Programming with Implicit Values, Functions, and Control

or, Implicit Functions: Dynamic Binding with Lexical Scoping

Microsoft Technical Report, MSR-TR-2019-7, v2.

JONATHAN BRACHTHÄUSER, Microsoft Research, USA
DAAN LEIJEN, Microsoft Research, USA

We introduce two new language features, called *implicit functions* and *implicit control*. Both generalize *implicit values* which are a typed implementation of dynamic binding. Implicit functions are *bound dynamically but evaluated in the lexical scope of their binding*. We show how this small generalization from regular implicit values leads to better abstraction. In particular, implicit functions encapsulate (side) effects at the definition site, as opposed to leaking them to the call site. Implicit control further generalizes implicit functions by adding the ability to return into the lexical scope of the binding or to resume to the call-site. We formalize the new features as an extension to Moreau's calculus of dynamic binding (1998). Unifying all three language features in one framework guarantees that the interaction between implicit values, functions, and control is well-defined. We also show how our semantics correspond to a macro-translation into algebraic effect handlers.

Additional Key Words and Phrases: Dynamic Scope, Implicit parameters, Algebraic Effects, Handlers

1. INTRODUCTION

In this article we introduce two new language features, called *implicit functions* and *implicit control*. These two new features are generalizations of *implicit values* (or parameters) which are essentially a typed implementation of dynamic variables. Following Moreau (1998) and Kiselyov, Shan, and Sabry (2006), a *dynamic variable* is a variable whose association with its value exist within the evaluation of an expression and it is not limited to its lexical scope. If several such associations exist, the innermost definition is used, which is called *dynamic binding*. Due to the overloaded meaning of both dynamic and variable, we prefer to use the term *implicit value* instead to refer to a typed discipline of dynamic variables – similar to the *implicit parameters* described Lewis, Launchbury, Meijer, and Shields (2000).

Implicit values associate values within the current execution context (or "stack") and can thus be used to pass extra data to functions and its callees without bloating the function's interfaces and manually threading around extra arguments – instead those arguments are bound implicitly. This is useful in many practical situations, ranging from passing a type environment in a compiler, maintaining context information like input-positions in parsers, to associating the current request object in an asynchronous web server.

Dynamic binding has a somewhat bad reputation since in an unrestricted setting (as in the original Lisp (McCarthy, 1960)) one might bind dynamic variables accidentally. We therefore follow the discipline of Lewis et al. (2000) where implicit values are dynamically bound but explicitly typed. In this article we use simple row types to track the used implicit values and ensure they are always bound. After giving an overview of implicit values in Section 2, we generalize implicit values in two steps, offering new ways of abstraction:

- Our main contribution are *implicit functions*: these are *bound dynamically but evaluated in the lexical scope of their binding*. We show how this small change from regular implicit values leads to better abstraction. In particular, we show how implicit functions allow users to provide precise interfaces without leaking the (side) effects of any particular implementation (Section 2.2).

- We then generalize the notion of implicit functions to *implicit control*. Implicit control allows us to implement control operations like exceptions and backtracking (Section 2.3). We show how local mutable state can be expressed in terms of implicit control (Section 2.4).
- We use the calculus of dynamic binding λ_{db} as defined by Moreau (1998) and Kiselyov, Shan, and Sabry (2006) to formalize the semantics of implicit values (which coincides exactly), and then extend it further, as the calculus λ_{db+}, with rules for implicit functions and control. We also define a new type system extended with *implicit rows* to guarantee that any evaluation is never *bp*-stuck (Kiselyov et al., 2006) (Section 3).
- We then show a typed *macro* translation (Felleisen, 1991) of the extended calculus λ_{db+} to a calculus of algebraic effect handlers (Plotkin and Pretnar, 2013; Plotkin and Power, 2003). The translation preserves typing and semantics, where reduction in the direct semantics corresponds to an equivalent reduction in the translated semantics (Section 4).
  Even though we can translate implicit values and functions to algebraic effect handlers, we argue that these concepts merit study by themselves. Following the principle of least power, each of the two new features gradually adds expressiveness to implicit values. Explicitly distinguishing between implicit values, functions, and control makes it easier to reason about programs, easier to learn the concepts individually, and can allow for more efficient implementation strategies. In this view, implicit functions relate to algebraic effect handlers as `while` statements relate to `goto`.

2. PROGRAMMING WITH IMPLICIT VALUES, FUNCTIONS, AND CONTROL

The ideas in this paper have been fully implemented in the Koka language (Leijen, 2019, 2014) and we use it to provide concrete code examples in this paper[1]. Koka is a strict functional language with full (side) effect tracking in the types (including exceptions and divergence). The reason to use Koka is two-fold: it already has a type system based on row-types which we adapted to track implicit bindings, and the run-time system supports algebraic effect handlers which we use to implement implicit control. However, the ideas described in this paper apply to many programming languages and are not tied to Koka in particular.

2.1. Implicit Values

To motivate the use of implicit values (or parameters), we start with an example of the canonical paper on implicit parameters by Lewis, Launchbury, Meijer, and Shields (2000). We assume a pretty printing library that produces pretty strings from documents:

```
fun pretty( d : doc ) : string
```

Unfortunately, it has a hard-coded maximum display width deep within the code:

```
... if (line.length ≤ 40) then ...
```

To abstract over the maximum display width, there are two choices. We can either make the width into a global mutable variable, or add an extra explicit width parameter to nearly every function in the library and thread it around manually. Neither choice is satisfactory. What is especially bad is that a conceptually small change requires a substantial change to the library.

However, with an *implicit value* we can solve this cleanly. We can globally *declare* the type of an implicit value as:

```
implicit val width : int
```

[1]Our implementation is available at `https://github/koka-lang/koka` in the `dev` branch. Most of the examples in this paper can be loaded in the Koka interpreter as `:l implicits`.

and can refer to it deep inside the library:

```
... if (line.length ≤ width) then ...
```

The type system tracks the use of such implicit values in row types denoted between angled brackets. In particular, the new inferred type of the pretty printing function is:

```
fun pretty( d : doc ) : ⟨width⟩ string
```

signifying that `pretty` can only be used in a context that binds the implicit `width` value. As illustrated by this example, being able to infer this type is important for maintainability. Implicit values are bound with "`with val` p `=` e_1 `in` e_2" expressions:

```
fun pretty-thin( d : doc ) : ⟨ ⟩ string {
  with val width = 40 in pretty(d)
}
```

Here the implicit value `width` is dynamically bound to `40` for the dynamic extent of evaluating the body expression `pretty(d)`. The scope of the `with`-binder is thus not limited to its lexical closure. As we see in the next section, we use the exact same semantics for implicit values as described by Moreau (1998) and Kiselyov, Shan, and Sabry (2006) for dynamic binding. Note how the inferred type of `pretty-thin` now reflects that there are no more dependencies on further implicit values.

There is also a statement form of the `with` expression that scopes over the rest of the current lexical block scope. Using this form we can write the previous example as:

```
fun pretty-thin(d) {
  with val width = 40
  pretty(d)
}
```

Of course, we can also re-bind implicit values. For example we might want to pretty print part of a document with twice the display width:

```
fun pretty-wide( d : doc ) : ⟨width⟩ string {
  with val width = width * 2
  pretty(d)
}
```

Here, the type of `pretty-wide` reflects that even though it binds `width`, it also still depends on a `width` binding in its own context.

We formalize and fully explain the semantics of implicit values (as dynamic binding) in Section 3, Figure 1, but peeking ahead, the essential reduction rule is:

$$\text{with val } p \;=\; v \text{ in } \mathsf{E}[p] \longrightarrow \text{with val } p \;=\; v \text{ in } \mathsf{E}[v] \quad \text{if } p \notin \mathrm{bp}(\mathsf{E})$$

where an implicit binding p finds its bound value v in the dynamic evaluation context E (i.e. the stack). The innermost binding is used due to the side condition $p \notin \mathrm{bp}(\mathsf{E})$ which ensures that p is not bound in E itself. The unchanged evaluation context is marked in *gray*.

Implicit values resolve to the closest with-binding that dynamically surrounds the implicit value. The following example illustrates this scoping of implicit bindings.

```
fun scope() {
  with val width = 40
  val g = fun() { with val width = 80 in width + 1 }
  val h = with val width = 80 in (fun(){ width + 1 })
  g().println
```

```
  h().println
}
```

Here we bind two static functions `g` and `h`, and call them. The first `g().println` prints `81`, since during the evaluation of `g()` the implicit value `width` is bound to its innermost binding of `80`. However, `h().println` prints `41` instead: during evaluation of `h()`, the `width` is bound dynamically to `40` – the binding to `80` was only present during the evaluation of the function value as such. This illustrates that functions do not close over implicit values.

2.2. Implicit Functions

Of course we can bind functions as an implicit value. For example,

```
implicit val emit-naive : ((s : string) → ⟨⟩ ())
```

where the pretty printing library functions can use this to emit partial output:

```
....
match(doc) {
  Text(s) → emit-naive(s)
....
```

Unfortunately, the implicit value declaration has a type signature that severely limits the number of possible implementations. For example, we might want to use the display width in an implementation like:

```
// type error: emit-naive cannot use implicit value 'width'
with val emit-naive = (fun(s : string){ ... s.truncate(width) ... })
...
```

This leads to a type error since the type signature of `emit-naive` promises to not use any implicit values. To work around this restriction, we can of course change the type signature of the value declaration to:

```
implicit val emit-naive : ((s : string) → ⟨width⟩ ())
```

But now the signature exposes accidental details of our particular implementation. Even worse, it might not be the desired semantics, since the width is dynamically bound at the *call site* of `emit-naive`, while we usually want to bind it at the *definition site* and encapsulate this implementation detail[2] in the definition of `emit-naive`.

Not only implicit bindings are resolved at the call-site – the same problem extends to other side-effects. For example, `emit-naive` might print the output directly to the console as:

```
with val emit-naive = (fun(s){ println(s) })
```

If `println` happens to throw an exception, it may be accidentally handled by some unintended exception handler enclosing the call-site of `emit-naive`. Even though we were able to abstract over the implementation of `emit-naive`, the effects and implicit bindings still leak into the calling context. This is especially a problem for languages that are disciplined about side-effects: in Haskell `println` requires the `IO` monad and functions using `emit-naive` would need to be lifted into the `IO` monad. Similarly, in a language like Koka (Leijen, 2014) that tracks effects in the type system, the functions using `emit-naive` would all get an extra `console` effect.

[2] In this particular example, we could also lookup the implicit value once, bind it to an explicit value and close over that value. However, as we will see, this solution does not scale to implicit control.

2.2.1. Dynamic binding with Lexical Scoping. What we need instead are *implicit functions*: such functions *are bound dynamically, but evaluated in the lexical context of their binding*. Similar to implicit values, we can declare an implicit function as:

```
implicit fun emit( s : string ) : ()
```

There is no syntactic difference between calling an implicit functions and calling any other function. However, if we call `emit(...)` in our pretty printer library, the inferred type of `pretty` now becomes:

```
fun pretty( d : doc ) : ⟨width,emit⟩ ()
```

The type now reflects that the function depends on an implicit binding for both the display width and an emitter function. Implicit functions are bound like implicit values, and we can change our implementation of `pretty-thin` to also provide a binding for `emit`:

```
fun pretty-thin(d) {
  implicit val width = 40
  implicit fun emit(s) { println(s.truncate(width)) }
  pretty(d)
}
```

This definition of `emit` is very different from the previous binding as an implicit value `emit-naive`, since `emit`'s body executes in the lexical context of `pretty-thin` and not in its calling context. In particular, the implicit value `width` in the body of `emit` is now always resolved to 40 no matter if `width` is rebound inside `pretty`. Also, any exception thrown by `println` is handled by the innermost exception handler around `pretty-thin`, not by some handler inside `pretty`.

