NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS, ST. PETERSBURG

DEPARTMENT OF COMPUTER SCIENCE

Alexey Trilis

Retrofitting Implicit Modules for 1ML

RESEARCH PROPOSAL

Scientific supervisor:
Daniil Berezun
PhD in Physico-Mathematical Sciences

ST. PETERSBURG
2021

Contents

Ad hoc polymorphism proved to be a highly desirable feature for any language. Lack of this feature for languages in the ML family can be considered a significant problem. In this work, we present the solution to this problem within the framework of experimental language 1ML. This language offers a minimal and uniform look on ML modules, uniting them with the core language. We extend 1ML with implicit modules, offering Scala-like ad hoc polymorphism. We compare our approach with a recent proposal of modular implicits for OCaml and claim to improve type checking completeness, using minimality and certain design choices of 1ML.

Keywords: implicit parameters, implicit modules, ML modules, type classes, ad hoc polymorphism, 1ML.

1 Introduction

Background. *Ad hoc polymorphism* is a programming language feature that allows functions with the same name to have different semantics depending on argument types. Simple but important examples include using the operator + to add up both integers and floating-point numbers and using the function `print` to print values of different types.

Given its apparent usability, ad hoc polymorphism is implemented in many languages. Two of the most notable general ideas about designing it are *constrained polymorphism* and *implicit parameters*. Constrained polymorphism makes it possible to add constraints on argument behavior to polymorphic functions. Languages where constrained polymorphism can be found include Haskell and various object-oriented languages with bounded generics, such as Java or C#. Implicit parameters are the technique of inferring some function parameters based on the context of their usage. This approach was popularized mainly by Scala.

ML can be characterized as the language with powerful type inference and an advanced module system. Along with its dialects, it enjoys fair popularity in certain domains. Aside from use in academia, one of the most popular ML dialects, OCaml, found extensive industry use in such domains as compiler development, static program analysis, automatic theorem proving, financial systems, and web development.

Despite all the positive aspects of ML, it does not currently support ad hoc polymorphism. Consequently, OCaml standard library contains separate adding operators: `+` for integers and `+.` for floating-point numbers, and multiple functions for printing, such as `print_int` or `print_string`. Coding in OCaml requires the programmer to choose between these options explicitly. Lack of polymorphic `print` function makes printing polymorphic parameters impossible. Overall, this is undesirable verbosity for ML, given that one of its main strengths is parametric polymorphism with the optionality of most explicit type annotations.

Another common point of criticism of ML is its separation into two languages: core language and module language. Historically, core language was designed first, and more powerful module language was designed later. As a result, modules in ML are not truly first-class citizens. 1ML (Rossberg, 2015) is a proposed redesign of ML, which acts as a solution to this problem: in 1ML, core and module language are unified.

Problem statement. Our main goal is to design and implement implicit modules for the 1ML language. We choose 1ML because, on the one hand, it is more minimal and uniform than traditional ML dialects, and on the other hand, it is more expressive and concise (Rossberg, 2015). Treatment of modules in 1ML makes several parts of designing implicit modules easier, so we plan to improve some results of the previous proposal for OCaml (White et al., 2015). Special attention should be made to improve the completeness of the implicit search and module unification algorithm, which are weak points of the previous proposal. To evaluate our result, we compare it with the previous proposal using a test suite consisting of various implicit module use cases.

Structure of the paper. The paper is structured as follows:

- section 2 explains the motivation behind the creation of 1ML and gives an overview of approaches to ad hoc polymorphism in different languages, with a focus on the ML family;
- section 3 describes our approach to design implicit modules;
- section 4 presents the anticipated results of this research.

2 Literature Review

ML modules and 1ML. ML module system was designed using dependent type machinery (MacQueen, 1986). While powerful, it is pretty verbose and sometimes difficult to integrate with core language. The second-class status of modules in ML has some practical effects. For example, it results in an inability to express something as trivial as dynamically choosing a module. Several works have been made to address this practical problem, most notably *packaged modules*, described first by Russo (2000) and then implemented for most ML dialects, including OCaml.

