Algebraic Effect Instance Scopes

(Zakresy instancji efektów algebraicznych)

Patrycja Balik

Praca inżynierska

Promotor: Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

7 września 2022

Abstract

Algebraic effects offer a modern solution to dealing with computational effects. They have seen widespread academic interest, and are now making their way into industrial-strength programming languages as well. One important topic of discussion in this area is how to use multiple instances of one effect while preserving static guarantees and convenience of use. The former question has been addressed by the introduction of lexically scoped instances [3]. However, while easier to use than the earlier effect coercions [2] and adaptors [5], it can still be inconvenient in large programs relying on a lot of instances, in many cases requiring explicit instance variable binding and passing. To remedy this we propose a systematic approach that extends lexically scoped instances. The key element is the separation of the notion of instance *names*, as given by the programmer, and instance *variables*, which allows us to use named instance application without sacrificing α-conversion. Once the distinction is made, we can add two useful mechanisms. The first is to allow instances to be declared as *implicit* for a set of function definitions, which can all become parametrized with the instance. The second mechanism is that instance parameters can be instantiated automatically based on the instance names associated with the function and present in the environment. We give formal semantics for a toy calculus equipped with these features via type-directed translation into a more standard calculus. To determine whether this approach is practical, we implement a proof of concept programming language, which in addition to the aforementioned mechanisms, offers other functionality useful for writing large programs, and demonstrate its use.

Streszczenie

Efekty algebraiczne oferują nowoczesne rozwiązanie dla zarządzania efektami obliczeniowym. Spotkały się z szerokim zainteresowaniem ze strony społeczności naukowej, i zaczynają pojawiać się także w przemysłowych językach programowania. Jednym z istotnych tematów dyskusji w tej dziedzinie jest to, jak używać wielu instancji tego samego efektu, jednocześnie zachowując zarówno statyczne gwarancje jak i wygodę użytkowania. W pierwszej z tych kwestii satysfakcjonujące rozwiązanie zostało znalezione poprzez dodanie leksykalnie wiązanych instancji [3]. Choć są prostsze w użyciu niż wcześniejsze koercje [2] lub adaptory [5], to nadal w dużych programach z wieloma instancjami często konieczne jest jawne wiązanie i przekazywanie zmiennych instancyjnych. By temu zaradzić, proponujemy systematyczne podejście rozszerzające leksykalne wiązanie instancji. Kluczowe okazuje się być rozdzielenie koncepcji *nazw* instancji od *zmiennych* instancyjnych, co pozwala na użycie aplikacji opartej na nazwach bez utraty α-konwersji. Gdy dokonamy tego rozróżnienia, możemy dodać dwa przydatne mechanizmy. Pierwszy z nich polega na możliwości zadeklarowania *niejawnych* instancji dla pewnego zbioru definicji funkcji, które mogą zostać sparametryzowane taką instancją. Drugi mechanizm pozwala na automatyczną instancjację parametrów instancyjnych na podstawie nazw powiązanych z funkcją i aktualnego środowiska. Przedstawiamy formalną semantykę dla niewielkiego rachunku wyposażonego w te elementy poprzez tłumaczenie sterowane typami do bardziej standardowego rachunku. Aby określić, czy to podejście jest praktyczne, zaimplementowaliśmy eksperymentalny język programowania, który poza wspomnianymi mechanizmami, oferuje też inne elementy przydatne do pisania dużych programów, i którego użycie zademonstrujemy.

Contents

Chapter 1

Introduction

Programs that never interact with their environment would not be very useful. Computational effects, such as input and output, stateful computation, exceptions, and backtracing are ubiquitous in programming, yet they are also a large source of errors due to subtle and unexpected interactions. In recent years, algebraic effects, as introduced by Plotkin and Power [10], and later equipped with effect handlers by Plotkin and Pretnar [11], have emerged as a promising and modern solution to managing computational effects, building on the idea that the invocation of an effect can be separated from its semantics. In the following section, we will briefly introduce algebraic effects. For a more thorough introduction, the reader may also consult the Overview section of Leijen's *Type Directed Compilation of Row-typed Algebraic Effects* [6], or another introduction due to Pretnar [13].

1.1 Algebraic Effects

So, how do we model a computational effect? Essentially, an effect is specified by the list of its *operations* (also called a *signature*). The operations may be called with some number of arguments, much like normal functions. In a typed setting, we will write the type of an operation with a bold arrow in the format of $\texttt{arg}_1$ $\texttt{arg}_2$ $\texttt{arg}_3$ $\ldots \Rightarrow$ `result`. By themselves, the operations have no specified meaning. To interpret the effect, an expression making use of its operations needs to be wrapped in an effect *handler*. This affords a high degree of modularity, as effectful functions may be defined independently of the semantics of their effects.

As a familiar example, exceptions may be modeled by an effect defined by the single `throw` operation, which receives some additional information about the exception that occurred (parametrized with the type variable `e` in Listing 1.1). We can use the empty type `Void` as its result type, because the operation should never return the control flow of the program to the point of its call, in line with how exceptions are usually implemented.

```
signature Exc e = throw : e ⇒ Void
```

Listing 1.1: The signature of the exception effect.

Much like languages that implement exceptions directly, the meaning of exception-throwing code is given by surrounding it with a handler that specifies what to do in case an exception occurs. For a somewhat silly example, consider Listing 1.2. If `b` is true, the value of the entire snippet is simply `1`. But if it is false, an exception is thrown, carrying with it the argument `0`. Then it gets "caught" by the enclosing handler, which overrides the result of the whole expression with `0`.

```
handle
  if b then 1 else throw 0
with
| throw n → n
end
```

Listing 1.2: An exception handler.

Algebraic effects are not restricted to operations that never return. In the clause which handles an operation, we may call `resume` with some value of the expected result type, and the computation will be restored at the point of the operation call, but replacing it with the result. An interesting fact is that `resume` is not a dedicated keyword, but rather just another variable associated with the *resumption*, which can be used like any other function and represents the rest of the computation waiting for a result from the operation. In Listing 1.3, we pair the classic `Reader` effect with a handler that resumes with a constant value. Each operation use is evaluated to `21`, and the final answer is `42`.

```
signature Reader a = ask : () ⇒ a

let x =
  handle ask () + ask () with
  | ask () → resume 21
  end
```

Listing 1.3: The `Reader` effect.

As it happens, `resume` may even be invoked multiple times, returning from the operation more than once. This renders effects such as nondeterminism trivial to implement in their most basic form. The handler for nondeterminism in Listing 1.4 produces a list of all the possible outcomes of the computation. It includes an additional `return` clause, which is not an operation, but specifies the behavior of the handler on plain values. When omitted, it is considered to be the identity function, but in this case, we want to wrap any individual values in singleton lists. In the clause for `flip`, we resume for both `True` and `False` and then concatenate the results. The outcome of the entire handler is the list `[('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')]`, which encompasses all the possibilities.

```
signature Nondet = flip : () ⇒ Bool

let x =
  handle
    let x = if flip () then 'a' else 'b' in
    let y = if flip () then 'c' else 'd' in
    (x, y)
  with
  | flip ()  → resume True @ resume False
  | return x → [x]
  end
```

Listing 1.4: A possible implementation of nondeterminism.