Again peeking ahead to the formalization in Section 3.2, Figure 4, the essential reduction rule for implicit functions is:

$$\text{with fun } p(x) \;=\; e \text{ in } \mathsf{E}[p(v)] \longrightarrow (\lambda y.\ \text{with fun } p(x) \;=\; e \text{ in } \mathsf{E}[y])(e[x{:=}v]) \qquad \text{if } p \notin \mathrm{bp}(\mathsf{E})$$

In contrast to the implicit value rule, we first evaluate the function body e with x substituted by v (denoted by $e[x{:=}v]$) *outside* the calling evaluation context E, and only after that resume in the original context with the result y.

2.2.2. A Novel Abstraction Mechanism. Changing the evaluation context to the defining scope seems a minor extension with respect to functions bound as values, but it has profound implications for abstraction. In particular, we are now able to contain implicit bindings and effects to the lexical context of the implicit function definition. Continuing with our example, we may want to collect all output using local mutable state:

```
fun emit-collect(action) {
  var out := ""
  with fun emit(s) { out := out + s + "\n" } in action()
  out
}
```

The dynamically bound `emit` function can access the locally scoped mutable variable `out` from its body and update it. As we will see in Section 2.4, our local mutable variables are not heap allocated and cannot be accessed outside of their lexical scope. If the `emit` function would be bound as an implicit value, the type checker would not allow a reference to the `out` variable. In contrast, with implicit functions this is allowed as the body executes in the lexical context, and `action` can use `emit` as a `string → ⟨emit⟩ ()` function where any effects are isolated to the scope of `emit-collect`. As an example, the expression

```
emit-collect(fun(){ emit("hello"); emit("world") })
```

just returns the string `"hello\nworld\n"` without any observable side-effects.
Languages like C#, JavaScript, and Scala inhabit a middle-ground: lambda expressions can capture local variables by reference and would thus behave like in our example. However, they still leak other effects, like throwing an exception, to the calling context. Also, in these languages the local variables can outlive the lexical scope and are heap allocated which breaks abstraction when combined with control effects as we show in Section 2.3.1.

2.2.3. Example: Depth-First Traversal. Another nice example of the abstraction power of implicit functions is illustrated by adapting an example of Lewis, Launchbury, Meijer, and Shields (2000). The authors describes a depth-first traversal of a graph (King and Launchbury, 1995), where the auxiliary function `dfs-loop` is implicitly parameterized by three functions: one to mark vertices, one to query if a vertex is marked, and one to get the children of a vertex in the (implicit) graph. In the original example, the authors then use `runST` (Peyton Jones and Launchbury, 1995) to efficiently implement the marking with isolated mutable state. With implicit functions we can implement this as:

```
alias vertex = int
type graph { ... }
type rose  { Rose(v : vertex, sub : list⟨rose⟩ ) }

implicit fun marked( v : vertex ) : bool
implicit fun mark( v : vertex ) : ()
implicit fun children( v : vertex ) : list⟨vertex⟩

fun dfs( g : graph, vs : list⟨vertex⟩ ) : list⟨rose⟩ {
  var visited := vector(g.bound,False)
  with fun children(v) { g.gchildren(v) }
  with fun marked(v)   { visited[v] }
  with fun mark(v)     { visited[v] := True }
  dfs-loop(vs)
}

fun dfs-loop( vs : list⟨vertex⟩ ) {
  match(vs) {
    Nil → Nil
    Cons(v,vv) →
      if (marked(v)) then dfs-loop(vv) else {
        mark(v)
        val sub = dfs-loop( children(v) )
        Cons( Rose(v,sub), dfs-loop(vv) )
      }
  }
}
```

The `dfs` function completely encapsulates the use of (scoped) mutable state to efficiently implement the marking of the visited vertices. The type of `dfs-loop` reflects that it only depends on the declared implicit functions and has no other side effects:

```
fun dfs-loop( vs : list⟨vertex⟩ ) : ⟨mark,marked,children⟩ list⟨rose⟩
```

In contrast, Lewis et al. (2000) bind functions by *value* and thus leak the side-effects of the particular implementation that uses mutable state into the `dfs-loop` function. The loop needs to be written in a monadic style and has the type:

```
dfsLoop :: (?children :: Graph → [Vertex],
            ?marked :: Vertex → ST s Bool,
            ?mark :: Vertex → ST s ()) ⇒ [Vertex] → ST s [Rose]
```

where the `ST` effect of the operations leaks into the definition of `dfsLoop`.

Note also how the type of `dfsLoop` is quite verbose. With implicit parameters the implicit names do not have a declared type which is more flexible but leads to large type signatures. With explicit type declarations for implicit values the types are more concise and allow for better type checking at the use site of an implicit function.

2.3. Implicit Control

Implicit functions are evaluated in the lexical scope of their definition but still return to the calling context just like regular functions. *Implicit control* functions are a further extension to implicit functions that *return to lexical scope of their definition instead* – similar to how exceptions "return" to their innermost `try` block. For example, let's extend our pretty printing example to stop once a certain amount of output has been produced[3]:

```
implicit control stop() : ()

fun pretty-stop(d) {
  with control stop() { "" }
  with emit-collect
  pretty-thin(d)
}
```

Inside the pretty printing rendering function, we can now add a check to stop pretty printing early on (assuming an implicit `produced` function):

```
... if (produced() ≥ 100) then stop() ...
```

If `stop` is called, it returns directly to its definition point with the empty string as the result of `pretty-stop`. If we like to return with the output produced up until the point of stopping, we can switch the binding site with `emit-collect`:

```
fun pretty-stop(d) {
  with emit-collect
  with control stop() { () }
  pretty-thin(d)
}
```

Implicit control also subsumes exception handling, where we can define an implicit function `throw` with a `try` handler implemented as an implicit control binding. For example, here is a function that transforms an exception throwing action to a `maybe` result:

[3]Here we use an extension of `with` statement syntax in Koka where we can pass a function expression that receives the rest of the block as a function argument, and the example desugars to:

```
fun pretty-stop(d) {
  with control stop() { "" }
  emit-collect(fun(){pretty-thin(d)})
}
```

```
implicit control throw(msg : string) : ()

fun to-maybe( action : () → ⟨throw⟩ a ) : maybe⟨a⟩ {
  with control throw(msg) { Nothing }
  Just(action())
}
```

Peeking ahead to the formalization in Section 3.2, Figure 4, we see that the reduction rule for implicit control can be simplified to:

$$\text{with control } p(x)\ r\ =\ e \text{ in } \mathsf{E}[p(v)] \longrightarrow (\lambda x.\ e)(v) \quad \text{if } p \notin \text{bp}(\mathsf{E})$$

where we ignore the binding for r now. The rule is basically equivalent to the rule for implicit functions except that we do not continue evaluation in the original context.

2.3.1. Resuming Control. In the previous pretty printing example, we defined a new implicit control `throw` in order to stop early. Can we also rephrase the example to implement all of the combined functionality in the definition of `emit` itself? To be able to do that, we need to be explicit about how to return: either resuming to the calling context, or returning to the definition context. To enable this, a control binding gets passed an extra argument `resume` that can be used to return to the calling context instead. We can now extend `emit` to do the check:

```
implicit control emit( s : string ): ()
fun emit-collect(action) {
  var out := ""
  with control emit(s) {
    out := out + s + "\n"
    if (out.length ≥ 100) then () else resume(())
  } in action()
  out
}
```

The `resume` argument is a first-class function and captures the delimited continuation. The formalization in Section 3.2, Figure 4 shows the full reduction rule for implicit control binding the resume function to r:

$$\begin{aligned}&\text{with control } p(x)\ r\ =\ e \text{ in } \mathsf{E}[p(v)]\\ &\quad\longrightarrow (\lambda x.\ \lambda r.\ e)(v)(\lambda y.\ \text{with control } p(x)\ r\ =\ e \text{ in } \mathsf{E}[y]) \quad \text{if } p \notin \text{bp}(\mathsf{E})\end{aligned}$$

giving us now a choice to resume in the original calling context.

As another example, we can use the resumption to implement backtracking by calling it multiple times:

```
implicit control choice() : bool

fun amb( action : () → ⟨choice|e⟩ string ) : e list⟨string⟩ {
  with control choice() { resume(True) + resume(False) }
  [action()]
}
```

Now, we can use `choice` in the pretty printing library to produce all possible layouts. For example, if

```
fun f() : list⟨string⟩ {
  with amb
  with emit-collect
  emit("hi")
  if (choice()) then emit("world") else emit("universe")
}
```

then `f()` returns:

```
["hi\nworld\n","hi\nuniverse\n"]
```

Note that the order of binding is important here: because `amb` is on the outside, each resumption resumes with all its captured variables reset to their value *at capture time*, e.g. in our example both outputs start with the captured `"hi\n"` output, and the second resumption does *not* include any output appended from the first resumption. In other words, local variables are stack-allocated and not heap-allocated.

2.4. Mutable Variables as Implicit Control

The previous example showed that the interaction between local mutable state and implicit control is subtle due to the first-class (delimited) continuation captured by `resume`. This is already remarked upon by Moreau (1998) who calls for "a single framework integrating continuations, side-effects, and dynamic binding." and studied by Kiselyov et al. (2006) in the context of delimited continuations.

However, it turns out we can view local mutable state in terms of implicit control itself and thus we do not need a special semantic treatment. In particular, we can use the same translation as by Kammar and Pretnar (2017) (Figure 7) where they show how to express mutable dynamic variables in term of algebraic effect handlers. We reuse their translation, except that we use implicit control instead of general effect handlers.

First we α-rename such that local variables are uniquely named. Every binding of a local variable `s` of some type τ is then translated to a function application of a $\texttt{local}_\texttt{s}$ function:

```
var s : τ := init   ⇝  local_s(init,fun(){ ... })
...
```

and lexically bound occurrences of `s` are translated to either $\texttt{get}_\texttt{s}$ or $\texttt{set}_\texttt{s}$ operations

```
s := expr   ⇝  set_s(expr)
s           ⇝  get_s()
```

where we define:

```
implicit control get_s()       : τ
implicit control set_s(x : τ) : ()

fun local_s( init : τ, action : () → ⟨get_s,set_s|e⟩ a ) : e a {
  val f = { with control get_s()  { (fun(st) { resume(st)(st) }) }
            with control set_s(x) { (fun(st) { resume(())(x) }) }
            val x = action()
            (fun(st){ x })
          }
  f(init)
}
```

The main difference with the translation in Kammar and Pretnar (2017) is that we use two separate implicit control functions while they group them under a single handler. Otherwise, both translations express the mutable state as a state monad returning a function that gets the current state as input.

This as also essentially the way Kiselyov et al. (2006) express dynamic binding in terms of delimited control where the occurrence of a binding `s` is shifted as (Kiselyov et al., 2006,Figure 3):

shift s as f in $\lambda y.\ f\ y\ y$

where f is our `resume` and y our `st` parameter, corresponding to the definition of $\texttt{get}_\texttt{s}$.

