A Modal Analysis of Metaprogramming, Revisited

Brigitte Pientka
McGill University, Montreal, QC, Canada
`http://www.cs.mcgill.ca/~bpientka`
bpientka@cs.mcgill.ca

Abstract

Metaprogramming is the art of writing programs that produce or manipulate other programs. This opens the possibility to eliminate boilerplate code and exploit domain-specific knowledge to build high-performance programs. Unfortunately, designing language extensions to support type-safe multi-staged metaprogramming remains very challenging.

In this talk, we outline a modal type-theoretic foundation for multi-staged metaprogramming which supports the generation and the analysis of polymorphic code. It has two main ingredients: first, we exploit contextual modal types to describe open code together with the context in which it is meaningful; second, we model code as a higher-order abstract syntax (HOAS) tree within a context. These two ideas provide the appropriate abstractions for both generating and pattern matching on open code without committing to a concrete representation of variable binding and contexts.

Our work is a first step towards building a general type-theoretic foundation for multi-staged metaprogramming which on the one hand enforces strong type guarantees and on the other hand makes it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing reliability of and trust in the code we are producing and running.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Type structures; Theory of computation → Lambda calculus; Theory of computation → Modal and temporal logics

Keywords and phrases Type systems, Metaprogramming, Modal Type System

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.2

Category Invited Talk

Funding *Brigitte Pientka*: This work has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Acknowledgements I would like to thank my collaborators: Wilson Cheang, Samuel Gélineau, Junyoung Jang, and Stefan Monnier.

1 Summary

Metaprogramming provides programmers with the ability to write programs that generate specialized and optimized code. This makes it possible to design and implement domain-specific program optimizations that complement general compiler optimizations yielding substantial performance gains. Unfortunately, designing language extensions to support writing type-safe meta-programs remains very challenging.

One widely used approach to metaprogramming going back to Lisp/Scheme is using quasiquotation which allows programmers to generate and compose code fragments. For example, the quasiquotation ⌈`2 + 2`⌉ is representing an abstract syntax tree (AST) of the expression `2 + 2`. We can embed and compose code fragments using unquote, written as ⌊ ⌋. Assuming that the function `square 2` generates code ⌈`2 * 2`⌉, the expression ⌈`2 +` ⌊`square 2`⌋⌉

evaluates to the code ⌈`2 + 2 * 2`⌉ where we splice in the generated code. There are two immediate questions that arise:

1. Can we express and reason statically about open code, i.e. code that may contain and refer to variables?
2. Can we analyze and further manipulate code via pattern matching?

A milestone in developing a logical foundation for characterizing code and reasoning about code generation is the work by Davies and Pfenning [2]. Davies and Pfenning distinguish between code and programs using the necessity modality. For example, the code ⌈`2 + 2`⌉ has the modal type ⌈`int`⌉, while the program `square` has type `int` $\rightarrow$ ⌈`int`⌉. This allows us to statically reason about different stages of computation. However, Pfenning and Davies' work has two limitations: first, it only allows us to generate and reason about closed code and second, it does not support analysis of code via pattern matching. Subsequent work by Nanevski, Pfenning and Pientka [3] suggests to characterize open code `2 + x` together with the context `x:int`, ascribing the code ⌈`x. 2 + x`⌉ the contextual type ⌈`x:int` $\vdash$ `int`⌉ thereby removing the first restriction. Yet, a type-safe multi-staged metaprogramming foundation that supports both the generation of and pattern matching on open code remains elusive.

In this talk, we outline a modal type-theoretic foundation for polymorphic multi-staged metaprogramming that brings together the generation and the analysis of open code within the same framework. In particular, we draw on the theory and practice of contextual types and first-class contexts in the Beluga proof and programming environment [4, 7, 8, 1, 6, 5] and adapt two main ideas to the metaprogramming setting: first, we exploit contextual modal types to describe open polymorphic code together with the context in which it is meaningful; second, we model code as a higher-order abstract syntax (HOAS) tree within a context. These two ideas provide the appropriate abstractions for both generating and pattern matching on open code without committing to a concrete representation of variable binding and contexts which is left open to the implementor of the language.

Our work is a first step towards building a general type-theoretic foundation for multi-staged metaprogramming which enforces strong type guarantees and whose provided abstractions make it easy to generate and manipulate code. This will allow us to exploit the full potential of metaprogramming without sacrificing reliability of and trust in the code we are producing and running.

References

1 Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

2 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001. doi:10.1145/382780.382785.

3 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

4 Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

5 Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rebecca Zucchini. A type theory for defining logics and proofs. In *34th IEEE/ ACM Symposium on Logic in Computer Science (LICS'19)*, pages 1–13. IEEE Computer Society, 2019.

6 Brigitte Pientka and Andrew Cave. Inductive Beluga:Programming Proofs (System Description). In *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer, 2015.

7 Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, 2008.

8 Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer, 2010.