather painful. To read the field a Java object, we first have to get the `fieldID` from the the class reference of the object, the field name and the field type using the `JNIEnv` entry

```
getFieldID :: JNIEnv -> Jclass ->
              String -> String -> IO FieldId
```

and then read its value via:

```
getObjectField :: JNIEnv -> Jobject ->
                  FieldID -> IO Jobject
```

We can get the class reference of an object via the `JNIEnv` entry:

```
getObjectClass :: JNIEnv -> Jobject -> IO Jclass
```

The functions `getStringUTFChars` and `newStringUTF` are entries in the `JNIEnv` method table that translate between Java and Haskell strings.

By calling all these functions in the right order, passing each one of them the `JNIEnv` pointer, we can implement function `getLine` in Haskell as follows:

```
getLine :: JNIEnv -> Jobject -> IO JString
getLine = \jnienv -> \that ->
  do{ cls <- getObjectClass jnienv that
    ; fid <- getFieldID jnienv cls "prompt"
             "Ljava/lang/String;"
    ; jprompt <- getObjectField jnienv that fid
    ; prompt <- getStringUTFChars jnienv jprompt
    ; putStr prompt; answer <- getLn
    ; newStringUTF jnienv answer
    }
```

Explicitly passing around the `JNIEnv` argument becomes rather tedious; we have to pass the same environment pointer to each call to a JNI primitive. This is where implicit parameters come to the rescue; we just make the `JNIEnv` argument implicit, rather similar to the way we made the environment in section 4.2 implicit.

All the functions in the `JNIEnv` method table get `(jnienv :: JNIEnv)` as an implicit parameter

```
getObjectClass :: (?jnienv :: JNIEnv) =>
  Jobject -> IO Jclass
getFieldID :: (?jnienv :: JNIEnv) =>
  Jclass -> String -> String -> IO FieldId
getObjectField :: (?jnienv :: JNIEnv) =>
  Jobject -> FieldID -> IO Jobject
getStringUTFChars :: (?jnienv :: JNIEnv) =>
  JString -> IO String
newStringUTF :: (?jnienv :: JNIEnv) =>
  String -> IO JString
```

The effect of making the `jnienv` implicit in the JNI primitives is that all functions which use them automatically get the `jnienv` as an implicit argument as well:

```
getLine :: (?jnienv :: JNIEnv) =>
           Jobject -> IO JString
getLine = \that ->
  do{ cls <- getObjectClass that
    ; fid <- getFieldID cls
             "prompt" "Ljava/lang/String;"
    ; jprompt <- getObjectField that fid
    ; prompt <- getStringUTFChars jprompt
    ; putStr prompt; answer <- getLn
    ; newStringUTF answer
    }
```

By using the expressive power of implicit parameters, we have been able to abstract from the irrelevant details of passing around the `JNIEnv` pointer. The resulting code is of a conciseness that is difficult to achieve when working n C or C++.

5 Implicit Parameters In-the-large

We now discuss some of the more subtle language design issues associated with adding implicit parameters to a full language such as Haskell or ML.

5.1 Call-by-need Languages

Call-by-need languages share the computational cost of evaluating let-bound terms by *updating*. For example

```
        let x = fib 10 in (x, x)
  ⟶  let x = 55 in (55, x)
  ⟶  let x = 55 in (55, 55)
```

Now, consider a let-bound term defined using implicit parameters:

```
      (let x = fib ?y in (x, x)) with ?y = 10
⟶  let x = fib ?y
     in (x with ?y = 10, x with ?y = 10)
⟶  let x = fib ?y
     in ((fib ?y) with ?y = 10,
         (fib ?y) with ?y = 10)
⟶  let x = fib ?y in (fib 10, fib 10)
```

Clearly, the cost of evaluating `fib 10` will not be shared.

The problem here is not one of semantics: clearly a let-bound term with implicit parameters can only be fully evaluated when all such parameters have been supplied. In other

words, such a term is a *value*, and there's no computational cost to share. Rather, the problem is that a programmer is accustomed to being able to distinguish a value from a computation by looking at its *syntax* alone, whereas in our system the *type* is also important.