The translation is a *macro* translation (Felleisen, 1991) in the sense that it is defined homomorphically over the syntax of the language without collecting global information. It keeps the core of the language unchanged – only translating local variables.

Of course, this translation using a state monad is useful from a semantic perspective, but in a practical implementation we can use more efficient mechanisms where we just need to ensure the state is properly captured and restored on a resume. In our implementation in Koka we use direct mutation of variables that are (handler) stack allocated.

2.4.1. Safety. Viewing scoped, local mutable variables in terms of implicit control has the added advantage that it can be used by the type checker to ensure that such variables do not escape from their lexical scope. Consider for example:

```
fun escape() {
  var s := 0
  (fun() { s := s + 1; s })
}
```

where the result of `escape()` is a function that captured the (supposedly) local mutable state `s`. If we apply the translation to implicit control though

```
fun escape() {
  local_s(0,fun(){
    (fun() { set_s(get_s() + 1; get_s() })
  })
}
```

it becomes clear that the type of the result function becomes `() → ⟨get_s,set_s⟩ int`, i.e. it will be impossible to use this function as it depends on two (unique and hidden) implicits that cannot be bound by the user. The type checker can easily check for such types at top-level and issue an error at the definition site.

3. FORMALIZATION

To formalize the basic calculus of implicit values we are using the calculus of dynamic binding, λ_{db}, by Moreau (1998) as shown in Figure 1, and the corresponding type system in Figure 2 as defined by Kiselyov, Shan, and Sabry (2006). Except for formatting, the calculus and type rules are exactly the same. The main cosmetic differences are:

- Dynamic binding, dlet $p\ =\ V$ in E, is formatted as: with val $p\ =\ v$ in e.
- Signatures, $\Sigma, p : \tau$, are formatted as: $\Sigma, p : \text{val}\ \tau$.
- Bound implicits ("parameters"), $BP(\mathsf{E})$, are formatted as: bp(E).

There are two disjoint sets of variables: lexical variables denoted with x and y, and dynamic variables (implicit names) denoted by p and q. Note that values v can contain references to dynamic bindings, like $\lambda x.p$ but that dynamic binding names are not values by themselves. The evaluation contexts E contains the clause with val $p\ =\ v$ in E, which shows that a context can capture a dynamic binding. The set of bound variables in a context E is denoted as bp(E) (and defined in Figure 1).

The operational semantics are given by the three rules for $\longrightarrow$ where the $\longmapsto$ relation lets us evaluate according to the evaluation context. The rule (*dval*) captures the semantics of implicit values (dynamic binding) where the condition $p \notin \mathsf{bp}(\mathsf{E})$ ensures that we always bind to the innermost binding.
The evaluation can get stuck if a dynamic variable is not bound, and Kiselyov et al. (2006) define such terms as *bp*-stuck:

Definition 1. (*bp-stuck*)
A term is *bp*-stuck if it has the form $\mathsf{E}[p]$ where $p \notin \mathsf{bp}(\mathsf{E})$.

The type system for λ_{db} is given in Figure 2. For simplicity it is given as a monomorphic type system but there are no difficulties extending this to a polymorphic setting and adding type inference. Unbound variables notwithstanding, the type system is sound and Kiselyov et al. (2006) prove progress and preservation:

Theorem 1. (*Preservation*)
If $\Gamma \vdash_{db} e : \tau$ and $e \longmapsto e'$, then $\Gamma \vdash_{db} e' : \tau$.

Theorem 2. (*Progress*)
If $\emptyset \vdash_{db} e : \tau$ and e is not a value and not *bp*-stuck, then $e \longmapsto e'$ for some term e'.

3.1. Static Types for Implicit Values

Figure 3 defines more precise type rules for λ_{db} that track the use of dynamic bindings by annotating every function arrow with a *implicit row* π. A row is either empty $\langle\rangle$ or an extension with a implicit name $\langle p|\pi\rangle$. We sometimes use the following shorthands:

$$\begin{aligned}\langle p_1, \ldots, p_n|\pi\rangle &= \langle p_1| \ldots \langle p_n|\pi\rangle \ldots \rangle \\ \langle p_1, \ldots, p_n\rangle &= \langle p_1| \ldots \langle p_n|\langle\rangle\rangle \ldots \rangle\end{aligned}$$

Following Leijen (2005), these rows can contain multiple occurrences of a name and are considered equal up to the order of the implicit names in the row. Leijen (2005) shows how the rows can be naturally extended with polymorphism and allow full unification, making them well suited to combine with Hindley-Milner style type inference. Allowing duplicates is important for typing implicit bindings that refer themselves to the same implicit name. For example, consider typing $(\lambda x.\ \mathsf{with\ val}\ p\ =\ x\ \mathsf{in}\ e)(p)$:

$$\frac{\dfrac{\ldots}{\Gamma \vdash_{imp} (\lambda x.\mathsf{with\ val}\ p\ =\ x\ \mathsf{in}\ e) : \tau_1 \to \langle p|\pi\rangle\ \tau \mid \langle p|\pi\rangle} \quad \dfrac{\Sigma(p) = \mathsf{val}\ \tau_1}{\Gamma \vdash_{imp} p : \tau_1 \mid \langle p|\pi\rangle}}{\Gamma \vdash_{imp} (\lambda x.\mathsf{with\ val}\ p\ =\ x\ \mathsf{in}\ e)(p) : \tau \mid \langle p|\pi\rangle}\ [\textsc{app}]$$

which means that the first premise is typed as:

$$\frac{\Sigma(p) = \mathsf{val}\ \tau_1 \quad \Gamma, x : \tau_1 \vdash_{imp} x : \tau_1 \mid \langle p|\pi\rangle \quad \Gamma \vdash_{imp} e : \tau \mid \langle p|\langle p|\pi\rangle\rangle}{\Gamma, x : \tau_1 \vdash_{imp} \mathsf{with\ val}\ p\ =\ x\ \mathsf{in}\ e : \tau \mid \langle p|\pi\rangle}\ [\textsc{wval}]$$

which leads to typing e with two occurrences of p in the implicit row. Having such duplicates keeps the system simple and avoids the need for special row constraints (Gaster and Jones, 1996; Rémy, 1994; Hillerström and Lindley, 2016).

The use of the implicit rows in the type rules now ensures that well-typed terms can never be *bp*-stuck:

Syntax:

Expressions	e	::=	v	value
		|	$e(e)$	application
		|	with b in e	dynamic binding
		|	p	implicit name
Values	v	::=	x	variables
		|	$\lambda x.\ e$	lambda expressions
Dynamic Binding	b	::=	val $p\ =\ v$	value binding
Evaluation Context	E	::=	$\square \mid \mathsf{E}(e) \mid v(\mathsf{E}) \mid$ with b in E	

Bound Implicits:

$$
\begin{aligned}
\mathsf{bp}(\square) &= \emptyset \\
\mathsf{bp}(\mathsf{E}(e)) &= \mathsf{bp}(\mathsf{E}) \\
\mathsf{bp}(v(\mathsf{E})) &= \mathsf{bp}(\mathsf{E}) \\
\mathsf{bp}(\text{with } b\ =\ v \text{ in } \mathsf{E}) &= \mathsf{bp}(b) \cup \mathsf{bp}(\mathsf{E}) \\
\\
\mathsf{bp}(\text{val } p\ =\ v) &= \{p\}
\end{aligned}
$$

Operational Semantics:

$$
\begin{array}{lllll}
(\beta) & (\lambda x.\ e)(v) & \longrightarrow & e[x := v] & \\
(dret) & \text{with } b \text{ in } v & \longrightarrow & v & \\
(dval) & \text{with val } p\ =\ v \text{ in } \mathsf{E}[p] & \longrightarrow & \text{with val } p\ =\ v \text{ in } \mathsf{E}[v] & \text{if } p\ \notin \mathsf{bp}(\mathsf{E})
\end{array}
$$

$$
\frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\ [\textsc{eval}]
$$

Fig. 1. Language of dynamic binding, λ_{db}

Theorem 3.
If $\Gamma \vdash_{imp} e : \tau \mid \langle\rangle$ then e is not bp-stuck.

To prove this, we first need the following Lemma:

Lemma 1. (*Implicits are meaningful*)
If $\Gamma \vdash_{imp} \mathsf{E}[p] : \tau \mid \pi$ and $p \notin \mathsf{bp}(\mathsf{E})$, then $p \in \pi$.

This is an important lemma as it states that implicit types cannot be discarded except through binding. It also means that if an expression has an empty implicit row, it will not use any dynamic binding.

Proof. We proceed by induction over the structure of the evaluation contexts, where we assume $p \notin \mathsf{bp}(\mathsf{E})$ and that the induction hypothesis holds for some $\mathsf{E}'[p]$.
case $\mathsf{E}[p] = p$: We can type $\Gamma \vdash_{imp} p : \tau \mid \langle p|\pi\rangle$ and thus $p \in \langle p|\pi\rangle$.

Syntax of Types:

Types	τ	::=	c	constants (constructors)
		|	$\tau \to \tau$	functions
Constants	c	::=	$\ldots$	
Type Environment	Γ	::=	$\emptyset \mid \Gamma, x : \tau$	
Implicit Signature	Σ	::=	$\emptyset \mid \Sigma, p : \mathsf{val}\ \tau$	

Type Rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{db} x : \tau}\ [\textsc{var}] \qquad \frac{\Gamma, x : \tau_1 \vdash_{db} e : \tau_2}{\Gamma \vdash_{db} \lambda x.e : \tau_1 \to \tau_2}\ [\textsc{lam}]$$

$$\frac{\Sigma(p) = \mathsf{val}\ \tau}{\Gamma \vdash_{db} p : \tau}\ [\textsc{dval}] \qquad \frac{\Gamma \vdash_{db} e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_{db} e_2 : \tau_2}{\Gamma \vdash_{db} e_1(e_2) : \tau_2}\ [\textsc{app}]$$

$$\frac{\Sigma(p) = \mathsf{val}\ \tau_1 \quad \Gamma \vdash_{db} v : \tau_1 \quad \Gamma \vdash_{db} e : \tau_2}{\Gamma \vdash_{db} \mathsf{with\ val}\ p = v\ \mathsf{in}\ e : \tau_2}\ [\textsc{wval}]$$

Fig. 2. Standard type rules for λ_{db}

Syntax of Types:

Types	τ	::=	c	type constructors
		|	$\tau \to \pi\ \tau$	functions
Implicit Row	π	::=	$\langle\rangle$	empty row
		|	$\langle p \mid \pi \rangle$	row extension
Constants	c	::=	$p\ \ldots$	
Type Environment	Γ	::=	$\emptyset \mid \Gamma, x : \tau$	
Implicit Signature	Σ	::=	$\emptyset \mid \Sigma, p : \mathsf{val}\ \tau$	