Functions may be defined with outstanding effects that are triggered once they are supplied with an argument. With a type-and-effect system, function types will be annotated with the effects that are used and unhandled. In Listing 1.5, the type of `f` is `Int →[Nondet, Reader] Int`. An effectful function can be called within a handler with the expected result of handling its effects.

```
let f x = if flip () then ask () else x + 1

let _ =
  handle handle f 0 with
  | tell () → ...
  end
  with
  | flip () → ...
  end
```

Listing 1.5: A simple function making use of effects.

1.2 Motivation

So far, we have not considered the situation in which more than one instance of the same effect is necessary. Addressing this need is nontrivial, and many of today's implementations forgo this feature altogether.

```
signature State = put : Int ⇒ () | get : () ⇒ Int
let inc () = put (get () + 1)
```

Listing 1.6: The `State` effect.

Consider the `State` effect (Listing 1.6), which models a single mutable cell, along with operations to store and retrieve its value. Should you need to store any additional state, you will have to define a brand new effect differing only in the operation names. The `inc` function and any others will need to be redefined as well. This workaround scales rather poorly—what of libraries, which may define a single fixed effect and a large collection of functions that make use of it? Several solutions have been proposed to permit a multitude of separate objects, called *instances*, that can be associated with the same effect and allow operations to be linked to the correct handler: the dynamic instances of some versions of Eff [1] or the lexically scoped instances of Helium [3]. In this work we will focus on the latter solution, as it offers a static type-and-effect system that we can build upon.

```
let inc 'a () = put 'a (get 'a () + 1)

let swap 'a 'b () =
  let x = get 'a () in
  put 'a (get 'b ());
  put 'b x
```

Listing 1.7: `State` with effect instances.

In Listing 1.7, we are able to implement a `swap` function that takes two mutable cells and swaps their contents by supplying the `put` and `get` operations with the correct instances. The new type of `inc` is `∀ 'a : State. () →['a] ()`, while the type of `swap` is `∀ 'a : State. ∀ 'b : State. () →['a, 'b] ()`. Note that the instance variables need to be bound in types as well, since now effect instances, and not effects, are listed in function types.

Lexically scoped instances can be bound explicitly as function arguments, which is reflected in the type of the function. Then, instances that are lexically in scope may be used with operations, or passed as arguments to other functions. Unlike normal variables, instance variables do not enjoy a first-class status in the language, and can only appear as a part of specific constructs, and not as general expressions.

Effect handlers also act as instance binders. For example, in `h_state` defined below, an instance is bound by the handler and passed to a function, implementing a reusable handler for the `State` effect. The handler evaluates to a function that takes some initial state. This entails that the result of calling the resumption is also such a function, so in the clauses for `get` and `put` it needs to be applied to the current state.

```
let h_state init f =
  handle 'a in f 'a () with
  | get _ → λs. resume s s
  | put s → λ_. resume () s
  | return x  → λ_. x
  end init
```

Listing 1.8: Effect handlers with instances.

In large programs, there will often be effects that are used across multiple functions and which are frequently passed down in function calls. Without instances, none of this effect binding and passing is necessary, but we sacrifice flexibility; with instances, definitions and calls can become unwieldy and verbose. Worse yet is modifying the code to extend it with an additional effect. What seems like a local change to a few functions deep down in the call tree may cascade to other functions which merely take a new argument and propagate it.

How to lower the overhead of using instances for the programmer, while maintaining the advantages? The current implementation of Helium offers some ad-hoc support for automatic abstraction and instantiation of instances based on their signatures. This works quite well in cases when only one instance of an effect is being used, but it is not clear how to generalize this method in a systematic way.

In this work, we propose an alternative solution which relies on instance names in order to determine how to perform the necessary generalizations and instantiations. Inspired by the section mechanism of Coq [15], described further in Section 5.2, we introduce implicit instance declarations, which specify instances eligible for generalization in let-definitions. In the process, we are forced to decouple the instance names given by the programmer from the notion of instance variables, and replace positional instance application with named application.

```
instance 'a : State
instance 'b : State

let inc () = put 'a (get 'a () + 1)

let swap () =
  let x = get 'a () in
  put 'a (get 'b ());
  put 'b x

let f () = inc (); inc (); inc {'a = 'b} (); swap ()
```

Listing 1.9: Implicit instances in use.

In Listing 1.9, `'a` ad `'b` are marked as implicit. Subsequently, `inc` is parametrized with `'a`, and `swap` and `f` are both parametrized with `'a` and `'b`. In the body of `f`, two uses of `inc` can be applied to `'a` automatically, while the final occurrence requires explicit named application to increment `'b` instead. Both instance arguments are supplied automatically to `swap`.

In the rest of this thesis, we present the type system and semantics of a simplified calculus that includes the proposed mechanisms. Since these mechanisms concern the surface language as exposed to the programmer, we are primarily interested in matters of language design. In order to evaluate this approach, an interpreter for a proof of concept language based on the calculus has been implemented and is distributed alongside this work[1]. It integrates this scheme with standard ML-style polymorphism and extends it with a variety of useful language features. As we set

[1]The source code of the implementation may be obtained from the Archive of Diploma Theses system at `https://apd.uni.wroc.pl`.

out to determine whether this approach is useful, easy to use and intuitive, we will analyze an example program implemented in the language in detail.

Chapter 2

The Calculus

In this chapter we will in fact present two different calculi. Of most interest is the *source* calculus, upon which we will base the design of our language. Its type-and-effect system will be presented, including the subtyping relation, which allows for effect instance generalization and instantiation. We will specify its semantics via type-directed translation into a second *target* calculus. Both calculi are based on the calculi developed by Biernacki et al. [3], with lexical scoping of effect instance binders, but without polymorphic signatures. Compared to the provided implementation, the calculi are simplified to only include the elements necessary to formalize the core ideas of the contribution, for the benefit of a concise presentation.

2.1 Syntax of the Source Calculus

The syntax of the core calculus, featuring the syntactic categories of expressions and definitions, is presented in Figure 2.1.

$$\begin{array}{rl}
\text{Variables} & r, x, y, z, \ldots \\
\text{Instance Names} & a, b, c, \ldots \\
\text{Definitions} & d ::= \mathbf{val}\ x = e \mid \mathbf{instance}_a \\
\text{Expressions} & e ::= (\,) \mid x \mid \mathbf{let}\ \bar{d}\ \mathbf{in}\ e \mid \lambda x.\, e \mid e\ e \mid \lambda a.\, e \mid e\ \{a = a\} \\
 & \quad \mid\ \mathbf{handle}_a\ e\ \{x, r.\, e;\ x.\, e\} \mid \mathbf{do}_a\ e
\end{array}$$

Figure 2.1: Syntax for the expressions and definitions of the calculus. Sequences of definitions are denoted as $\bar{d}$.