Recent studies demonstrated that dependent types are unnecessary, because the entire ML module system can be expressed in System F_ω (Rossberg et al., 2010). As a continuation of these studies and as a solution to known issues of ML modules, language 1ML (Rossberg, 2015) was created. In this language, each language construction can be viewed as a module and elaborated into System F_ω, eliminating both the need for dependent type theory and the module-core stratification. A prototype interpreter is provided by the author, which will serve as the basis for this work's practical part.

Ad hoc polymorphism. The classic formalized approach to ad hoc polymorphism is based on *type classes*. They were introduced in Haskell (Wadler & Blott, 1989) and then replicated in several other languages, such as Agda (Devriese & Piessens, 2011) and Rust (Matsakis & Klock, 2014).

Type classes require *canonicity*: each type class cannot have multiple instances for one type. Consequently, type classes are not a possible choice for all languages. White et al. (2015) noted that in OCaml, we could not check canonicity in the general case because canonicity violations can be hidden behind modular abstractions. The same reasoning can be applied to 1ML as well. This makes type classes in the classical sense not a viable choice for our problem.

Implicit parameters and *implicit conversions*, collectively referred to as *implicits*, were introduced in Scala (Oliveira et al., 2010) as a lightweight replacement for type classes in an object-oriented language. Unlike type classes, which can be inferred from function body, implicits need to be declared in a function definition. On the other side, implicits do not require canonicity, relying on the weaker property of *non-ambiguity*.

Approaches for ML/OCaml. Dreyer et al. (2007) described *modular type classes*, offering type class functionality in ML. However, to make them work without canonicity, the authors impose rather strong restrictions.

Another attempt to design implicits in OCaml was made by White et al. (2015) and serves as the main inspiration for this work. In general, the authors follow the Scala approach. Their proposal only covers implicit parameters, not implicit conversions, arguing that the latter unnecessarily complicates reasoning about code. Another difference is ambiguity handling: in Scala, implicits are resolved using some precedence rules, and ambiguity occurs only in the presence of implicit candidates with the same precedence, while in OCaml proposal having several matching implicit candidates in scope leads to ambiguity error.

Although White et al. give an informal description and even present a minimal prototype, their work is far from complete, and OCaml (as of version 4.12) still lacks implicits. Of course, one of the reasons for this delay is the grand scope of this project, as designing and implementing this feature in an already mature language like OCaml requires extensive work, both theoretical and practical. However, the authors point out some particular difficulties they encounter.

Firstly, implementing implicit modules requires unification at the module level, which OCaml lacks. Implementing this unification poses a significant challenge. In 1ML, this part is much easier because of its non-dependent typing and first-class module status.

Secondly, the order of resolving implicit modules is important: resolving them in incorrect order can lead to ambiguity errors. To achieve predictability of type checking in prototype, authors give some weak guarantees on resolve order and fail if compilation succeeded only because of an ordering not ensured by these guarantees. They argue that it is common practice in OCaml for other order-dependent features. We, however, want to explore solutions that improve the completeness of implicit search.

3 Methodology

Implicit resolving and type inference are closely related in that one cannot be done before the other (White et al., 2015). This restriction determines the phase in which resolving of implicits must be done: simultaneously with type inference.

We process implicits in their order in the syntax tree. When our algorithm encounters an application involving implicit parameters, these parameters are resolved by searching for all declarations marked as implicit and trying to match them. We obtain the list of possible candidates as the result of this procedure. If it contains only one expression, this expression is inserted as a resolved implicit argument. Otherwise, the error is reported.

We use subtyping relation to determine if a declaration matches. When trying to match a declaration that has implicit parameters, our algorithm resolves these parameters recursively.

The described solution is already designed and implemented. However, it does not address the ordering problem: as we traverse the syntax tree in a predetermined order, we can report ambiguity in code that would be correctly resolved with a different ordering.

To solve this problem, we extend our algorithm with the following approach. If the algorithm cannot resolve implicit due to ambiguity, it postpones it until more information is available. More precisely, a placeholder variable is inserted in the place of this implicit. We maintain the context of these variables along with types: for each variable, we store the most specific known type. Processing other expressions may reveal more specific types for placeholder variables. Every time this specification happens, we rerun the search algorithm and replace the placeholder with actual expressions if successful. We only report ambiguity if there are still unresolved variables at the end of the algorithm.

4 Anticipated Results

We designed an approach to extend 1ML with implicit modules, implemented the core part of this approach, and plan to implement it fully.