Type Rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{val} x : \tau}\ [\textsc{var}] \qquad \frac{\Gamma, x : \tau_1 \vdash_{imp} e : \tau_2 \mid \pi}{\Gamma \vdash_{val} \lambda x.e : \tau_1 \to \pi\ \tau_2}\ [\textsc{lam}]$$

$$\frac{\Gamma \vdash_{val} v : \tau}{\Gamma \vdash_{imp} v : \tau \mid \pi}\ [\textsc{val}] \qquad \frac{\Gamma \vdash_{imp} e_1 : \tau_1 \to \pi\ \tau_2 \mid \pi \quad \Gamma \vdash_{imp} e_2 : \tau_2 \mid \pi}{\Gamma \vdash_{imp} e_1(e_2) : \tau_2 \mid \pi}\ [\textsc{app}]$$

$$\frac{\Sigma(p) = \mathsf{val}\ \tau}{\Gamma \vdash_{imp} p : \tau \mid \langle p \mid \pi \rangle}\ [\textsc{dval}] \qquad \frac{\Sigma(p) = \mathsf{val}\ \tau_1 \quad \Gamma \vdash_{val} v : \tau_1 \quad \Gamma \vdash_{imp} e : \tau_2 \mid \langle p \mid \pi \rangle}{\Gamma \vdash_{imp} \mathsf{with\ val}\ p = v\ \mathsf{in}\ e : \tau_2 \mid \pi}\ [\textsc{wval}]$$

Fig. 3. Improved type rules for λ_{db}. We add *implicit rows* to ensure all implicit values are bound in the end. Implicit rows are considered equivalent up to the order of the names in the row.

case $\mathsf{E}[p] = \mathsf{E}'[p](v)$: Since $\Gamma \vdash_{imp} \mathsf{E}[p] : \tau \mid \pi$, we can use APP and thus $\Gamma \vdash_{imp} \mathsf{E}'[p] : \tau_1 \to \pi\ \tau \mid \pi$ **(1)** (and $\Gamma \vdash_{imp} v : \tau_1 \mid \pi$). By definition $p \notin \mathrm{bp}(\mathsf{E})$ implies $p \notin \mathrm{bp}(\mathsf{E}')$, and with (1) and the induction hypothesis we have $p \notin \pi$.
case $\mathsf{E}[p] = v(\mathsf{E}'[p])$: Similar to the previous case.
case $\mathsf{E}[p] = \text{with val } q = v \text{ in } \mathsf{E}'[p]$: Since $p \notin \mathrm{bp}(\mathsf{E})$, we have that $p \notin (\{q\} \cup \mathrm{bp}(\mathsf{E}'))$ and thus $p \neq q$ **(2)** and $p \notin \mathrm{bp}(\mathsf{E}')$ **(3)**. Applying rule WVAL, we have $\Gamma \vdash_{imp} \text{with val } q = v \textit{ in } \mathsf{E}'[p] : \tau \mid \pi$ and thus $\Gamma \vdash_{imp} \mathsf{E}'[p] : \tau \mid \langle q|\pi\rangle$ **(4)**. We can apply the induction hypothesis to (4) with (3) to derive $p \notin \pi$, and with (2), $p \notin \langle q|\pi\rangle$. □

Proof. (*Of Theorem 3*) With Lemma 1 we can now prove Theorem 3 by contradiction: suppose that there is some e such that $\Gamma \vdash_{imp} e : \tau \mid \langle\rangle$ where e is bp-stuck. In that case, by the definition of bp-stuck, e must be of the form $\mathsf{E}[p]$ where $p \notin \mathrm{bp}(\mathsf{E})$. But in that case we can apply Lemma 1 and derive that $p \in \langle\rangle$, dismissing our assumption. □

Our type system is also a conservative extension of the original type system. If we define a simple erasure function $\hat{\cdot} : \tau_{imp} \to \tau_{db}$ as:

$$\hat{c} = c$$
$$\widehat{(\tau_1 \to \pi\ \tau_2)} = \hat{\tau_1} \to \hat{\tau_2}$$

and extend that naturally over type environments, we can then state the following lemma:

Lemma 2. (*Conservative Extension*)
If $\Gamma \vdash_{imp} e : \tau \mid \pi$ then $\hat{\Gamma} \vdash_{db} e : \hat{\tau}$.

This is immediate by the erasure of the implicit rows from the derivation and removing identity (VAL) derivations. We can now show progress as a corollary:

Theorem 4. (*Progress*)
If $\emptyset \vdash_{imp} e : \tau \mid \langle\rangle$ and e is not a value, then $e \longmapsto e'$ for some term e'.

Proof. From Lemma 2 we know $\emptyset \vdash_{db} e : \hat{\tau}$ and from Theorem 3 that e is not db-stuck. We can now apply Theorem 2 to conclude $e \longmapsto e'$. □

Theorem 5. (*Preservation*)
If $\Gamma \vdash_{imp} e : \tau \mid \pi$ and $e \longmapsto e'$ for some term e', then $\Gamma \vdash_{imp} e' : \tau \mid \pi$.

To show preservation, we need to redo the various lemmas from Kiselyov et al. (2006) but the proofs carry over almost unchanged.

Lemma 3. (*Value Substitution*)
If $\Gamma \vdash_{val} v : \tau'$, and $\Gamma, x : \tau' \vdash_{imp} e : \tau \mid \pi$ then $\Gamma \vdash_{imp} e[x{:=}v] : \tau \mid \pi$.

Lemma 4. (*Context Substitution*)
If $\Gamma \vdash_{imp} \mathsf{E}[e] : \tau \mid \pi$, then there exist a τ' such that $\Gamma \vdash_{imp} e : \tau' \mid \pi'$ and forall e' with $\Gamma \vdash_{imp} e' : \tau' \mid \pi'$ we also have $\Gamma \vdash_{imp} \mathsf{E}[e'] : \tau \mid \pi$.

Proof. (*Of Theorem 5*) By case analysis over the evaluation rules:
case $(\lambda x.e)(v) \longrightarrow e[x{:=}v]$: Assuming $\Gamma \vdash_{imp} (\lambda x.e)(v) : \tau \mid \pi$, we have $\Gamma \vdash_{imp} \lambda x.e : \tau_1 \to \pi\ \tau \mid \pi$ **(1)** and $\Gamma \vdash_{imp} v : \tau_1 \mid \pi$ **(2)**. (1) must be derived through rule LAM and thus $\Gamma, x : \tau_1 \vdash_{imp} e : \tau \mid \pi$. We can now use lemma 3 with (2) to conclude $\Gamma \vdash_{imp} e[x{:=}v] : \tau \mid \pi$.

Extended Syntax:

Expressions	e	::=	…	
		|	$p(v)$	dynamic application
Binding	b	::=	…	
		|	fun $p(x) = e$	function binding
		|	control $p(x)\ r = e$	control binding

Extended Bound Implicits:

$\text{bp}(\text{fun } p(x) = e) = \{p\}$
$\text{bp}(\text{control } p(x)\ r = e) = \{p\}$

Extended Semantics:

$$(dfun)\quad \text{with fun } p(x) = e \text{ in } \mathsf{E}[p(v)] \longrightarrow (\lambda y.\ \text{with fun } p(x) = e \text{ in } \mathsf{E}[y])(e[x{:=}v]) \quad \text{if } p \notin \text{bp}(\mathsf{E})$$

$$(dctl)\quad \text{with control } p(x)\ r = e \text{ in } \mathsf{E}[p(v)] \longrightarrow e[x{:=}v, r{:=}\lambda y.\ \text{with control } p(x)\ r = e \text{ in } \mathsf{E}[y]] \quad \text{if } p \notin \text{bp}(\mathsf{E})$$

with fresh y

Extended Type Rules:

Implicit Signature $\Sigma ::= \emptyset \mid \Sigma, p : \text{val } \tau \mid \Sigma, p : \text{fun } \tau_1 \to \tau_2$

$$\frac{\Sigma(p) = \text{fun } \tau_1 \to \tau_2 \quad \Gamma \vdash_{val} v : \tau_1}{\Gamma \vdash_{imp} p(v) : \tau_2 \mid \langle p|\pi\rangle}\ [\textsc{dfun}]$$

$$\frac{\Sigma(p) = \text{fun } \tau_1 \to \tau_2 \quad \Gamma, x : \tau_1 \vdash_{imp} e_1 : \tau_2 \mid \pi \quad \Gamma \vdash_{imp} e_2 : \tau \mid \langle p|\pi\rangle}{\Gamma \vdash_{imp} \text{with fun } p(x) = e_1 \text{ in } e_2 : \tau \mid \pi}\ [\textsc{wfun}]$$

$$\frac{\Sigma(p) = \text{fun } \tau_1 \to \tau_2 \quad \Gamma, x : \tau_1, r : \tau_2 \to \pi\ \tau \vdash_{imp} e_1 : \tau \mid \pi \quad \Gamma \vdash_{imp} e_2 : \tau \mid \langle p|\pi\rangle}{\Gamma \vdash_{imp} \text{with control } p(x)\ r = e_1 \text{ in } e_2 : \tau \mid \pi}\ [\textsc{wctl}]$$

Fig. 4. The language λ_{db+} extends λ_{db} with dynamic functions and dynamic control

case with val $p = v'$ in $v \longrightarrow v$: Assuming $\Gamma \vdash_{imp}$ with val $p = v$ in $v : \tau \mid \pi$ **(1)** we conclude from rule WVAL that $\Gamma \vdash_{imp} v : \tau \mid \langle p|\pi\rangle$. Since it is a value, it must be derived through rule VAL and thus $\Gamma \vdash_{val} v : \tau$. We can now use VAL again to derive $\Gamma \vdash_{imp} v : \tau \mid \pi$.
case with val $p = v$ in $\mathsf{E}[p] \longrightarrow$ with val $p = v$ in $\mathsf{E}[v]$ with $p \notin \text{bp}(\mathsf{E})$: Assuming the premise $\Gamma \vdash_{imp}$ with val $p = v$ in $\mathsf{E}[p] : \tau \mid \pi$ **(1)** we conclude from rule WVAL that $\Gamma \vdash_{val} v : \tau_1$ **(2)** and from rule DVAL that $\Gamma \vdash_{imp} p : \tau_1 \mid \langle p|\pi'\rangle$ **(3)**. Using the type rule VAL we can derive with (2) that $\Gamma \vdash_{imp} v : \tau_1 \mid \langle p|\pi'\rangle$. We can now use the context substitution lemma 4 with (1) and (3) to finally derive $\Gamma \vdash_{imp}$ with val $p = v$ in $\mathsf{E}[v] : \tau \mid \pi$. □

3.2. Extending λ_{db} with Implicit Functions and Control

Figure 4 extends the base language λ_{db} with implicit functions and implicit control. We call the extension λ_{db+}, the calculus of implicit values, functions, and control. For simplicity we restrict the implicit functions and control to a single argument. Also, instead of using an implicit `resume` binding, we pass the resumption function explicitly as r in the control bindings.

The evaluation rule (*dfun*) for implicit functions clearly shows how we evaluate the function, $(\lambda x.\ e)(v)$, outside the calling context E and in the context of the binding itself. After evaluation, we resume the original context with the result y as with $\mathsf{fun}\ p(x)\ =\ e\ \mathit{in}\ \mathsf{E}[y]$. The rule for control (*dctl*) is similar except that the resumption function is passed explicitly and bound to r.

The implicit signatures Σ are extended with one new form $\mathsf{fun}\ \tau_1 \to \tau_2$. For simplicity we use this for both function and control declarations. In a concrete language design though, one might want to reject control bindings for implicits declared as an implicit function.