The designers of Haskell [14] also encountered this subtlety. A let-bound term which contains unresolved overloading is also a value as far as sharing is concerned, but this can only be determined by knowing the term's type. Their solution was to introduce the *monomorphism restriction*: a let-bound term that looks like a computation must not be generalized.

What would be the effect of adopting this restriction with implicit parameters? Consider the following:

```
let x = z + 2
    z = 2 * ?y
in (x with ?y = 1) + x
with ?y = 2
```

Since `x` looks like a computation, the monomorphism restriction would kick in, and `x` would not be generalized. The result is that `?y` would be statically bound in `x` with the binding `?y = 2`, in stark disagreement with our axiomatic semantics. Thus, in an effort to preserve sharing, we have altered our language semantics—hardly a happy situation.

We conclude that, in the presence of implicit parameters, the monomorphism restriction is the wrong solution to the sharing problem. To address sharing, the programmer must be given either knowledge or control, and not be subject to editorial distinctions that the language designers might make. To give the programmer knowledge, we suggest that languages with implicit parameters (and, for that matter, Haskell-style overloading) need programming environments in which type information is immediately available to the programmer—even whilst editing! Such environments make distinguishing values from computations trivial. Alternately, we could give the programmer more control by providing two versions of `let`—a non-generalizing one which promises sharing, and a generalizing one which doesn't. In this way, the type checker can validate the programmer's intuition with regards to sharing.

5.2 Call-by-value Languages

There is no technical difficulty in adding implicit parameters to a call-by-value language. The translation of Figure 4 may be used unchanged, though the axiomatic semantics of Figure 3 must be weakened as usual.

Things become interesting when we consider an ML-like language with side-effects. Consider

```
let x = print (fib 10) in (x, x)
```

which will output "`55`" once. Now, if we were to add an implicit parameter

```
let x = print (fib ?y) in (x, x)) with ?y = 10
```

evaluation will output "`55`" twice. So the timing of the effect depends on its type: once at `let` binding time if it is free of implicit parameters, versus once per each use of `x` if it contains implicit parameters.

This is exactly the same problem as for call-by-need above! In call-by-need the lazy programmer was "surprised" by a duplication of work. In call-by-value the eager programmer was "surprised" by a duplication of side-effects. But again, there is no surprise if the programmer knows the type of `x`.

ML has its own version of Haskell's monomorphism restriction, namely the value restriction, although its motivation is to prevent a loss of type-soundness [12]. Since any term requiring implicit parameters is semantically a value, this restriction may be somewhat relaxed in our system without compromising soundness.

5.3 Haskell-style Overloading

In Section 3.3, we saw that our translation, which turns implicit parameters into explicit parameters, is based on the dictionary translation, which turns overloading into explicit dictionary passing [16]. Thus, Haskell already has a form of anonymous implicit parameters, and as a pleasant consequence, implicit parameters and Haskell-style overloading coexist happily. This is witnessed by our implementation of implicit parameters within Hugs.

The work presented here is the first half of a larger research programme to de-construct the complex type class system of Haskell into simpler, orthogonal language features [11]. This paper elevates the dictionary translation into a self-contained language feature, rather than just a semantics for type classes left "under the hood."

Can we replace Haskell type classes with just implicit parameters alone? Almost! Following the original proposal for type classes [16], we can encode class declarations as record types, and instance declarations as values. For example, consider the standard `Functor` class:

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
```

We can encode the class itself as a datatype:

```
data Functor f = Functor
  { map_ :: forall a, b.
            (a -> b) -> (f a -> f b) }
```

Then we introduce the class methods as implicitly parameterized values[1]:

```
map :: forall a, b . (?functor :: Functor f) =>
       (a -> b) -> (f a -> f b)
map = map_ ?functor
```

But the type class system was primarily designed to solve the problem of overloading, i.e. using an identifier at more than one type within the same scope. Consider the following example in Haskell.

