Modular Implicits internship report

7th October 2023

This 8-week project was done at the University of Cambridge Department of Computer Science and Technology in August-September 2023 by Patrick Reader and Daniel Vlasits, and was supervised by Jeremy Yallop.

Contents

Introduction

Modular Implicits is an extension to the OCaml language which enables ad-hoc polymorphism. By extending the OCaml module system, it allows you to write code that operates on a constrained set of types. The feature is roughly analogous to type classes in Haskell, traits in Rust, or interfaces in Java. This example illustrates what is possible:

```
module type Showable = sig
  type t
  val show : t -> string
```

```
end

implicit module Showable_Int = struct
  type t = int
  let show x = string_of_int x
end

implicit module Showable_List {S: Showable} = struct
  type t = S.t list
  let show xs = string_of_list S.show xs
end

let show {S: Showable} x = S.show x

let () =
  print_endline (show 5);
  print_endline (show [1; 2; 3])
```

Further details are given in the paper by Leo White, Frédéric Bour, and Jeremy Yallop in 2015, which first introduced modular implicits.

We worked on a variety of small projects. Mostly they involved taking advanced functional programming concepts from other languages (mainly Haskell), and reimplementing them in OCaml with modular implicits. We also added modular implicits to some existing OCaml code, to make it easier to use.

All the code we wrote is available to view on GitHub, under the modular-implicits organisation. They are also all packaged into our opam repository.

Along the way we were able to find and document some of the limitations in the design and implementation of modular implicits. We hope future developers can use this knowledge to make modular implicits better, so that it can potentially eventually be included in the main OCaml language.

Projects

imp

"imp" is a library which was originally mostly written by Leo White. It contained definitions of some fundamental functional programming abstractions. This includes interfaces for functors, monads, monoids, and basic operations like comparisons and converting objects to strings.

It seems like Leo developed this library just for his own experimentation. We thought it could be more like a standard library for OCaml with modular implicits, so we cleaned up the code, and added some more useful concepts to it.

One of these concepts was monad transformers, with implementations of the reader and state monads. Once these were implemented we didn't bother with any others, because we knew they'd be possible. We might come back to this and add some more in future.

We found a limitation while trying to define the MonadTrans typeclass. The root cause of the problem is one of the fundamental differences between OCaml's module system and the typeclass system in Haskell. Most of the time, modules can be used to model typeclasses, but differences do exist. In OCaml, each instantiation of a module is considered a different "thing", so `X {Y}` is not the same as `X {Y}` if they were created in different places. This contrasts to Haskell, where if types `X` and `Y` are both the same, then `X Y` is always the same in all contexts.

Lenses
"lens" is a Haskell library mainly written by Edward Kmett. He gave a good introduction to the concepts it uses in his talk Lenses, Folds, and Traversals. We created a port of some of the core functionality to OCaml.

In Haskell, the lens library requires a lot of language extensions. However, almost all its functionality could be implemented in OCaml using just modular implicits. The biggest problem we encountered was the lack of module subtyping. More details are given in this GitHub issue.

Categories

The categories library is drawn from the data-category haskell library. Each category contains a GADT with two parameters representing the morphisms. This works very nicely within a category and morphism compositions can be cleanly expressed, with the type checker verifying all compositions have been handled. At the value level objects are represented by their identity morphisms.

The module for Functors is coded, however it is sadly not very useful as there is no satisfactory way to map types in OCaml and therefore the types of Functors you can implement are limited.

Arrows

The arrows library is drawn from the Control.Arrows library in Haskell, is fully implemented and works well. The last step would be adding arrow do notation, but this would require adding major syntax to the compiler.

Data structures

This is the beginnings of a library to create data structures which operate implicitly, and don't need to be defined in advance. Currently Set and Map from the OCaml standard library have been implemented.

Automatic differentiation

The auto differentiation library is drawn mostly from the paper Beautiful Differentiation. The library allows you to construct functions operating on numbers, such as

```
let add {F : Floating} x y = x + y
```

and then differentiate them automatically. You can then pass the `d` data type to this function and it will track the derivatives for you, as a forward pass.

Backporting let binding syntax

OCaml 4.08 introduced let-binding operators, which make it much more convenient to write code that uses monads. However, the Modular Implicits fork is based on OCaml 4.03, so it lacks that feature. We decided it would be very useful to have binding operators available for our work, so we backported it. The backport was only a "minimal viable product", just making the changes compatible with the older codebase. We didn't add proper integration between modular implicits and let syntax, so the backport doesn't have support for binding operators which are defined polymorphically with modular implicits.

Generics

The generics library provides data types for representing any (non-recursive) algebraic data type. This was based on the GHC.Generics library in Haskell. Combined with modular implicits, it allows you to construct functions which work on any data type with a valid generic representation. The implementation of the generic interface could easily be derived by the compiler in future. When that is done, it would allow derivation of arbitrary type classes using pure OCaml, without needing to modify the compiler further.

Staged generics

BER MetaOCaml is another extension of the language, which adds support for staged programming. This is a system similar to macros in Lisp, which allows code to be generated using information that is not known until a later *stage* of program execution.