The expression syntax includes the unit value $(\,)$, variables, **let**-expressions, the usual abstraction and application of the λ-calculus, effect instance abstraction, and named application for instances. For simplicity, there is only one operation, **do**, which is always supplied with an instance name and an argument. Effect handlers bind an instance within the body of the handled expression, and provide a clause for the sole operation, where the passed argument x and resumption r are bound, and a return clause.

Let-bindings take a form similar to the syntax of Standard ML [8]: a **let**-expression consists of a sequence of definitions followed by an expression in which the bindings are visible. A definition, represented by d in Figure 2.1, is either the standard **val** $x = e$ form, binding the result of e to x, or the new construct introduced in this work, $\textbf{instance}_a$. This construct creates a new implicit effect instance *scope* for all following definitions in the sequence.

2.2 Typing the Calculus

Before we proceed with providing a type-and-effect system for the source calculus, we have to work out some details regarding effect instances. So far, the syntax of expressions only included effect instance *names*. The programmer-provided names are important: they are used for explicit instance application, and for automatically matching up instance parameters with instances available in the environment, the specifics of which we will define in this section. One problem with instance names is that we lose the ability to define capture-avoiding substitution in types, which is needed for many important language features, such as polymorphism. The issue arises because we are not allowed to just replace bound names with sufficiently fresh ones, as that would break our instance application mechanisms. To regain α-conversion, we decouple these names from the concept of instance *variables*. The syntax will still only mention names, while types and effects will refer to the variables. Finally, instance quantifiers will connect the two entities, binding the variable and associating it with a name. This is reflected in the grammar of Figure 2.2.

$$\begin{array}{rl}
\text{Instance Variables} & i, j, \ldots \\
\text{Types} & \tau ::= \textbf{unit} \mid \tau \to_\varepsilon \tau \mid \forall a{=}i : \sigma .\, \tau \\
\text{Effects} & \varepsilon ::= \iota \mid i \mid \varepsilon \cdot \varepsilon \\
\text{Signatures} & \sigma ::= \tau \Rightarrow \tau
\end{array}$$

Figure 2.2: Grammar for the types, effects and signatures of the calculus.

The types consist of the base **unit** type, the arrow type, with an effect attached, and the instance quantifier. Effects are simply treated like finite sets, with the pure effect ι acting like the empty set, instance variables treated as singletons, and the $\cdot$ operation signifying union. There is only one constructor for effect signatures, $\tau \Rightarrow \tau$.

The well-formedness relations of Figure 2.3 all feature the single context Δ, which contains all the instance variables that may appear free, along with their names and signatures.

Well-formed types. $\boxed{\Delta \vdash \tau :: \text{Type}}$

$$\frac{}{\Delta \vdash \textbf{unit} :: \text{Type}} \qquad \frac{\Delta \vdash \tau_1 :: \text{Type} \quad \Delta \vdash \varepsilon :: \text{Effect} \quad \Delta \vdash \tau_2 :: \text{Type}}{\Delta \vdash \tau_1 \rightarrow_\varepsilon \tau_2 :: \text{Type}}$$

$$\frac{\Delta \vdash \sigma :: \text{Signature} \quad \Delta, a{=}i : \sigma \vdash \tau :: \text{Type}}{\Delta \vdash \forall a{=}i : \sigma . \tau :: \text{Type}}$$

Well-formed effects. $\boxed{\Delta \vdash \varepsilon :: \text{Effect}}$

$$\frac{}{\Delta \vdash \iota :: \text{Effect}} \qquad \frac{a{=}i : \sigma \in \Delta}{\Delta \vdash i :: \text{Effect}} \qquad \frac{\Delta \vdash \varepsilon_1 :: \text{Effect} \quad \Delta \vdash \varepsilon_2 :: \text{Effect}}{\Delta \vdash \varepsilon_1 \cdot \varepsilon_2 :: \text{Effect}}$$

Well-formed signatures. $\boxed{\Delta \vdash \sigma :: \text{Signature}}$

$$\frac{\Delta \vdash \tau_1 :: \text{Type} \quad \Delta \vdash \tau_2 :: \text{Type}}{\Delta \vdash \tau_1 \Rightarrow \tau_2 :: \text{Signature}}$$

Figure 2.3: Well-formedness relations for types, effects and signatures.

The typing relation for expressions is presented in Figure 2.4 with the various well-formedness premises omitted, as they are all quite simple and occur in the expected instance environments. The relation is defined with the usual context for regular variables, Γ, and the instance environment Δ. As opposed to $\in$, as used in the well-formedness relations, we use the alternative $\sqsubseteq$ inclusion operator for Δ in the typing rules. The distinction is based on the shadowing behavior: $\in$ does not care for shadowing, as it is used to "search" the context based on instance variables, for which we can admit Barendregt's convention and only allow distinct instance variables in the context. $\sqsubseteq$ accounts for shadowing of instance names—which we cannot rename—by only allowing the most recently added instance with a given name. One of the premises in the rule for **let** involves an additional relation for definition sequences.

The environment for the relation for definitions, as shown in Figure 2.5, includes an additional stack for implicit instances, Θ, which is of the same form as Δ. It is built up by **instance**, and always starts out empty in the premise of the rule for **let**-expressions.

Typing expressions. $\boxed{\Gamma; \Delta \vdash e : \tau \ / \ \varepsilon}$

$$\frac{}{\Gamma; \Delta \vdash (\,) : \mathbf{unit} \ / \ \iota} \qquad \frac{x : \tau \in \Gamma}{\Gamma; \Delta \vdash x : \tau \ / \ \iota}$$

$$\frac{\Gamma, x : \tau_1; \Delta \vdash e : \tau_2 \ / \ \varepsilon}{\Gamma; \Delta \vdash \lambda x.\, e : \tau_1 \rightarrow_\varepsilon \tau_2 \ / \ \iota} \qquad \frac{\Gamma; \Delta \vdash e_1 : \tau_1 \rightarrow_\varepsilon \tau_2 \ / \ \varepsilon \quad \Gamma; \Delta \vdash e_2 : \tau_1 \ / \ \varepsilon}{\Gamma; \Delta \vdash e_1\ e_2 : \tau_2 \ / \ \varepsilon}$$

$$\frac{\Gamma; \Delta, a{=}i : \sigma \vdash e : \tau \ / \ \iota}{\Gamma; \Delta \vdash \lambda a.\, e : \forall a{=}i : \sigma.\, \tau \ / \ \iota} \qquad \frac{b{=}j : \sigma \sqsubseteq \Delta \quad \Gamma; \Delta \vdash e : \forall a{=}i : \sigma.\, \tau \ / \ \varepsilon}{\Gamma; \Delta \vdash e\ \{a = b\} : \tau\{i \mapsto j\} \ / \ \varepsilon}$$

$$\frac{\Gamma; \Delta; \cdot \vdash \bar{d} \rightsquigarrow \Gamma' \ / \ \varepsilon \quad \Gamma'; \Delta \vdash e : \tau \ / \ \varepsilon}{\Gamma; \Delta \vdash \mathbf{let}\ \bar{d}\ \mathbf{in}\ e : \tau \ / \ \varepsilon}$$