```
(map (+ 1) [1, 2, 3], map (+ 1) id)
```

This would have the type `([Int],Int -> Int)`. Unfortunately, implicit parameters come with the constraint that all instances of an implicit parameter must have the same

[1] This example assumes that implicit parameters may be given higher-ranked polymorphic types when sufficient type annotations are provided.

type. Thus, this example would be ill-typed if `map` were implemented using implicit parameters, since `map` is used at two different types.

So clearly our programme remains unfinished.

5.4 Signatures

Another fly in the ointment with respect to implicit parameters has to do with signatures. One of the selling points of implicit parameters is that they provide a very low impact way of adding additional parameters to an existing program. However, consider the program that has been painstakingly annotated with type signatures on most definitions. This is, after all, considered "good style". Unfortunately, while the programmer doesn't need to modify most functions that have become implicitly parameterized, the types of those functions change. Thus, a tedious global change to the type annotations may be required.

Fortunately, this is easily mitigated in a way that is compatible with the use of type signatures, namely by allowing type signatures which only partially constrain the context of a type. An ellipses at the end of a context, for example, could be used to indicate that there may be arbitrary additional context elements that are not constrained. Thus, you might say something like the following to indicate the signature for `pretty` without constraining what might be in its context.

```
pretty :: ... => Doc -> String
```

This solution is also well-suited to Haskell-style overloading, which suffers exactly the same problem.

6 Related Work

6.1 Dynamic Scoping in Lisp

We would be quite remiss if we didn't mention Lisp, which introduced dynamic scoping in the beginning, albeit as a bug that took decades to stamp out. Although most modern Lisps now have static scoping of variables, MIT Scheme, while being statically scoped, has a `fluid-let` construct that dynamically binds variables by side-effect, and takes scrupulous care to ensure that the previous binding is fully-reinstated afterwards [4].

The biggest problem with dynamic scoping in Lisp involved the use of higher-order functions, and is known affectionately as the "downward funarg problem". A function passed as a parameter to another function might, often unintentionally, have its free variables captured by a local environment.

Implicit parameters provide the same functionality as dynamic scoping in Lisp, except that implicitly parameterized functions are not first-class, and thus can't be passed as arguments to functions—implicit parameters always float towards outer contexts, they don't enter inner ones. Thus, we would argue that implicit parameters give you the best of dynamic scoping, while avoiding the its worst pitfalls. Furthermore, this is the first type system that we know of that records the use of dynamic parameters in the types.

6.2 Qualified Types

The system λ^{IP} is very close to the syntax-directed variant of Jones' system OML [5], a formal system designed to capture the essence of, and generalize, Haskell's type classes. Our system differs from OML in two key ways: all instances of an implicit parameter are assumed to be the same, and implicit parameters have a local binding construct, `with`.

In OML, the label on an element of the context is associated with a family of types (the collection of instances), whereas with implicit parameters, it is associated with an individual parameter. Thus, while with type classes, two unconstrained uses of an overloaded construct must be assumed to be different, with implicit parameters they must be assumed to be the same.

The significance of a local binding construct is that it affects the design choices we can make in the LET rule. Our system follows the most conservative route of binding *all* implicit parameters to the `let`-bound variable.

Haskell itself, however, allows some type predicates to escape binding—in particular those whose type depends on type variables bound in Γ. I.e., if we cannot generalize over all type variables in a predicate, that predicate is not bound, and becomes a predicate of the `let` as a whole. Following this approach, we could have chosen the following LET rule.

$$\frac{C \cup D; \Gamma \vdash u : v \qquad \sigma = \mathtt{Gen}(D, \Gamma, v) \qquad \mathrm{TV}(D) \cap \mathrm{TV}(\Gamma) = \emptyset \qquad C; \Gamma, p : \forall \overline{\alpha}. D \Rightarrow v \vdash t : \tau}{C; \Gamma \vdash \mathtt{let}\ p = u\ \mathtt{in}\ t : \tau}$$

However, with this LET rule, our system would not have principal types. Consider the term