The standard generics approach is useful, but suffers from poor performance because all values must be converted to the "representation" type before you can do any generic operations on them. Staging can help with this, by generating code which can be specialised for each type, using a collection of type classes for each building block of algebraic types. For example, the Sum typeclass has operations for constructing values of each variant, and for pattern-matching. This is similar in aim, though not in implementation, to Jeremy Yallop's paper Staged Generic Programming.

compact

We created the library "compact" as an example use-case for staged generics. It encodes and decodes values of arbitrary types into opaque bunches of bits. (It currently only supports finite non-recursive types, though.) This could be useful for implementing binary trie data structures for non-integer user-defined types, for example for memoisation.

Compact is a bit like the serde library in Rust (which also provides generic serialisation and deserialisation for arbitrary types). Serde uses procedural macros to derive implementations for user-defined data types, but compact uses the staged generics interface, which is more declarative. Unlike serde, which requires just one line, you still have to manually write instances of Sum or Product to use compact. However, these instances could then be used for all kinds of generic code. In future, the compiler could automatically derive these generic instances, like how the Haskell compiler can derive Generic.

Staged tries

The staged tries library allows you to implement a mapping data structure with arbitrary user-defined data types as the keys, using generics to derive the implementation. This also uses staged generics.

We ran into another problem: the value restriction does not consider quoted expressions to be values, so they cannot be given polymorphic types. This meant we had to add some annoying redundant parameters of type `unit` to get the derived instances to work.

QuickCheck

This library is based on the Haskell quickcheck library. Implicits worked especially well for this library, it allows anything with the Testable property to be run on many test cases and analysed.

SearchM

searchM contains a highly general monadic Depth First Search implementation, similar to the search-algorithms library for Haskell. The code is hard to read as we tried to work around a compiler bug.

Limitations

Here we explain in more detail some of the other limitations that we found. These would be good starting points for further work on the design and implementation of modular implicits.

"Any" parameterisation

This is easiest to understand with a code example. Here is a definition of a monoid:

```
module type Monoid = sig
    type t
    val empty : t
    val append : t -> t -> t
end
```

Now we would like to be able to write something like this (using hypothetical syntax):

```
implicit module List (type a) : Monoid with type t = a list
= struct
    type t = a list
    let empty = []
    let append = (@)
end
```

But modules can only be parameterised by modules, not types directly, so we have to declare a wrapper module type (called "Any") to hold the type:

```
module type Any = sig
    type t
end

implicit module List {A: Any} : Monoid with type t = A.t list
= (* ... *)
```

This also means we have to declare an implicit instance of Any for all types we might want to use. Alternatively, instead of having a syntax for passing types directly to modules, the compiler could be extended to automatically derive an instance of `Any` for all types. (In fact, the compiler could possibly automatically derive instances of any module type that consists only of type definitions)

Compiler problems
The modular implicits compiler is not perfect, and we ran into several bugs and crashes which we had to work around. They are all listed as issues on the GitHub repository for modular implicits.

In addition, when using complicated systems of modules, the compiler became very slow. This was especially noticeable with lenses and staged generics. We would like to know if the compiler can be sped up for these cases, or if we could avoid the slowness by designing our module types more carefully.

Also, we often had difficulty using dune and other tools in the OCaml ecosystem, because the modular implicits compiler is based on a OCaml version 4.03, which is very old. It would be much easier to use if it were rebased on a modern version. We also wouldn't have had to backport the let-binding syntax, for example.

Structural typing of modules
In OCaml, modules are typed structurally, rather than nominally. This can cause problems with ambiguous instances in cases where the module signature is very simple, as it might be fitted by several instances by coincidence. For example, because `Any` has only `type t`, any other instance containing a type called t will fit the pattern of Any, which could cause ambiguities in resolving the modular implicit. To avoid this, we added a special marker value called `__any__` to Any, which is never used, but disambiguates those cases. This is called the "blessing signatures" pattern, which was first described in this talk by Jacques Garrigue and Frédéric Bour.

Bidirectional type inference in instance declarations
In Haskell, you can provide a more general type than is required for an instance method, and it will be automatically narrowed to fit the method's type as defined in the class. With modular implicits in OCaml though, you can't do that - the type of a method is only checked *after* it's defined, rather than being used to inform its inferred type.

Possible backwards compatibility issues due to having to name implicit modules
Unlike in Rust or Haskell, instances (implicit modules) have to be given names - and specific instances can be referred to explicitly using their names. This creates potential backwards compatibility issues, if a particular instance's derivation changes. In most languages, with type classes, this kind of change would be

unobservable to code that just uses the instance, but in OCaml with modular implicits, user code might have named that instance explicitly, so changing its derivation would break that code.

This is not necessarily a big problem, but it's still something to be aware of.

Module subtyping
See the section on lenses

Module identity
See the section on monad transformers in the imp library

Value restriction on quoted expressions
See the section on staged tries

Evaluation

We both thoroughly enjoyed this internship. It was nice to have the freedom to work on broad problems in our own time, and put our technical and academic skills into practice. We have learnt a lot about how software engineering projects are really conducted, and practised our OCaml and functional programming.

Acknowledgements

We are grateful to the OCaml Software Foundation for funding this project, to Jeremy Yallop for supervising and guiding us throughout, and to the University of Cambridge Department of Computer Science and Technology for organising the internship.