We also extend the type rules with implicit rows with two new rules for checking implicit functions and control. The rules are similar to the WVAL rule in Figure 3 where the implicit name p is discharged from the implicit row. The type of the resumption function r in the type rule for control is interesting: it gets an argument of type τ_2, the result type of the implicit function p; and returns a value of τ, the result type of the body e_2. Since the resumption is evaluated itself under a with binding (see (*dctl*)), the implicit row is just π without p. Extending the proofs of progress and preservation (Theorem 4 and 5) present no further difficulties and is similar to the case for WVAL.

The extension of the monomorphic type system to polymorphism is also possible without difficulties. In particular, we have a full implementation of implicits in Koka with (higher-rank) polymorphic type inference, including extensible implicit row types.

4. TRANSLATING TO ALGEBRAIC EFFECT HANDLERS

To gain confidence in the given semantics, we define a translation of implicits to a calculus of algebraic effects and handlers (Plotkin and Power, 2003; Plotkin and Pretnar, 2013). The strong theoretical foundation and expressiveness of algebraic effects make this an excellent target – and not without precedent, as Kammar and Pretnar (2017) already show how to translate mutable dynamic binding to algebraic effects.

We use the effect calculus defined by Leijen (2017b) since that expression language corresponds most closely to λ_{db} calculus of Moreau (1998). Figure 5 defines the syntax and semantics of λ_{aeh}. We make a small change to the original calculus by Leijen (2017b) where instead of using special H_{op} contexts, we use regular E contexts together with a side-condition $op \notin \mathrm{bop}(\mathsf{E})$ where bop are the bound operations in the context E. We can show that these constraints are equivalent.

Lemma 5.
If $op \notin \mathrm{bop}(\mathsf{E})$ then $\mathsf{E}\ =\ \mathsf{H}_{op}$, where we use the original definition of H_{op} (Leijen, 2017b):

$$\begin{aligned}\mathsf{H}_{op} ::= {} & \square \mid \mathsf{H}_{op}(e) \mid v(\mathsf{H}_{op}) \\ \mid {} & \mathsf{handle}_h(\mathsf{H}_{op}) \quad \text{if } op \notin h\end{aligned}$$

Proof. We can show this by induction over the structure of the evaluation context. The interesting case is for the handle expression. Assuming inductively that the lemma holds for E, we have $\mathsf{E}'\ =\ \mathsf{handle}_h(\mathsf{E})$ with $op \notin \mathrm{bop}(\mathsf{E}')$. By the definition of bop, we have $op\ \notin \{op_1, \ldots, op_n\} \cup \mathrm{bop}(\mathsf{E})$, and thus $op \notin h$ **(1)** and $op \notin \mathrm{bop}(\mathsf{E})$ **(2)**. From (2) and the induction hypothesis we have that $\mathsf{E}\ =\ \mathsf{H}_{op}$, and together with (1) we can derive that $\mathsf{E}'\ =\ \mathsf{H}'_{op}\ =\ \mathsf{handle}_h(\mathsf{H}_{op})$.

Syntax:

Expressions	e	::=	v	values
		|	$e(e)$	application
		|	$\mathsf{handle}_h(e)$	handler
		|	$op(v)$	operation invocation
Values	v	::=	x	variables
		|	$\lambda x.\ e$	lambda expressions
Clauses	h	::=	$\mathsf{return}\ x \to e$	return clause
		|	$op(x)\ r \to e;\ h$	operation clause
Evaluation Context	E	::=	$\square \mid \mathsf{E}(e) \mid v(\mathsf{E}) \mid \mathsf{handle}_h(\mathsf{E})$	

Bound Operations:

$$
\begin{aligned}
&\mathsf{bop}(\square) &&= \emptyset \\
&\mathsf{bop}(\mathsf{E}(e)) &&= \mathsf{bop}(\mathsf{E}) \\
&\mathsf{bop}(v(\mathsf{E})) &&= \mathsf{bop}(\mathsf{E}) \\
&\mathsf{bop}(\mathsf{handle}\{op_1(x_1)\ r_1 \to e_1;\ \ldots;\ op_n(x_n)\ r_n \to e_n;\ \mathsf{return}(x) \to e_r\ \}(\mathsf{E})) \\
&\qquad\qquad &&= \{op_1, \ldots, op_n\} \cup \mathsf{bop}(\mathsf{E})
\end{aligned}
$$

Operational Semantics:

$$
\begin{array}{llcll}
(\beta) & (\lambda x.\ e)(v) & \longrightarrow & e[x{:=}v] & \\
(ret) & \mathsf{handle}_h(v) & \longrightarrow & e[x{:=}v] & \text{if } (\mathsf{return}\ x \to e) \in h \\
(hndl) & \mathsf{handle}_h(\mathsf{E}[op(v)]) & \longrightarrow & e[x{:=}v, r{:=}\lambda y.\mathsf{handle}_h(\mathsf{E}[y])] & \text{if } (op(x)\ r \to e) \in h \text{ and } op \notin \mathsf{bop}(\mathsf{E})
\end{array}
$$

$$
\frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\ [\textsc{eval}]
$$

Fig. 5. The language of algebraic effect handlers, λ_{aeh}

Figure 6 defines the type rules for λ_{aeh}. These are essentially the same as the rules defined by Leijen (2017b) except simplified to only use monomorphic types and ground constructors since that suffices for our purposes. As such, all proofs carry over mostly unchanged. In particular, Leijen (2017b) shows the following useful properties:

Theorem 6. (*Semantic Soundness*)
If $\emptyset \vdash_{aeh} e : \tau \mid \epsilon$ then either e diverges, or evaluates to a value $e \longmapsto^* v$ where $\emptyset \vdash_{val} v : \tau$.

Lemma 6. (*Effects are meaningful*)
If $\Gamma \vdash_{aeh} \mathsf{E}[op(v)] : \tau \mid \epsilon$ with $op \notin \mathsf{bop}(\mathsf{E})$ and $\Sigma(l) = \{ \ldots\ op : \tau' \ldots \}$, then $l \in \epsilon$.

This lemma basically states that effect types cannot be discarded except through handlers. It also implies that effects are meaningful, i.e. if a function does not have an exception effect, it will never throw an exception.

Syntax of Types:

Types	τ	::=	c	constants (constructors)
		|	$\tau \to \epsilon\ \tau$	functions
Effect Row	ϵ	::=	$\langle\rangle$	empty row
		|	$\langle l \mid \epsilon \rangle$	row extension
Constants	c	::=	$unit \mid bool \mid \ldots$	set of builtin types
Effect Labels	l	::=	$exn \mid \ldots$	set of effect constants
Type Environment	Γ	::=	$\emptyset \mid \Gamma, x : \tau$	
Effect Signatures	Σ	::=	$\emptyset \mid \Sigma, l : \{op_1 : \tau_1 \to \tau_1', \ldots, op_n : \tau_n \to \tau_n'\}$	

Type rules:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{val} x : \tau}\ [\textsc{var}] \qquad \frac{\Gamma, x : \tau_1 \vdash_{aeh} e : \tau_2 \mid \epsilon}{\Gamma \vdash_{val} \lambda x.e : \tau_1 \to \epsilon\ \tau_2}\ [\textsc{lam}]$$

$$\frac{\Gamma \vdash_{val} v : \tau}{\Gamma \vdash_{aeh} v : \tau \mid \epsilon}\ [\textsc{val}] \qquad \frac{\Gamma \vdash_{aeh} e_1 : \tau_1 \to \epsilon\ \tau_2 \mid \epsilon \quad \Gamma \vdash_{aeh} e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash_{aeh} e_1(e_2) : \tau_2 \mid \epsilon}\ [\textsc{app}]$$

$$\frac{\begin{array}{l}\Gamma \vdash_{aeh} e : \tau \mid \langle l \mid \epsilon \rangle \quad \Sigma(l) = \{op_1 : \tau_1 \to \tau_1', \ldots, op_n : \tau_n \to \tau_n'\} \quad \Gamma, x : \tau \vdash e_r : \tau_r \mid \epsilon \\ \Gamma, x : \tau_1, r_1 : \tau_1' \to \epsilon\ \tau_r \vdash_{aeh} e_1 : \tau_r \mid \epsilon \quad \ldots \quad \Gamma, x : \tau_n, r_n : \tau_n' \to \epsilon\ \tau_r \vdash_{aeh} e_n : \tau_r \mid \epsilon\end{array}}{\Gamma \vdash_{aeh} \mathsf{handle}\{op_1(x_1)\ r_1 \to e_1; \ldots; op_n(x_n)\ r_n \to e_n;\ \mathsf{return}\ x \to e_r\}(e) : \tau_r \mid \epsilon}\ [\textsc{handle}]$$

Fig. 6. Type rules for algebraic effect handlers, λ_{aeh}

4.1. Translation

Figure 7 defines a translation function $\lceil \cdot \rceil$ from λ_{db+} to λ_{aeh}, translating values, expressions, evaluation contexts, and signatures. Since we translate every implicit name binding p to a single handler with one operation, we make labels l and operation names op coincide with implicit names p, and effect rows ϵ with implicit rows π – which means we do not need to translate types at all.

The translation is type preserving:

Theorem 7. (*Type Preservation*)
If $\Gamma \vdash_{imp} e : \tau \mid \epsilon$, then $\Gamma \vdash_{aeh} \lceil e \rceil : \tau \mid \epsilon$.

Proof. Straightforward induction over the type rules of λ_{db+}. For example, for the case of rule WFUN we type $\Gamma \vdash_{imp}$ with fun $p(x) = e_1$ in $e_2 : \tau \mid \pi$. By induction, we can assume $\lceil \Sigma \rceil(p) = \{ p : \tau_1 \to \tau_2 \}$ **(1)**, $\Gamma, x : \tau_1 \vdash_{val} \lceil e_1 \rceil : \tau_2 \mid \pi$ **(2)** and $\Gamma \vdash_{aeh} e_2 : \tau \mid \langle p \mid \pi \rangle$ **(3)**. We need to verify now that we can check $\Gamma \vdash_{aeh} \lceil$with fun $p(x) = e_1$ in $e_2 \rceil : \tau \mid \pi$ which equals $\Gamma \vdash_{aeh}$ handle$\{p(x)\ r = r(\lceil e_1 \rceil)\}(\lceil e_2 \rceil)$ (with $r \notin \mathsf{fv}(e_1)$). Using rule HANDLE we can satisfy two of its premises using (1) (using $p = op$) and (3) (using $l = p$ and $\epsilon = \pi$). That leaves us to derive $\Gamma, x : \tau_1, r : \tau_1' \to \epsilon\ \tau_r \vdash_{aeh} r(\lceil e_1 \rceil) : \tau_r \mid \epsilon$ where $\tau_1' = \tau_2$ and due to the absence of a return clause, $\tau_r = \tau$. We can use rule $\textsc{app}_{aeh}$ now and check $\Gamma, x : \tau_1, r : \tau_2 \to \epsilon\ \tau_r \vdash_{aeh} \lceil e_1 \rceil : \tau_2 \mid \epsilon$. Since $r \notin \mathsf{fv}(e_1)$, it suffices to show $\Gamma, x : \tau_1 \vdash_{aeh} \lceil e_1 \rceil : \tau_2 \mid \epsilon$ which holds by (2).