```
let p = ?x in (p with ?x = 1)
```

This would have the following two typings: `Int` and $(\mathtt{?x} : \alpha) \Rightarrow \alpha$, depending on whether `p` captured `?x` as an implicit parameter or not (and thus whether the inner `with` binding has any effect or not). Unfortunately, these two types are incomparable, and their lub isn't a type for the term.

This problem doesn't affect Haskell, because there is no local binding mechanism corresponding to `with`. All instance declarations in Haskell are global. The result is that any constraint that arises due to use of a `let`-bound identifier in the body of a `let` will propagate out and become a constraint of the whole `let` anyway.

Since the ideas in this paper were first distributed, Mark Jones has further refined OML to include the notion of *functional dependencies*[7, 6]. The resulting system is a step towards being able to encode both implicit parameters and overloading within a single system.

6.3 Other related work

Odersky, *et al.* [13], proposed a system for overloading where individual identifiers are overloaded instead of whole classes of operators, as in Haskell. Their proposal was intended to overcome a number of difficulties that arise with type classes. Because individual identifiers are overloaded, their type constraints bear a striking similarity to implicit parameter contexts. However, their system is about overloading, and lacks any local binding construct.

The Label-Selective Lambda-Calculus of Garrigue and Kaci [3, 2] also allows both dynamic and static binding to coexist. However, their system requires a change at the very foundation of our languages, namely λ-abstraction. It is unclear as to how such a change would integrate well with existing functional languages.

7 Future Work

Over the past ten years *monads* have become a popular way to provide semantics for many systems, especially those involving effects such as state and exceptions. We were intrigued, therefore, when it appeared that *comonads* were the mathematical structure underlying implicit parameters.

The intuition is as follows: monads model the effect of *performing* a computation, and are thus associated with outputs, or the right-hand sides of semantic type judgments. Comonads, on the other hand, model the structure of *environments*, and are thus associated with inputs, or the left-hand sides of judgements. For example, in separated (multiplicative, intuitionistic) linear logic, comonads show up when modeling the intuitionistic segment of the environment [1]. In our system, comonads are used to model the implicit portion of the environment.

We are currently working on a categorical, comonadic semantics for implicit parameters. Using it, we hope to demonstrate that our translation semantics simply represents the term language of a family of coKleisli categories within the term language of the base category.

References

[1] Benton, N. A mixed linear and non-linear logic: Proofs, terms and models. Tech. Rep. 352, University of Cambridge Computer Laboratory, Oct. 1994.

[2] Garrigue, J. Dynamic binding and lexical binding in a transformation calculus. In *Proc. of the Fuji International Workshop on Functional and Logic Programming.* (1995).

[3] Garrigue, J., and Kaci, H. The Typed Polymorphic Label-Selective Lambda-Calculus. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, 1994).

[4] Hanson, C. *MIT Scheme Reference.* Cambridge, MA: MIT Press, Apr. 96.

[5] Jones, M. P. *Qualified Types: Theory and Practice.* Cambridge University Press, Cambridge, England, 1994.

[6] Jones, M. P. Exploring the design space for type-based implicit parameterization. Tech. rep., Oregon Graduate Institute, July 1999.

[7] Jones, M. P. Type classes with functional dependencies, Oct. 1999. (Submitted for publication).

[8] Jones, M. P., and Peterson, J. C. Hugs 98 user manual. `http://www.haskell.org/hugs/`, May 1999.

[9] King, D. J., and Launchbury, J. Structuring depth-first search algorithms in Haskell. In *ACM Symposium on Principles of Programming Languages* (San Francisco, California, Jan. 1995), pp. 344–354.

[10] Liang, S. *The Java Native Interface (Programmer's guide and Specification).* The Java Series. Addison Wesley, 1999.

[11] Meijer, E., and Claessen, K. The design and implementation of Mondrian. In *Proceedings of the Haskell Workshop* (1997).

[12] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML – Revised.* The MIT Press, 1997.

[13] Odersky, M., Wadler, P., and Wehr, M. A second look at overloading. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture* (San Diego, California, June 1995), ACM Press.

[14] Peyton Jones, S., and Hughes, J. Haskell 98: A non-strict, purely functional language. `http://haskell.cs.yale.edu/onlinereport/`, Jan. 1999.

[15] Remy, D. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 1989).

[16] Wadler, P., and Blott, S. How to make *ad-hoc* polymorphism less *ad hoc.* In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages* (1989).