$$\frac{\Gamma; \Delta, a{=}i : \tau_1 \Rightarrow \tau_2 \vdash e : \tau' \ / \ i \cdot \varepsilon \quad \Gamma, x : \tau_1, r : \tau_2 \rightarrow_\varepsilon \tau; \Delta \vdash e_h : \tau \ / \ \varepsilon \quad \Gamma, x : \tau'; \Delta \vdash e_r : \tau \ / \ \varepsilon}{\Gamma; \Delta \vdash \mathbf{handle}_a\ e\ \{x, r.\, e_h;\ x.\, e_r\} : \tau \ / \ \varepsilon}$$

$$\frac{a{=}i : \tau_1 \Rightarrow \tau_2 \sqsubseteq \Delta \quad \Gamma; \Delta \vdash e : \tau_1 \ / \ \varepsilon}{\Gamma; \Delta \vdash \mathbf{do}_a\ e : \tau_2 \ / \ i \cdot \varepsilon}$$

$$\frac{\Gamma; \Delta \vdash e : \tau' \ / \ \varepsilon' \quad \Delta \vdash \tau' <: \tau \quad \Delta \vdash \varepsilon' <: \varepsilon}{\Gamma; \Delta \vdash e : \tau \ / \ \varepsilon}$$

Figure 2.4: The typing relation.

As the rule for **instance** is unusual in its use of the well-formedness predicates, we will examine it in full now.

$$\frac{\Delta, \Theta \vdash \sigma :: \text{Signature} \quad \Gamma; \Delta; \Theta, a{=}i : \sigma \vdash \bar{d} \rightsquigarrow \Gamma' \ / \ \varepsilon}{\Gamma; \Delta; \Theta \vdash \mathbf{instance}_a\ \bar{d} \rightsquigarrow \Gamma' \ / \ \varepsilon}$$

Of note is the fact that the instances allowed to appear in the signature include any prior implicit instances in the sequence, as Δ and Θ are merged in the well-formedness rule. These are not visible in any other situation until bound. The decision to make this exception stems from its usefulness in programming, and from the more intuitive behavior it provides: one way to think about it is that an implicit instance declaration is a promise that the instance will be eventually bound, if needed. Then it seems natural that a promise may refer to the ones made prior, as they all live in the context of potential future instances.

In the case of **val**, there are two rules. The first does not restrict the effects available for the expression, performs no generalization and none of the implicit variables are accessible in e. The second, requiring e to be pure, allows for some of the

Typing definitions. $\boxed{\Gamma; \Delta; \Theta \vdash \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon}$

$$\frac{}{\Gamma; \Delta; \Theta \vdash \varnothing \rightsquigarrow \Gamma \mathrel{/} \varepsilon} \qquad \frac{\Gamma; \Delta; \Theta, a{=}i : \sigma \vdash \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon}{\Gamma; \Delta; \Theta \vdash \mathbf{instance}_a\ \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon}$$

$$\frac{\Gamma; \Delta \vdash e : \tau \mathrel{/} \varepsilon \qquad \Gamma, x : \tau; \Delta; \Theta \vdash \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon}{\Gamma; \Delta; \Theta \vdash \mathbf{val}\ x = e\ \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon}$$

$$\frac{\Gamma; \Delta, \bar{\theta} \vdash e : \tau \mathrel{/} \iota \qquad \Gamma, x : \forall \bar{\theta}.\, \tau; \Delta; \Theta \vdash \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon \qquad \Delta \vdash \bar{\theta} \leq \Theta}{\Gamma; \Delta; \Theta \vdash \mathbf{val}\ x = e\ \bar{d} \rightsquigarrow \Gamma' \mathrel{/} \varepsilon}$$

Figure 2.5: Typing the definitions.

Valid instance subsequences. $\boxed{\Delta \vdash \bar{\theta} \leq \Theta}$

$$\frac{}{\Delta \vdash \varnothing \leq \Theta} \qquad \frac{\Delta \vdash \bar{\theta} \leq \Theta}{\Delta \vdash \bar{\theta} \leq a{=}i : \sigma, \Theta} \qquad \frac{\Delta \vdash \sigma :: \text{Signature} \qquad \Delta, a{=}i : \sigma \vdash \bar{\theta} \leq \Theta}{\Delta \vdash a{=}i : \sigma, \bar{\theta} \leq a{=}i : \sigma, \Theta}$$

Figure 2.6: Dependency-respecting subsequences of implicit instances.

Subtyping. $\boxed{\Delta \vdash \tau_1 <: \tau_2}$

$$\frac{}{\Delta \vdash \tau <: \tau} \qquad \frac{\Delta \vdash \tau_1 <: \tau \qquad \Delta \vdash \tau <: \tau_2}{\Delta \vdash \tau_1 <: \tau_2}$$

$$\frac{\Delta \vdash \tau_1' <: \tau_1 \qquad \Delta \vdash \varepsilon <: \varepsilon' \qquad \Delta \vdash \tau_2 <: \tau_2'}{\Delta \vdash \tau_1 \to_\varepsilon \tau_2 <: \tau_1' \to_{\varepsilon'} \tau_2'}$$

$$\frac{\Delta, a{=}i : \sigma \vdash \tau_1 <: \tau_2}{\Delta \vdash \tau_1 <: \forall a{=}i : \sigma.\, \tau_2} \qquad \frac{a{=}j : \sigma \sqsubseteq \Delta \qquad \Delta \vdash \tau_1\{i \mapsto j\} <: \tau_2}{\Delta \vdash \forall a{=}i : \sigma.\, \tau_1 <: \tau_2}$$

Figure 2.7: The subtyping relation.

variables in Θ to become bound in e, and generalizes its type. When choosing some of the available implicit instances, we need to be careful about instance dependencies which can arise within signatures. We can utilize the well-formedness relation for signatures along with maintaining an appropriate instance context to define a relation that expresses this.

Subtyping is defined to accommodate automatic instantiation and generalization for effect instances. A subeffecting relation is also useful. The subeffecting relation

$\boxed{\Delta \vdash \varepsilon_1 <: \varepsilon_2}$ is essentially set inclusion with the appropriate context-checking premises, and as such is omitted.

The subtyping relation (Figure 2.7) includes the standard rules for reflexivity, transitivity and function types. It also has two rules for the generalization and instantiation of effect instance quantifiers. Note that, as in [3], effect signatures are not subject to subtyping.