This brings us to the main theorem that our translation preserves semantics too:

$\lceil\cdot\rceil\colon\ \lambda_{db+} \to \lambda_{aeh},\ \Sigma_{db+} \to \Sigma_{aeh},\ \mathsf{E}_{db+} \to \mathsf{E}_{aeh},\ b \to h$

$$
\begin{array}{lcl}
\lceil x \rceil & = & x \\
\lceil \lambda x.e \rceil & = & \lambda x.\lceil e \rceil
\end{array}
$$

$$
\begin{array}{lcl}
\lceil e(e') \rceil & = & \lceil e \rceil(\lceil e' \rceil) \\
\lceil p \rceil & = & p(\mathit{unit}) \\
\lceil p(v) \rceil & = & p(\lceil v \rceil) \\
\lceil \mathsf{with}\ b\ \mathsf{in}\ e \rceil & = & \mathsf{handle}\{\ \lceil b \rceil\ \}(\lceil e \rceil)
\end{array}
$$

$$
\begin{array}{lcl}
\lceil \mathsf{val}\ p\ =\ v \rceil & = & p(x)\ r \to r(\lceil v \rceil)\ \ \text{with } x, r \notin \mathsf{fv}(v) \\
\lceil \mathsf{fun}\ p(x)\ =\ e \rceil & = & p(x)\ r \to r(\lceil e \rceil)\ \ \text{with } r \notin \mathsf{fv}(e) \\
\lceil \mathsf{control}\ p(x)\ r\ =\ e \rceil & = & p(x)\ r \to \lceil e \rceil
\end{array}
$$

$$
\begin{array}{lcl}
\lceil \emptyset \rceil & = & \emptyset \\
\lceil \Sigma, p\ :\ \mathsf{val}\ \tau \rceil & = & \lceil \Sigma \rceil,\ p : \{p\ :\ \mathit{unit} \to \tau\ \} \\
\lceil \Sigma, p\ :\ \mathsf{fun}\ \tau \to \tau' \rceil & = & \lceil \Sigma \rceil,\ p : \{p\ :\ \tau \to \tau'\ \}
\end{array}
$$

$$
\begin{array}{lcl}
\lceil \Box \rceil & = & \Box \\
\lceil \mathsf{E}(e) \rceil & = & \lceil \mathsf{E} \rceil(\lceil e \rceil) \\
\lceil v(\mathsf{E}) \rceil & = & \lceil v \rceil(\lceil \mathsf{E} \rceil) \\
\lceil \mathsf{with}\ b\ \mathsf{in}\ \mathsf{E} \rceil & = & \mathsf{handle}\{\ \lceil b \rceil\ \}(\lceil \mathsf{E} \rceil)
\end{array}
$$

Fig. 7. Translating the language of implicit values, functions, and control λ_{db+} to algebraic effect handlers λ_{aeh}

Theorem 8. (*Semantic Soundness*)
If $e \longmapsto e'$ then $\lceil e \rceil \longmapsto^* \lceil e' \rceil$

To prove this, we need the following lemmas about the the translation:

Lemma 7. (*Translation preserves free variables*)
$\mathsf{fv}(e) = \mathsf{fv}(\lceil e \rceil)$

Lemma 8. (*Translation preserves bound implicits*)
$\mathsf{bp}(\mathsf{E}) = \mathsf{bop}(\lceil \mathsf{E} \rceil)$

Lemma 9. (*Translation preserves contexts*)
$\lceil \mathsf{E}[e] \rceil = \lceil \mathsf{E} \rceil[\lceil e \rceil]$

Lemma 10. (*Translation is substitution safe*)
$\lceil e[x{:=}v] \rceil = \lceil e \rceil[x{:=}\lceil v \rceil]$

Proof. (*Of Theorem 8*) The proof proceeds with induction over the reduction rules of λ_{db+}. Here we show the case for implicit functions (*dfun*), where $\mathsf{with\ fun}\ p(x) = e\ \mathsf{in}\ \mathsf{E}[p(v)]$ reduces to $(\lambda y.\ \mathsf{with\ fun}\ p(x) = e\ \mathsf{in}\ \mathsf{E}[y])(e[x{:=}v])$ with $p \notin \mathsf{bp}(\mathsf{E})$ **(1)**, and by Lemma 8, $p \notin \mathsf{bop}(\lceil \mathsf{E} \rceil)$ **(2)**. Using the translated handler, we can now derive

$$
\begin{aligned}
&\lceil \text{with fun } p(x) \;=\; e \text{ in } \mathsf{E}[p(v)] \rceil \\
&\quad = \; \{\text{with } r \notin \mathrm{fv}(e)\ (3)\ \} \\
&\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E}[p(v)] \rceil)\ \} \\
&\quad = \; \{\ \text{Lemma 9}\} \\
&\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E} \rceil[p(\lceil v \rceil)]) \\
&\quad \longrightarrow \{\ (2)\ \} \\
&(r(\lceil e \rceil)[x{:=}\lceil v \rceil, r{:=}\lambda y.\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E} \rceil[y])] \\
&\quad = \; \{\ \text{substitute}\ \} \\
&(\lambda y.\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E} \rceil[y])])(\lceil e \rceil[x{:=}\lceil v \rceil, r{:=}\lambda y.\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E} \rceil[y])]) \\
&\quad = \; \{\ (3) \text{ and Lemma 7}\ \} \\
&(\lambda y.\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E} \rceil[y])])(\lceil e \rceil[x{:=}\lceil v \rceil]) \\
&\quad = \; \{\ \text{Lemma 10}\} \\
&(\lambda y.\text{handle}\{\ p(x)\ r \to r(\lceil e \rceil)\}(\lceil \mathsf{E} \rceil[y])])(\lceil e[x{:=}v] \rceil) \\
&\quad = \; \{\ (3) \text{ and } \lceil . \rceil \text{ definition}\ \} \\
&\lceil (\lambda y.\ \text{with fun } p(x) \;=\; e \text{ in } \mathsf{E}[y])(e[x{:=}v]) \rceil
\end{aligned}
$$

□

5. RELATED WORK

Implicit values are perhaps most closely related to *implicit parameters* as described by Lewis, Launchbury, Meijer, and Shields (2000). In particular, implicit parameters are immutable, named, and statically typed. In contrast to our approach, implicit parameters do not need to be declared and can be used and bound at any type. This is flexible, but can lead to large types (as shown in Section 2.2.2) and delays possible type errors to the binding site. Lewis et al. (2000) show how implicit parameters can be elegantly implemented using regular parameter passing similar to the dictionary passing translation of type classes (Wadler and Blott, 1989; Jones, 1992). Such translation could certainly be applied to implicit values as well, turning every member p in an implicit row into an evidence parameter. For implicit functions and control, which manipulate the stack, this may work in combination with a corresponding stack *prompt* to delimit the dynamic scope. This technique of combining explicit parameter passing with prompts is used for example by the Scala Effekt library (Brachthäuser, 2019; Brachthäuser and Schuster, 2017) to efficiently implement algebraic effect handlers on the JVM.

In an untyped setting, dynamic binding first appeared in McCarthy Lisp (as a bug) (McCarthy, 1960). Modern dialects have lexical scoping but still provide dynamic binding: in Common Lisp one can use the `special` declaration (Steele Jr., 1990), and MIT Scheme has `fluid-let` bindings (Hanson, 1991). The semantics of dynamic binding was formalized by Moreau (1998). Kiselyov, Shan, and Sabry (2006) extend upon that work by giving a translation into delimited control operations, giving a unified framework for continuations, side-effects, and dynamic binding. Later, Kammar and Pretnar (2017) do a similar translation where they show how (mutable) dynamic variables can be expressed in terms of algebraic effect handlers – and our translation of local mutable variables in Section 2.4 is based on this. Forster, Kammar, Lindley, and Pretnar (2017) also show that in a untyped setting, algebraic effect handlers, delimited control, and monads, can all express each other through a local macro-translation (Felleisen, 1991) and thus all can express dynamic binding.

Instead of binding implicit values explicitly, there are many designs that resolve implicit bindings *implicitly* based on their type. The most commonly used are implicit parameters in Scala (Oliveira and Gibbons, 2010; Odersky, 2010; Odersky et al., 2017) where implicit parameters are declared on

a method signature but provided automatically at the call-site based on their type. Siek and Lumsdaine (2005) introduce system F^G which uses type based resolution for an implicit parameter mechanism used for concept-based generic programming. Haskell type classes (Wadler and Blott, 1989; Jones, 1992; Kiselyov and Shan, 2004) are another instance where dictionaries are passed implicitly and resolved based on their type. Oliveira, Schrijvers, Choi, Lee, and Yi (2012) describe the *implicit calculus* as a core formalization of implicit parameters that are resolved by their type and they discuss how the previous instances can be expressed in the implicit calculus. The implicit calculus is interesting as the implicit values are not only resolved by their type, but also referred to by their type and no explicit names are used – for example, `implicit 1 in implicit True in (even(?int) && ?bool)` evaluates to `False` where `?` is used for implicit type-based binding.

Algebraic effects (Plotkin and Power, 2003) and handlers (Plotkin and Pretnar, 2013) provide a categorical foundation to reason about (side) effects in programming languages, and are a powerful abstraction to describe all kinds of control structures. Various languages (Bauer and Pretnar, 2015; Lindley et al., 2017; Hillerström and Lindley, 2016; Dolan et al., 2015; Leijen, 2017b) and libraries (Wu et al., 2014; Brachthäuser et al., 2018; Brachthäuser, 2019; Leijen, 2017a) support algebraic effects nowadays. Leijen (2018b) (Section 5) describes a particular optimization for tail-resumptive effect handlers using "skip" frames to avoid capturing a continuation and directly evaluate such clause in the existing stack. This optimization applies naturally to the implementation of implicit functions as well and we use this in the current implementation in Koka to make implicit function calls very efficient.

6. CONCLUSION

We introduced two new language features based on implicit values in this article: implicit functions and implicit control. In particular, implicit functions are a small extension that creates new opportunities for abstraction while avoiding the need for full continuations in the implementation. We hope to see more languages that will support this feature.

REFERENCES

Andrej Bauer, and Matija Pretnar. "Programming with Algebraic Effects and Handlers." *J. Log. Algebr. Meth. Program.* 84 (1): 108–123. 2015. doi:10.1016/j.jlamp.2014.02.001.

Jonathan Immanuel Brachthäuser. "Effekt: Type- and Effect-Safe, Extensible Effect Handlers in Scala." Jan. 2019. `http://ps.informatik.uni-tuebingen.de/publications/brachthaeuser19effekt`. Under consideration for publication in the Journal of Functional Programming.

Jonathan Immanuel Brachthäuser, and Philipp Schuster. "Effekt: Extensible Algebraic Effects in Scala." In *Scala'17*. Vancouver, CA. Oct. 2017.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. "Effect Handlers for the Masses." In *International Conference on Object Oriented Programming Systems Languages & Applications*. ACM press, New York. 2018.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop*. Sep. 2015.

Matthias Felleisen. "On the Expressive Power of Programming Languages." *Science of Computer Programming* 17 (1-3). Elsevier: 35–75. 1991.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. "On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control." In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming*. ICFP'17. 2017. arXiv:1610.09161.

Ben R. Gaster, and Mark P. Jones. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham. 1996.

Chris Hanson. "MIT Scheme Reference Manual." Jan. 1991. Massachusetts Institute of Technology.

Daniel Hillerström, and Sam Lindley. "Liberating Effects with Rows and Handlers." In *Proceedings of the 1st International Workshop on Type-Driven Development*, 15–27. TyDe 2016. Nara, Japan. 2016. doi:10.1145/2976022.2976033.