One of our goals is to evaluate our approach by comparing the expressiveness of our solution with the solution of White et al. To do so, we plan to collect the test suite, consisting of various 1ML sources using implicits along with corresponding OCaml sources. We will base this test suite on examples from the paper by White et al. and tests from their prototype. Also, we will use examples from the literature regarding Scala and implementations of traditional functional data structures, such as monads. Note that translation of these tests poses a particular challenge as some of the language features, such as ADTs and some recursion forms, are absent from 1ML and may not be trivially expressable.

In the best-case scenario, our solution will work correctly on all tests where the OCaml solution works, excluding cases in which translation of tests to 1ML is impossible without a significant amount of work. Additionally, we can reasonably expect that our approach will work on some tests that OCaml does not support, such as ordering-dependent cases. Our implementation also supports *implicit functors*, which the authors of the OCaml solution mentioned as a possible area of future work.

Conclusion

While extending languages in the ML family with ad hoc polymorphism is highly desirable, it remains an open problem. We take a step further in developing a solution for this problem by designing implicit modules for the 1ML language. We also have implemented this solution as the expansion of the prototype 1ML interpreter.

We offer the solution that is more complete than solutions for OCaml. Because of the nature of 1ML, our solution also does not require unification at the module level, which prevents integrating the OCaml solution into the main language. From another point of view, our work contributes to researching the capabilities of 1ML and improving its features. Besides implicit modules, we also extend 1ML with several minor language features as a side effect of the test and comparison process.

Since 1ML can be viewed as a user-friendly syntax for System F_ω, our work can also lead to expanding System F_ω with term inference, which can be interesting as this system serves as the theoretical foundation for several programming languages. This problem is much more complicated than this work's scope, but our work can serve as a first step to solve it.

References

Devriese, D., & Piessens, F. (2011). On the bright side of type classes: Instance arguments in Agda. In *Proceedings of the 16th acm sigplan international conference on functional programming* (p. 143–155). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2034773.2034796` doi: 10.1145/2034773.2034796

Dreyer, D., Harper, R., Chakravarty, M. M. T., & Keller, G. (2007). Modular type classes. In *Proceedings of the 34th annual acm sigplan-sigact symposium on principles of programming languages* (p. 63–70). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1190216.1190229` doi: 10.1145/1190216.1190229

MacQueen, D. B. (1986). Using dependent types to express modular structure. In *Proceedings of the 13th acm sigact-sigplan symposium on principles of programming languages* (p. 277–286). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/512644.512670` doi: 10.1145/512644.512670

Matsakis, N. D., & Klock, F. S. (2014). The Rust language. In *Proceedings of the 2014 acm sigada annual conference on high integrity language technology* (p. 103–104). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2663171.2663188` doi: 10.1145/2663171.2663188

Oliveira, B. C., Moors, A., & Odersky, M. (2010). Type classes as objects and implicits. In *Proceedings of the acm international conference on object oriented programming systems languages and applications* (p. 341–360). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1869459.1869489` doi: 10.1145/1869459.1869489

Rossberg, A. (2015). 1ML – core and modules united (F-ing first-class modules). In *Proceedings of the 20th acm sigplan international conference on functional programming*

(p. 35–47). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2784731.2784738` doi: 10.1145/2784731.2784738

Rossberg, A., Russo, C. V., & Dreyer, D. (2010). F-ing modules. In *Proceedings of the 5th acm sigplan workshop on types in language design and implementation* (p. 89–102). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1708016.1708028` doi: 10.1145/1708016.1708028

Russo, C. V. (2000). First-class structures for Standard ML. In G. Smolka (Ed.), *Programming languages and systems* (pp. 336–350). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from `https://doi.org/10.1007/3-540-46425-5_22` doi: 10.1007/3-540-46425-5_22

Wadler, P., & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th acm sigplan-sigact symposium on principles of programming languages* (p. 60–76). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/75277.75283` doi: 10.1145/75277.75283

White, L., Bour, F., & Yallop, J. (2015, Dec). Modular implicits. *Electronic Proceedings in Theoretical Computer Science*, *198*, 22–63. Retrieved from `http://dx.doi.org/10.4204/EPTCS.198.2` doi: 10.4204/eptcs.198.2