Combining the two quantifier rules and named instance application, we obtain a useful feature: the order of instance arguments can now be changed, since subtyping allows us to rearrange instance quantifiers, as in the following example (where $\Delta \equiv b{=}j : \sigma',\ a{=}i : \sigma$).

$$\cfrac{\cfrac{a{=}i : \sigma \sqsubseteq \Delta \qquad \cfrac{b{=}j : \sigma \sqsubseteq \Delta \qquad \cfrac{}{\Delta \vdash \tau <: \tau}}{\Delta \vdash \forall b{=}j : \sigma'.\, \tau <: \tau}}{\cfrac{\Delta \vdash \forall a{=}i : \sigma.\, \forall b{=}j : \sigma'.\, \tau <: \tau}{b{=}j : \sigma' \vdash \forall a{=}i : \sigma.\, \forall b{=}j : \sigma'.\, \tau <: \forall a{=}i : \sigma.\, \tau}}}{\vdash \forall a{=}i : \sigma.\, \forall b{=}j : \sigma'.\, \tau <: \forall b{=}j : \sigma'.\, \forall a{=}i : \sigma.\, \tau}$$

2.3 Translation Semantics

2.3.1 Target Calculus for Translation

Designing direct semantics for the source calculus is not straightforward, as the presence of instance names and implicit instances complicates matters due to their somewhat dynamic nature. Therefore, instead of developing the semantics of the proposed calculus from scratch, it may be preferable to translate its well-typed terms to something more closely resembling other calculi. This also opens up the possibility of benefiting from existing insight and results.

$$\begin{array}{rl}
\text{Variables} & r, x, y, z, \ldots \\
\text{Instance Variables} & i, j, \ldots \\
\text{Definitions} & d ::= \mathbf{val}\ x = e \\
\text{Expressions} & e ::= (\,) \mid x \mid \mathbf{let}\ \bar{d}\ \mathbf{in}\ e \mid \lambda x.\, e \mid e\ e \mid \lambda i.\, e \mid e\ i \\
& \quad \mid\ \mathbf{handle}_{i}\ e\ \{x, r.\, e;\ x.\, e\} \mid \mathbf{do}_{i}\ e
\end{array}$$

Figure 2.8: The target calculus for translation.

The syntax of the target calculus is shown in Figure 2.8, where modifications of the source calculus are shaded. **instance** is no longer in the calculus, and instance application follows a simple positional scheme, like regular function arguments. The dichotomy of instance names and variables is no longer necessary, and all the names are erased.

2.3.2 The Translation

The target calculus, as presented, has no type system[1]. For this reason, the subeffecting derivations do not need to be considered during translation. The translation to the target calculus contains two interesting components:

- instance generalization at **let**-definitions, analogously to the typing rule, and
- instantiation and generalization by adding coercions resulting from the subtyping derivations.

Therefore, we need functions to translate the typing and subtyping derivations of the source calculus.

Translating expressions. The translation function for expressions takes a type derivation tree as its input and returns an expression of the target language. Most of the cases for translating expressions simply recurse into subtrees, and as such, only a few cases are mentioned here to set an example. For the subsumption rule, we apply the expression resulting from translating the subtyping premise to the translation of the expression before subtyping.

$$\left\llbracket \frac{\mathcal{D} :: \Gamma, x : \tau_1; \Delta \vdash e : \tau_2 \;/\; \varepsilon}{\Gamma; \Delta \vdash \lambda x.\, e : \tau_1 \to_\varepsilon \tau_2 \;/\; \iota} \right\rrbracket = \lambda x.\, \llbracket D \rrbracket$$

$$\left\llbracket \frac{\begin{array}{l} \mathcal{D}_1 :: \Gamma; \Delta \vdash e : \tau' \;/\; \varepsilon' \\ \mathcal{D}_2 :: \Delta \vdash \tau' <: \tau \\ \Delta \vdash \varepsilon' <: \varepsilon \end{array}}{\Gamma; \Delta \vdash e : \tau \;/\; \varepsilon} \right\rrbracket = \llbracket \mathcal{D}_2 \rrbracket \; \llbracket \mathcal{D}_1 \rrbracket$$

$$\cdots$$

Translating definitions. The derivations for typing definitions are translated into definitions of the target calculus, with the standard syntactic sugar for a series of

[1]The type system could be easily defined, and like the calculus, would be similar to the one presented by Biernacki et al. [3]. In fact, the implementation uses such a type system internally to catch potential errors made in the earlier phases. However, for the purposes of this exposition of the translation, it was not deemed important enough to warrant its inclusion.

λ-abstractions in the second rule for **val**. Uses of **instance** may be simply erased.

$$[\![\, \Gamma; \Delta; \Theta \vdash \varnothing \rightsquigarrow \Gamma \;/\; \varepsilon \,]\!] = \varnothing$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma; \Delta; \Theta, a{=}i : \sigma \vdash \bar{d} \rightsquigarrow \Gamma' \;/\; \varepsilon}{\Gamma; \Delta; \Theta \vdash \mathbf{instance}_a\ \bar{d} \rightsquigarrow \Gamma' \;/\; \varepsilon} \right]\!\!\right] = [\![\mathcal{D}]\!]$$

$$\left[\!\!\left[\frac{\begin{array}{l}\mathcal{D}_1 :: \Gamma; \Delta \vdash e : \tau \;/\; \varepsilon \\ \mathcal{D}_2 :: \Gamma, x : \tau; \Delta; \Theta \vdash \bar{d} \rightsquigarrow \Gamma' \;/\; \varepsilon\end{array}}{\Gamma; \Delta; \Theta \vdash \mathbf{val}\ x = e\ \bar{d} \rightsquigarrow \Gamma' \;/\; \varepsilon} \right]\!\!\right] = \mathbf{val}\ x = [\![\mathcal{D}_1]\!]\ \ [\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\begin{array}{l}\mathcal{D}_1 :: \Gamma; \Delta, \bar{\theta} \vdash e : \tau \;/\; \iota \\ \mathcal{D}_2 :: \Gamma, x : \forall \bar{\theta}.\, \tau; \Delta; \Theta \vdash \bar{d} \rightsquigarrow \Gamma' \;/\; \varepsilon \\ \Delta \vdash \bar{\theta} \leq \Theta\end{array}}{\Gamma; \Delta; \Theta \vdash \mathbf{val}\ x = e\ \bar{d} \rightsquigarrow \Gamma' \;/\; \varepsilon} \right]\!\!\right] = \mathbf{val}\ x = \lambda i_1, \ldots, i_n.\, [\![\mathcal{D}_1]\!]\ \ [\![\mathcal{D}_2]\!]$$

$$(\text{where } \bar{\theta} \text{ is } a_1{=}i_1 : \sigma_1, \ldots, a_n{=}i_n : \sigma_n)$$

Translating subtyping derivations. The function translating subtyping derivation produces functions of the target calculus which are used for subsumption. The style of this translation is similar to that of Breazu-Tannen et al. [4], but since our target has no types, we only create terms, rather than typing derivations.