Mark P. Jones. "A Theory of Qualified Types." In *4th. European Symposium on Programming (ESOP'92)*, 582:287–306. Lecture Notes in Computer Science. Springer-Verlag, Rennes, France. Feb. 1992. doi:10.1007/3-540-55253-7_17.

Ohad Kammar, and Matija Pretnar. "No Value Restriction Is Needed for Algebraic Effects and Handlers." *Journal of Functional Programming* 27 (1). Cambridge University Press. Jan. 2017.

David J. King, and John Launchbury. "Structuring Depth-First Search Algorithms in Haskell." In *22nd ACM Symp. on Principles of Programming Languages (POPL'95)*, 344–354. San Francisco, California, United States. 1995.

Oleg Kiselyov, and Chung-chieh Shan. "Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types." In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 33–44. Haskell '04. ACM, Snowbird, Utah, USA. 2004.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. "Delimited Dynamic Binding." In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 26–37. ICFP '06. ACM, New York, NY, USA. 2006.

Daan Leijen. "Extensible Records with Scoped Labels." In *Proceedings of the 2005 Symposium on Trends in Functional Programming*, 297–312. 2005.

Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types." In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. 2014. doi:10.4204/EPTCS.153.8.

Daan Leijen. "Implementing Algebraic Effects in C: Or Monads for Free in C." Edited by Bor-Yuh Evan Chang. *Programming Languages and Systems*, LNCS, 10695 (1). Springer International Publishing, Suzhou, China: 339–363. 2017. APLAS'17, Suzhou, China.

Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

Daan Leijen. "First Class Dynamic Effect Handlers: Or, Polymorphic Heaps with Dynamic Effect Handlers." In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, 51–64. TyDe 2018. ACM, St. Louis, MO, USA. 2018. doi:10.1145/3240719.3241789.

Daan Leijen. *Algebraic Effect Handlers with Resources and Deep Finalization*. MSR-TR-2018-10. Microsoft Research. Apr. 2018.

Daan Leijen. "Koka Repository." 2019. `https://github.com/koka-lang/koka`.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. "Implicit Parameters: Dynamic Scoping with Static Types." In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 108–118. POPL'00. ACM, Boston, MA, USA. 2000.

Sam Lindley, Connor McBride, and Craig McLaughlin. "Do Be Do Be Do." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 500–514. Paris, France. Jan. 2017. doi:10.1145/3009837.3009897.

John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Commun. ACM* 3 (4). ACM: 184–195. Apr. 1960. doi:10.1145/367177.367199.

Luc Moreau. "A Syntactic Theory of Dynamic Binding." *Higher-Order and Symbolic Computation* 11 (3): 233–279. Sep. 1998. doi:10.1023/A:1010087314987.

Martin Odersky. "The Scala Language Specification, 2.8." 2010. `https://docs.scala-lang.org/tour/implicit-parameters.html`.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. "Simplicitly: Foundations and Applications of Implicit Function Types." *Proc. ACM Program. Lang.* 2 (POPL). ACM, New York, NY, USA: 42:1–42:29. Dec. 2017. doi:10.1145/3158130.

Bruno C. d. S. Oliveira, and Jeremy Gibbons. "Scala for Generic Programmers." *Journal of Functional Programming* 20 (3–4): 303–352. Oct. 2010.

Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. "The Implicit Calculus: A New Foundation for Generic Programming." In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 35–44. PLDI '12. ACM, Beijing, China. 2012.

Simon L Peyton Jones, and John Launchbury. "State in Haskell." *Lisp and Symbolic Comp.* 8 (4): 293–341. 1995. doi:10.1007/BF01018827.

Gordon D. Plotkin, and John Power. "Algebraic Operations and Generic Effects." *Applied Categorical Structures* 11 (1): 69–94. 2003. doi:10.1023/A:1023064908962.

Gordon D. Plotkin, and Matija Pretnar. "Handling Algebraic Effects." In *Logical Methods in Computer Science*, volume 9. 4. 2013. doi:10.2168/LMCS-9(4:23)2013.

Didier Rémy. "Type Inference for Records in Natural Extension of ML." In *Theoretical Aspects of Object-Oriented Programming*, 67–95. 1994. doi:10.1.1.48.5873.

Jeremy G. Siek, and Andrew Lumsdaine. "Essential Language Support for Generic Programming." In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 73–84. PLDI '05. ACM, Chicago, IL, USA. 2005. doi:10.1145/1065010.1065021.

Guy Lewis Steele Jr. *Common Lisp. The Language, Second Edition*. Digital Press. 1990.

Philip Wadler, and Stephen Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc." In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 60–76. POPL '89. ACM, Austin, Texas, USA. 1989. doi:10.1145/75277.75283.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect Handlers in Scope." In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell '14. Göthenburg, Sweden. 2014. doi:10.1145/2633357.2633358.

A. FURTHER PROOFS

Proof. (*Of Lemma 7*) We show free variables are preserved by the translation by induction over the structure of expressions. We use fv in the proof for fv_{aeh}.
case $\mathsf{fv}_{db+}(x)$: This is $\{x\}$ which is equal to $\mathsf{fv}(x)$ which equals $\mathsf{fv}(\lceil x \rceil)$ by the definition of $\lceil \cdot \rceil$.
case $\mathsf{fv}_{db+}(\lambda x.e)$: This is $\mathsf{fv}_{db+}(e) - \{x\}$, and by induction $\mathsf{fv}(\lceil e \rceil) - \{x\} = \mathsf{fv}(\lambda x.\lceil e \rceil) = \mathsf{fv}(\lceil \lambda x.e \rceil)$.
case $\mathsf{fv}_{db+}(e_1(e_2))$: This is $\mathsf{fv}_{db+}(e_1) \cup \mathsf{fv}_{db+}(e_2)$ and by induction $\mathsf{fv}(\lceil e_1 \rceil) \cup \mathsf{fv}(\lceil e_2 \rceil) = \mathsf{fv}(\lceil e_1 \rceil(\lceil e_2 \rceil))$ $= \mathsf{fv}(\lceil e_1(e_2) \rceil)$.
case $\mathsf{fv}_{db+}(\mathsf{with\ fun}\ p(x)\ =\ e_1\ \mathsf{in}\ e_2)$: This is $(\mathsf{fv}_{db+}(e_1) - \{x\}) \cup \mathsf{fv}_{db+}(e_2)$, and by induction we have $(\mathsf{fv}(\lceil e_1 \rceil) - \{x\}) \cup \mathsf{fv}(\lceil e_2 \rceil) = (\mathsf{fv}(\lceil e_1 \rceil) - \{x, r\}) \cup \mathsf{fv}(\lceil e_2 \rceil)$ for a fresh $r \notin \mathsf{fv}(\lceil e_1 \rceil)$, and thus $\mathsf{fv}(\mathsf{handle}\{\ op(x)\ r \to \lceil e_1 \rceil\ \}(\lceil e_2 \rceil)) = \mathsf{fv}(\lceil \mathsf{with\ fun}\ p(x)\ =\ e_1\ \mathsf{in}\ e_2 \rceil)$.
case $\mathsf{fv}_{db+}(\mathsf{with\ control}\ p(x)\ r\ =\ e_1\ \mathsf{in}\ e_2)$: Similar to the previous case.
case $\mathsf{fv}_{db+}(\mathsf{with\ val}\ p\ =\ v\ \mathsf{in}\ e_2)$: Similar to the previous cases. □

Proof. (*Of Lemma 8*) We show how bound parameters are preserved, where $\mathsf{bp}(\mathsf{E}) = \mathsf{bop}(\lceil \mathsf{E} \rceil)$. We use induction over the structure of the evaluation context.
case $\mathsf{bp}(\square) = \emptyset = \mathsf{bop}(\square) = \mathsf{bop}(\lceil \square \rceil)$.
case $\mathsf{bp}(\mathsf{E}(e)) = \mathsf{bp}(\mathsf{E})$, and by induction, $= \mathsf{bop}(\lceil \mathsf{E} \rceil) = \mathsf{bop}(\lceil \mathsf{E} \rceil(\lceil e \rceil)) = \mathsf{bop}(\lceil \mathsf{E}(e) \rceil)$.
case $\mathsf{bp}(v(\mathsf{E})) = \mathsf{bp}(\mathsf{E})$, and by induction, $= \mathsf{bop}(\lceil \mathsf{E} \rceil) = \mathsf{bop}(\lceil v \rceil(\lceil \mathsf{E} \rceil) = \mathsf{bop}(\lceil v(\mathsf{E}) \rceil)$.
case $\mathsf{bp}(\mathsf{with}\ b\ \mathsf{in}\ \mathsf{E}) = \mathsf{bp}(b) \cup \mathsf{bp}(\mathsf{E})$, and by induction, $\mathsf{bp}(b) \cup \mathsf{bop}(\lceil \mathsf{E} \rceil)$. We now need to do a case analysis on b to show $\mathsf{bp}(b) = \mathsf{bop}(\lceil b \rceil)$. We show the case for fun here: $\mathsf{bp}(\mathsf{fun}\ p(x) \to e') = \{p\}$ $= \mathsf{bop}(op(x)\ r \to \lceil e' \rceil) = \mathsf{bop}(\lceil b \rceil)$. We can now derive $\mathsf{bop}(\lceil b \rceil) \cup \mathsf{bop}(\lceil \mathsf{E} \rceil) = \mathsf{bop}(\lceil \mathsf{with}\ b\ \mathsf{in}\ \mathsf{E} \rceil)$. □

Proof. (*Of Lemma 9*) We show that the translation preserves contexts where $\lceil \mathsf{E}[e] \rceil = \lceil \mathsf{E} \rceil[\lceil e \rceil]$. We proceed by induction over the structure of the evaluation context:
case $\lceil \square[e] \rceil = \lceil e \rceil = \square[\lceil e \rceil] = \lceil \square \rceil[\lceil e \rceil]$.
case $\mathsf{E}' = \mathsf{E}(e')$: then $\lceil (\mathsf{E}[e])(e') \rceil = \lceil \mathsf{E}[e] \rceil(\lceil e' \rceil)$, and by induction $= \lceil \mathsf{E} \rceil[\lceil e \rceil](\lceil e' \rceil) = \lceil \mathsf{E}' \rceil[\lceil e \rceil]$.
case $\mathsf{E}' = v(\mathsf{E})$: then $\lceil v(\mathsf{E}[e]) \rceil = \lceil v \rceil(\lceil \mathsf{E}[e] \rceil)$, and by induction $= \lceil v \rceil(\lceil \mathsf{E} \rceil[\lceil e \rceil]) = \lceil \mathsf{E}' \rceil[\lceil e \rceil]$.
case $\mathsf{E}' = \mathsf{with}\ b\ \mathsf{in}\ \mathsf{E}$: then $\lceil \mathsf{with}\ b\ \mathsf{in}\ \mathsf{E}[e] \rceil = \mathsf{handle}\{\lceil b \rceil\}(\lceil \mathsf{E}[e] \rceil)$, and by induction we have, $= \mathsf{handle}\{\lceil b \rceil\}(\lceil \mathsf{E} \rceil[\lceil e \rceil]) = \lceil \mathsf{with}\ b\ \mathsf{in}\ \mathsf{E} \rceil[\lceil e \rceil] = \lceil \mathsf{E}' \rceil[\lceil e \rceil]$.