$$[\![\, \Delta \vdash \tau <: \tau \,]\!] = \lambda x.\, x$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Delta \vdash \tau_1 <: \tau \qquad \mathcal{D}_2 :: \Delta \vdash \tau <: \tau_2}{\Delta \vdash \tau_1 <: \tau_2} \right]\!\!\right] = \lambda x.\, \mathcal{D}_2\ ([\![\mathcal{D}_1]\!]\ x)$$

$$\left[\!\!\left[\frac{\begin{array}{c}\mathcal{D}_1 :: \Delta \vdash \tau_1' <: \tau_1 \\ \Delta \vdash \varepsilon <: \varepsilon' \\ \mathcal{D}_2 :: \Delta \vdash \tau_2 <: \tau_2'\end{array}}{\Delta \vdash \tau_1 \to_\varepsilon \tau_2 <: \tau_1' \to_{\varepsilon'} \tau_2'} \right]\!\!\right] = \lambda f.\, \lambda x.\, [\![\mathcal{D}_2]\!]\ (f\ ([\![\mathcal{D}_1]\!]\ x))$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Delta, a{=}i : \sigma \vdash \tau_1 <: \tau_2}{\Delta \vdash \tau_1 <: \forall a{=}i : \sigma.\, \tau_2} \right]\!\!\right] = \lambda x.\, \lambda i.\, [\![\mathcal{D}]\!]\ x$$

$$\left[\!\!\left[\frac{a{=}j : \sigma \sqsubseteq \Delta \qquad \mathcal{D} :: \Delta \vdash \tau_1\{i \mapsto j\} <: \tau_2}{\Delta \vdash \forall a{=}i : \sigma.\, \tau_1 <: \tau_2} \right]\!\!\right] = \lambda f.\, [\![\mathcal{D}]\!]\ (f\ j)$$

Chapter 3

Implementation

In this chapter we will discuss the proof of concept programming language that has been implemented as a playground for implicit effect instances.

3.1 Interaction with Type Inference

Programmers, particularly those of functional programming languages, are accustomed to type inference and let-polymorphism. Although polymorphism had no bearing on the details of the source calculus described in the previous chapter, and thus could be elided there, it needs to be considered when discussing the practical implementation of a programming language. Our proof of concept interpreter offers type inference based on first-order unification, with the expected let-polymorphism. It sacrifices completeness of inference for simplicity, though in practice, the types of useful programs can often be inferred.

3.1.1 Instance Variables

The considered calculus involves instance variable binding in types. When dealing with unification, substitution for variables becomes tricky. However, there are a few things we can do to avoid creating too much complexity. The crucial observation is that instance variables can only be substituted for other instance variables. This is reminiscent of the restrictions considered in the area of nominal logic [9]. As it turns out, we can adopt some of the ideas from that field for our purposes. Urban et al. [16] have proposed first-order unification for that setting, and the theory developed therein serves as the basis for our type inference. Instead of substitution, we will consider swapping, or permutation, of variables. In the presence of appropriate freshness constraints, this is equivalent to variable-for-variable substitution, but offers many attractive properties, such as invertibility.

3.1.2 Unification Variables

Since our type system involves instance variables in types, we need to be careful so that those variables do not escape their scope when instantiating a unification variable. Every unification variable is associated with a constraint—the set of instance variables that may occur free within.

When attempting to substitute instance variables within a unification variable, we need some way to suspend this substitution with the particular *occurrence* of the variable. Building on the insight from nominal unification, each occurrence is associated with a permutation of instance variables.

3.1.3 Effect Rows

Though this is not a restriction of the source calculus, the implementation uses rows to represent effects. This is a source of ambiguity when resolving inequalities with unification variables, such as $\langle a \mid ?X \rangle <: \langle a \rangle$, where $\langle a \rangle$ can be read as a closed row containing only a, and $\langle a \mid ?X \rangle$ is an open row with a and the unification variable $?X$. Since we treat our rows as sets rather than multisets, the two solutions are $?X = \langle a \rangle$ and $?X = \langle\rangle$ (the empty row). As this is a largely orthogonal concern, we pay no heed to it and choose the solution that empirically tends to allow useful programs to typecheck. Effect rows have seen much use in other implementations of algebraic effects, and some of them boast complete type inference [6].

3.1.4 Generalization

Generalization of expressions bound within **let** is subject to the purity restriction. Type variables are generalized as in normal let-polymorphism. Effect instance generalization based on the **val** typing rules also occurs at this point, and as the signatures may contain unification variables that are subject to generalization, it has to take place first. The set of instances to generalize is determined by maintaining the instances that are sufficient to type a given expression, which may be smaller than the set of all instances available in context.

3.2 Other Language Features

Apart from type inference and polymorphism, the implementation has been equipped with a variety of other features that enrich the core calculus into a programming language that may already be used to write some larger programs.

Algebraic effect signatures. Compared to the source calculus, which has only one operation, the implementation offers full-fledged algebraic signatures, with polymorphism for the signature and its operations. For example, the following signature for exceptions may be written:

```
signature Exc e = throw : ∀ a. e ⇒ a
```

Algebraic data types. In the same spirit as algebraic signatures, ADTs may also be defined. Moreover, type and signature definitions may be mutually recursive.

Standard library and built-in functions. The prelude defines various basic types and effects, and exposes some built-in functions for integer and string manipulation and standard input and output.

Recursive functions. Function definitions may be declared (mutually) recursive.

Syntactic sugar. Some common syntactic sugar is available, such as function parameters in let-definitions, conditionals, and more.

```
let f x y = x
```

Primitive support for multiple files. Files consist of a preamble, which may contain `include` statements, and a sequence of definitions. Included files are processed as if they were input verbatim.

Chapter 4

Case Study

In this chapter we will work our way through a simplified interpreter for Prolog (without features such as cuts or negation). Various details that have no bearing on the usage of effects or are relatively standard have been omitted, and the full interpreter is available in the `examples/` subdirectory of the provided source code.

4.1 Prolog

Setting up the stage. We need to define some types and effects first. A term is either a variable or a functor with a list of arguments, which are themselves terms. Prolog programs are made up of a series of clauses, each consisting of a conclusion and its goals.

```
type Term = Var String | Fun String (List Term)
type Clause = Cl Term (List Term)
```

We will certainly need to be able to unify terms, which may cause some of the variables to become instantiated with a term. Therefore, we need some form of state. The state can be modeled as an effect with two operations: `set`, which takes a variable name and a term, and `get`, which, given a variable name, produces `Some t` if there is a term associated with the variable, or `None` otherwise. Since we are not restricted in the number of instances of an effect, we can go ahead and define a polymorphic state effect signature, which will be useful later. Since the variable state will be ubiquitous thorough our implementation, we may like to declare an instance as implicit.

```
signature State a = get : () ⇒ a | put : a  ⇒ ()
let update 'st f = put 'st (f (get 'st ()))

instance 'st : State (String → Option Term)
let var_get x = get 'st () x
let var_set x t = update (λs y. if string_eq x y then Some t else s y)
```

When resolving a query, we need to consider every clause in the program, and to backtrack from a unification that fails. The backtracking effect needs to provide nondeterminism, for example with a “coin flip” operation, and a way to cut off the computation. Once again, we will use an implicit instance. The syntax of the language necessitates that the instance names be specified each time an operation is used, but we generally only want one instance of `BT` at a time. We can get around this by defining regular functions which are parametrized with these instances, as their arguments will be provided automatically. Additionally, `choose` picks an element from a list using these two operations.

```
signature BT = flip : () ⇒ Bool | fail : ∀ a. () ⇒ a
instance 'bt : BT
let flip = flip 'bt
let fail = fail 'bt
let rec choose 'bt xs = ...
```