Proof. (*Of Lemma 10*) We show that the translation preserves substitution where $\lceil e[x{:=}v] \rceil$ equals $\lceil e \rceil[x{:=}\lceil v \rceil]$. We do this by induction over the size of the expressions. We only show the most interesting cases:
case $\lceil x[x{:=}v] \rceil = \lceil v \rceil = x[x{:=}\lceil v \rceil] = \lceil x \rceil[x{:=}\lceil v \rceil]$.
case $\lceil y[x{:=}v] \rceil$ where $x \neq y$, then this equals $\lceil y \rceil = y = y[x{:=}\lceil v \rceil] = \lceil y \rceil[x{:=}\lceil v \rceil]$.
case $\lceil (\mathsf{with}\ b\ \mathsf{in}\ e)[x{:=}v] \rceil = \lceil \mathsf{with}\ b[x{:=}v]\ \mathsf{in}\ e[x{:=}v] \rceil = \mathsf{handle}\{\lceil b[x{:=}v] \rceil\}(\lceil e[x{:=}v] \rceil)$, and by induction $\mathsf{handle}\{\lceil b[x{:=}v] \rceil\}(\lceil e \rceil[x{:=}\lceil v \rceil])$. Assuming $\lceil b[x{:=}v] \rceil = \lceil b \rceil[x{:=}\lceil v \rceil]$ **(1)**, we can derive that $\mathsf{handle}\{\lceil b \rceil[x{:=}\lceil v \rceil]\}(\lceil e \rceil[x{:=}\lceil v \rceil]) = \mathsf{handle}\{\lceil b \rceil\}(\lceil e \rceil)[x{:=}\lceil v \rceil]) = \lceil \mathsf{with}\ b\ \mathsf{in}\ e \rceil[x{:=}\lceil v \rceil]$. So, it remains to show (1). We do case analysis on b, and show the case for $b = \mathsf{control}\ p(x)\ r \to e'$, where we can α-rename to ensure $x \neq y$ and $x \neq r$, we have $\lceil b[x{:=}v] \rceil = \lceil (\mathsf{control}\ p(x)\ r \to e')[x{:=}v] \rceil = \lceil \mathsf{control}\ p(x)\ r \to e'[x{:=}v] \rceil = op(x)\ r \to \lceil e'[x{:=}v] \rceil$, and by induction, $op(x)\ r \to \lceil e' \rceil[x{:=}\lceil v \rceil] = (op(x)\ r \to \lceil e' \rceil)[x{:=}\lceil v \rceil] = \lceil \mathsf{control}\ p(x)\ r \to e' \rceil[x{:=}\lceil v \rceil]\ \lceil b \rceil[x{:=}\lceil v \rceil]$. □

B. ALGEBRAIC EFFECT HANDLERS AS IMPLICIT CONTROL

After seeing how algebraic effect handlers can express implicit values, functions, and control, you may wonder if the opposite is possible where implicits can express any algebraic effect handler? We believe this is the case.

A translation is not directly obvious though since the effect handler calculus is much richer with return clauses and grouping of effect operations. We therefore take a layered approach where we first translate λ_{aeh} to λ_{kr} which does not contain any return clauses. From there we show a translation to λ_{k1} where every handler contains just one operation clause. Finally, we can then show how λ_{k1} translates to the implicit calculus.

B.1. Removing Return.

One may first suppose that we can easily translate return clauses away by pushing the clause into the handler body, e.g.

$$\lfloor \mathsf{handle}\{\ \ldots;\ \mathsf{return}\ x \to e_r\}(e)\ \rfloor\ =\ \mathsf{handle}\{\ \ldots\ \}((\lambda x.\ \lfloor e_r \rfloor)(\lfloor e \rfloor)) \quad \text{(wrong)}$$

but that is not correct: if e_r contains a reference to an operation op in the handler itself it would now be handled by the same handler instead of an outer one! Leijen (2018a) (Section 3.2) shows how to translate this correctly by assuming an extra "return" operation op_{ret} for every effect l that handles non-trivial return clauses:

$$\lfloor \mathsf{handle}\{\ \ldots;\ \mathsf{return}\ x \to e_r\}(e) \rfloor\ =\ \mathsf{handle}\{\ \lfloor \ldots \rfloor;\ op_{ret}(x)\ r \to \lfloor e_r \rfloor\ \}(op_{ret}(\lfloor e \rfloor)) \quad \text{with } r \notin \mathsf{fv}(e_r)$$

We can show that this preserves the original return semantics; Starting from the return reduction, we have:

$$\begin{array}{l}
\lfloor \mathsf{handle}\{\ldots;\ \mathsf{return}\ x \to e_r\}(v) \rfloor \\
=\ \{\mathsf{with}\ r \notin \mathsf{fv}(e_r)\ (\mathbf{1})\ \} \\
\mathsf{handle}\{\lfloor \ldots \rfloor;\ op_{ret}(x)\ r \to \lfloor e_r \rfloor\}(op_{ret}(\lfloor v \rfloor)) \\
\longrightarrow \\
\lfloor e_r \rfloor[x{:=}\lfloor v \rfloor, r{:=}\lfloor \ldots \rfloor] \\
=\ \{\ (1)\ \} \\
\lfloor e_r \rfloor[x{:=}\lfloor v \rfloor] \\
= \\
\lfloor e_r[x{:=}v] \rfloor
\end{array}$$

which is equivalent to the original rule.

B.2. Single Operation Handlers.

Translating a group of operation clauses to a group of handlers with each just one operation suffers from the same problem as the naive return translation as clauses would start to handle operations from each other. Consider the following obvious but wrong translation:

$$
\begin{array}{l}
\lfloor \mathsf{handle}\{op_1(x_1)\ r_1 \to e_1;\ \ldots;\ op_n(x_n)\ r_n \to e_n;\ op_{ret}(x_r)\ r_r \to e_r\ \}(e) \rfloor \\
= \\
\mathsf{handle}\{\ op_1(x_1)\ r_1 \to \lfloor e_1 \rfloor\ \}(\\
\ \ldots \\
\ \ \mathsf{handle}\{\ op_n(x_n)\ r_n \to \lfloor e_n \rfloor\ \}(\\
\ \ \ \mathsf{handle}\{\ op_{ret}(x_r)\ r_r \to \lfloor e_r \rfloor\ \}(\lfloor e \rfloor)) \ldots)
\end{array}
$$

If now e_r invokes $op_1(v)$ it is handled by the op_1 clause in the current handler instead of an outer one as in the original semantics. To work around this problem one possible approach is to encode the operations in a data type under a single operation op_l where every operation op_i becomes a constructor. This is troublesome from a typing perspective though as the type of the resume binding differs for each operation (and we need GADT's or some other form of dependent typing).

Another solution exist though that needs no extensions to our calculus where we use "shadow" operations. For every operation op, we introduce a shadow operation op', and translate every handler with multiple operations into singleton handlers for both an operation and its shadow:

$$
\begin{array}{l}
\lfloor \mathsf{handle}\{op_1(x_1)\ r_1 \to e_1;\ \ldots;\ op_n(x_n)\ r_n \to e_n;\ op_{ret}(x_r)\ r_r \to e_r\ \}(e) \rfloor \\
= \\
\mathsf{handle}\{\ op'_1(x_1)\ r_1 \to \lfloor e_1 \rfloor\ \}(\\
\ \ldots \\
\ \ \mathsf{handle}\{\ op'_n(x_n)\ r_n \to \lfloor e_n \rfloor\ \}(\\
\ \ \ \mathsf{handle}\{\ op'_{ret}(x_r)\ r_r \to \lfloor e_r \rfloor\ \}(\\
\ \ \ \ \mathsf{handle}\{\ op_1(x_1)\ r_1 \to r_1(op'_1(x_1))\ \}(\\
\ \ \ \ \ \ldots \\
\ \ \ \ \ \mathsf{handle}\{\ op_n(x_n)\ r_n \to r_n(op'_n(x_n))\ \}(\\
\ \ \ \ \ \ \mathsf{handle}\{\ op_{ret}(x_r)\ r_r \to r_r(op'_{ret}(x_r))\ \}(\\
\ \ \ \ \ \ \ (\lfloor e \rfloor))) \ldots))) \ldots)
\end{array}
$$

In the new translation, every operation op immediately forwards to its shadow operation op'. By construction, every expression clause e_i contains no reference to a shadow operation so they do not handle each other. Since every shadow operation is immediately discharged these operations never show in the types, or anywhere else in the program. Typing is otherwise preserved, except that every effect l is now broken up in its individual operations:

$$
\lfloor l \rfloor\ =\ op_1 | \ldots | op_n \quad \text{where } \Sigma(l)\ =\ \{op_1 : \tau_1 \to \tau'_1,\ \ldots,\ op_n : \tau_n \to \tau'_n\}
$$

$$
\begin{array}{l}
\lfloor \emptyset \rfloor\ =\ \emptyset \\
\lfloor \Sigma, l : \{op_1 : \tau_1 \to \tau'_1,\ \ldots,\ op_n : \tau_n \to \tau'_n\} \rfloor \\
\quad =\ \lfloor \Sigma \rfloor, op_1 : \{op_1 : \tau_1 \to \tau'_1\}, op'_1 : \{op'_1 : \tau_1 \to \tau'_1\}, \ldots, op_n : \{op_n : \tau_n \to \tau'_n\}, op'_n : \{op'_n : \tau_n \to \tau'_n\}
\end{array}
$$

B.3. Translating Back

Now that we can translate λ_{aeh} to λ_{k1}, we only need to consider single operation handlers without return clauses where every effect l contains just one operation op where we assume every effect is named after its operation, i.e. $l\ =\ op$. The translation is straightforward now:

$$
\begin{array}{ll}
\lfloor x \rfloor & = x \\
\lfloor \lambda x.e \rfloor & = \lambda x.\lfloor e \rfloor
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor e(e') \rfloor & = & \lfloor e \rfloor (\lfloor e' \rfloor) \\
\lfloor op(v) \rfloor & = & op(\lfloor v \rfloor) \\
\lfloor \mathsf{handle}\{\ p(x)\ r \to r(e_1)\ \}(e_2) \rfloor & = & \mathsf{with\ fun}\ op(x)\ =\ \lfloor e_1 \rfloor\ \mathsf{in}\ \lfloor e_2 \rfloor \quad \text{if } r \notin \mathsf{fv}(e_1) \\
\lfloor \mathsf{handle}\{\ p(x)\ r \to e_1\ \}(e_2) \rfloor & = & \mathsf{with\ control}\ op(x)\ r\ =\ \lfloor e_1 \rfloor\ \mathsf{in}\ \lfloor e_2 \rfloor
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor \emptyset \rfloor & = & \emptyset \\
\lfloor \Sigma, op\ :\ \{op\ : \tau_1 \to \tau_2\ \} \rfloor & = & \Sigma, op\ :\ \mathsf{fun}\ \tau_1 \to \tau_2
\end{array}
$$

We can show that the translation preserves both types and semantics.

Theorem 9. (*Type Preservation*)
If $\Gamma \vdash_{aeh} e\ :\ \tau \mid \epsilon$ then also $\Gamma \vdash_{imp} \lfloor e \rfloor\ :\ \tau \mid \epsilon$.

Theorem 10. (*Semantic Soundness*)
If $e \longmapsto_{aeh} e'$ then also $\lfloor e \rfloor \longmapsto^*_{imp} \lfloor e' \rfloor$.