The pattern of defining functions for each operation turned out to be common, and can be tedious with a large number of them, so syntactic sugar is available to automate the creation of these definitions.

```
instance 'bt : BT with defaults
```

Now writing a variety of utility functions is straightforward without ever needing to mention the instances used. Here, `view` is parametrized with `'st`, while `occurs` gets both `'st` and `'bt`.

```
let rec view t =
  case t of
  | Var x →
    case get x of
    | None → t
    | Some t →
      let t = view t in
      set x t; t
    end
  | Fun f ts → t
  end

let rec occurs x t =
  case view t of
  | Var y → if x = y then fail () else ()
  | Fun f ts → iter (λt. occurs x t) ts
  end
```

Using these, `unify` can be defined. To unify the arguments, we may use a higher-order function such as `iter2`. If the lists turn out to have different lengths, `iter2` will signify this by throwing an exception, which is itself just another effect.

```
signature Exc e = throw : ∀ a. e ⇒ a
```

```
let rec iter2 'len_mismatch f xs ys = ...

let rec unify t t' =
  case view t of
  | Var x → ...
  | Fun f ts →
    case view t' of
    | Var x      → occurs x t; var_set x t
    | Fun f' ts' →
      if string_eq f f' then
        handle 'len_mismatch in iter2 unify ts ts' with
        | throw _ → fail ()
        end
      else fail ()
    end
  end
```

Evaluating queries. Now that unification is ready, we need a source of knowledge—the list of rules available—and a way to search for solutions to queries. To increase generality, the database may be yet another effect instance. A “reader” effect would suffice for a read-only database, but using `State` with a different instance will allow us to expand it over time.

```
instance 'db : State (List Clause)
let add_to_db c = update {'st = 'db} (Cons c)
let get_db = get 'db
```

The intended semantics of our interpreter are such that the variables are local to each clause, so each time we retrieve a clause from the database, we need to refresh its variables. We can add a source of fresh identifiers as another instance.

```
signature Fresh = fresh : () ⇒ String
instance 'fresh : Fresh with defaults

let rec eval t =
  let cl = refresh_clause (choose (get_db ())) in
  case cl of
  | Cl t' ts →
    unify t t';
    iter eval ts
  end
```

It is a routine matter of supplying some handlers for `'st`, `'bt` and `'fresh` to resolve a single query. The `h_bt_bool` handler returns a boolean signifying if any successful execution was possible—in order words, whether there is a solution.

```
let query t =
  h_varstate (λ'st ().
  h_bt_bool (λ'bt ().
  h_fresh (λ'fresh (). eval t)))
```

With the addition of a parser, we now have a working REPL that can accept `:add CLAUSE` commands, which modify the database, and run queries.

```
let rec main () =
  handle 'eof in
    let input = list_of_string (read_line ()) in
    case parse_cmd input of
    | Left e  → print_endline (pp_parse_error e)
    | Right r →
      case fst r of
      | Rule cl → add_to_db cl
      | Query t → print_endline (if query t then "Yes." else "No.")
      end
    end;
    main ()
  with
  | throw _ → print_endline "Quitting."
  end

let _ = h_db main
```

Nevertheless, merely outputting whether the query succeeded appears lacking. The resulting variable assignment, or a goal "trace" during execution, may also be useful.

Adding configurability. Providing all that information unconditionally is not ideal, so we will augment the interpreter with some rudimentary runtime configuration. Boolean flags are easily added and propagated; we only need to declare an implicit instance and add a handler surrounding the main loop, and the extra state is added anywhere necessary. In this case, `query` and `main` must be changed to return and process the bindings, and to process a new command, `:set show bindings on|off`.

```
instance 'show_bindings : State Bool

...

let query t =
  ...
  h_bt_opt (λ'bt ().
  eval t;
  if show_bindings () then
    filter_map binding (term_vars t)
  else [])
  ...

...
```

```
let _ = h_db (λ'db _. h_config main)
```

We can proceed in a similar manner to add a toggle for goal traces, but on top of that, we can improve modularity and extensibility further by using a `Writer` instance for tracing output, giving more control over where it ultimately ends up, and the ability to process it further. As an example, here we add tracing to the `eval` function.

```
signature Writer a = tell : a ⇒ ()

instance 'trace : Writer String
let trace = tell 'trace
let trace_enabled () = get 'enable_trace ()

...

let rec eval t =
  trace ("Goal: " ^ pp_term t);
  let cl = refresh_clause (choose (get_db ())) in
  case cl of
  | Cl t' ts →
    unify t t';
    iter eval ts;
    trace ("Success: " ^ pp_term t)
  end

...

let rec main () =
  handle 'trace in
  ...
  with
  | tell s →
    if trace_enabled () then print_endline s else ();
    resume ()
  end
```

4.2 Concluding the Case Study

The example program, while relatively small, used many different effects and separate instances. A pleasant outcome is that even when we decided to go back and make additions to the implementation, we could indeed manage to keep the changes local, as we initially hoped to. Implicit instance declarations mostly occurred at the top-level, and based on the nature of their usage, it is conceivable that in a larger program with modules and multiple files, instances would frequently be declared within the scope of those units. The code reads in a straightforward manner, and explicit instance abstractions were largely avoided, save for the usage of functions implementing generic handlers, such as in the definition of `query`. Even this could be alleviated with the addition of first-class handlers, however. Explicit instance applications are used primarily for renaming instances when using a standard handler for effects like `State`, such as when we defined `add_to_db`. The usage of instances is invisible a lot of the time, especially in the cases where they are simply propagated to the functions that need them, which were a pain point of effect instances. This suggests that this approach may be viable.

Chapter 5

Related Work

5.1 Effect Instances

Over the years many research languages have been implemented with the goal of studying algebraic effects, and the ideas thus developed are making their way into languages with wider industry use as well, notably in the upcoming 5.0 version of OCaml [14]. Yet few of these languages offer a simple mechanism to use multiple instances of an effect, and many are not able to check what effects occur and whether they are handled at compile time.

The Eff language. Version 3.0 of Eff [1] features effect instances that are dynamically created. Unlike our language, instances are treated like normal expressions and can be passed as function arguments, returned from functions, and so on. Any language feature permitting omission of normal program variable binding and passing, such as those already present in some other languages, would have generalized to this notion of effect instance. However, this treatment of instances sacrifices the static guarantees resulting from a type-and-effect system.

Later versions of Eff and OCaml 5.0. Multicore OCaml and modern implementations of Eff have no effect instances of any sort.

Helium. The current implementation of Helium is based on the work of Biernacki et al. [3], and implements lexically scoped effect instances that are tracked statically by its type-and-effect system. The resulting language is reasonably easy to use and enjoys good static guarantees, but unlike the system underlying Eff, its effect instance variables constitute a new entity and can only be used in specific contexts. They can be bound, passed as an argument and used with an operation, but are not considered as expressions in their own right.

Helium already attempts to improve the economy of effect instance mentions by allowing for the instance names to be omitted in an ad-hoc manner. Abstractions and applications can be inferred whenever it is determined that there is only one instance with a particular signature in a given scope. In our implementation, by relying on names rather than signatures, we can easily deal with multiple instances of an effect as well.

5.2 Implicit Binding

On the other axis, it is worthwhile to look at the efforts to reduce the need to explicitly generalize and instantiate variables, even when they are not related to algebraic effects. These may not entirely suit all our needs, given that we treat instance variables as a special entity, but they still prove to be a valuable source of inspiration and context.

Coq sections and implicit arguments. One such mechanism, and the initial inspiration for this work, is the *section* feature of Coq [15] used along with the `Variable` and `Context` keywords. An example usage of this is showcased in Listing 5.1.

```
Section a.
  Variables n m : nat.
  Definition f := n + m.
  Check f : nat.

  Variable (A : Type) (x : A).
  Definition id := x.
  Check id : A.
End a.

Check f : nat → nat → nat.
Check id : ∀ A : Type, A → A.
```

(a)

```
Section b.
  Context {A : Type}.
  Variable x : A.
  Definition id := x.
  Check id : A.
End b.

Check id 0 : nat.
Check id (A := nat) : nat → nat.
```

(b)

Listing 5.1: Coq sections in action.

Within the section of Listing 5.1a, the types of `f` and `id` are respectively `nat` and `A`. After the `End s` command, these types become `nat → nat → nat` and `∀ A : Type, A → A`, as the variables cease being visible and definitions from the section are generalized. Furthermore, arguments can often be inferred by combining sections with *implicit arguments*. The `Context` command can be used to declare them implicit, as in Listing 5.1b. Coq's implicit argument inference relies on types rather than names, including information that can be obtained from the dependent type of a function and the surrounding context. However, they can be passed by name explicitly using the `f (name := expr)` syntax. Since the section feature is a part of the command language, it cannot be freely mixed with expressions, though the sections may be nested.

Haskell's `ImplicitParams`. Haskell offers a language extension that adds *implicit parameters* [7], which share some similarities with the facilities we desire. Those special parameters, like Haskell's typeclasses, are tracked using *constraints*, and they are propagated automatically when performing function calls. To bind an implicit parameter, a binding form such as `let` needs to be used, functioning as a way of passing them explicitly. Implicit parameters are distinguished from normal variables by prepending `?`. Unlike our solution, the parameters in Haskell need not be declared beforehand. Another difference is that such parametrized functions are not considered first-class entities, and must always be instantiated at point of use. As a result, they cannot be passed to another function without instantiating the implicit parameters at the call-site.

Chapter 6

Future Work and Conclusion

Though the proposed approach seems promising so far, it would be beneficial to try to integrate it into a more feature-rich language to see how it can interact with other functionality. A natural application of this is the usage of implicit instances in tandem with ML-like modules, which could superficially resemble sections in Coq [15] even more than what is available in our proof of concept. An obvious target for this is to modify the front-end of Helium [3], as the translation semantics would allow for much of the implementation to be reused.

Another point of interest from the perspective of a language implementer is to evaluate the possible enhancements of type inference. As mentioned in Section 3.1, the current implementation takes an approach that emphasizes simplicity, while trading off completeness. Some notable directions to consider include the more refined treatment of inference with subtyping proposed by Pottier [12] or an implementation of effects that is not restricted to row types.

As for the theoretical foundations, though apparently nontrivial, designing direct semantics for the calculus could provide new insight into the intuitions that can be relied on while programming. If such semantics could be related to the translation semantics, it could provide a programmer with additional tools for reasoning about programs while still reaping the benefits of a translation-based implementation.

6.1 Conclusion

While lexically scoped effect instances such as those implemented in Helium [3] provide a way to use multiple instances of an effect in a controlled and statically safe manner, explicit instance abstraction and application can be tedious and inconvenient to manage for the programmer. We suggest a possible systematic solution that makes it possible to declare an instance that may be implicitly bound by the definitions in its scope, and to perform automatic passing of instance arguments. To support our belief that this solution may be practical, apart from describing the core calculus, an experimental implementation of a programming language has been developed. Preliminary evidence gathered while using this implementations to write effectful programs allows us to believe that this approach is a direction worth exploring in the design space of languages of algebraic effects.

Bibliography

[1] A. Bauer and M. Pretnar. “Programming with algebraic effects and handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001.

[2] D. Biernacki et al. “Abstracting Algebraic Effects”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290319.

[3] D. Biernacki et al. “Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371116.

[4] V. Breazu-Tannen et al. “Inheritance as implicit coercion”. In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 172–221. DOI: 10.1016/0890-5401(91)90055-7.

[5] L. Convent et al. “Doo bee doo bee doo”. In: *Journal of Functional Programming* 30 (2020), e9. DOI: 10.1017/S0956796820000039.

[6] D. Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. DOI: 10.1145/3009837.3009872.

[7] J. R. Lewis et al. “Implicit Parameters: Dynamic Scoping with Static Types”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’00. Boston, MA, USA: Association for Computing Machinery, 2000, pp. 108–118. DOI: 10.1145/325694.325708.

[8] R. Milner et al. *The Definition of Standard ML*. The MIT Press, May 1997. DOI: 10.7551/mitpress/2319.001.0001.

[9] A. M. Pitts. “Nominal logic, a first order theory of names and binding”. In: *Information and Computation* 186.2 (2003). Theoretical Aspects of Computer Software (TACS 2001), pp. 165–193. DOI: 10.1016/S0890-5401(03)00138-X.

[10] G. Plotkin and J. Power. “Computational Effects and Operations: An Overview”. In: *Electronic Notes in Theoretical Computer Science* 73 (2004). Proceedings of the Workshop on Domains VI, pp. 149–163. DOI: 10.1016/j.entcs.2004.08.008.

[11] G. D. Plotkin and M. Pretnar. “Handling Algebraic Effects”. In: *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). DOI: 10.2168/LMCS-9(4:23)2013.

[12] F. Pottier. *Type Inference in the Presence of Subtyping: from Theory to Practice*. Research Report RR-3483. INRIA, 1998. URL: https://hal.inria.fr/inria-00073205.

[13] M. Pretnar. “An Introduction to Algebraic Effects and Handlers. Invited tutorial paper”. In: *Electronic Notes in Theoretical Computer Science* 319 (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)., pp. 19–35. DOI: 10.1016/j.entcs.2015.12.003.

[14] K. Sivaramakrishnan et al. “Retrofitting Effect Handlers onto OCaml”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 206–221. DOI: 10.1145/3453483.3454039.

[15] The Coq Development Team. *The Coq Proof Assistant*. Version 8.15. Jan. 2022. DOI: 10.5281/zenodo.5846982.

[16] C. Urban, A. Pitts, and M. Gabbay. “Nominal Unification”. In: *Computer Science Logic*. Ed. by M. Baaz and J. A. Makowsky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 513–527. DOI: 10.1007/978-3-540-45220-1_41.