Types and Semantics of Extensible Data Types

Cas van der Rest[1] and Casper Bach Poulsen[1]

Technische Universiteit Delft, Delft, The Netherlands

A common litmus test for a language's capability for modularity is whether the programmer is able to both extend existing data with new ways to construct it and add new functionality for this data. All in a way that preserves static type safety; a conundrum which Wadler [14] dubbed the *expression problem*. In the context of pure functional programming further modularity concerns arise from the need to model a program's side effects explicitly using *monads* [8], whose syntax and implementation we would ideally define separately and in a modular fashion.

Traditionally, these modularity questions are tackled in functional languages by embedding the *initial algebra semantics* [4] of inductive data types. This approach was popularized by Swierstra's *Data Types à la Carte* [11] as a solution to the expression problem, and was later applied to modularize the syntax and semantics of both first-order and higher-order effectful computations [7, 15, 10, 12] through various kinds of inductively defined *free monads*. The key idea that unifies these approaches is the use of *signature functors* that act as a syntactic representation of inductive data types or inductively defined free monads, from which we recover the desired structure using a type-level fixpoint. This separation of syntax and recursion permits the composition of data types and effect trees by means of a general co-product of signature functors, an operation that is not available for native data types. However, while embedding signature functors is a tremendously useful technique for enhancing functional languages with a higher degree of (type-safe) modularity, there are still some downsides to the approach.

Problem statement. Since are working an embedding of the semantics of data types, we introduce an additional layer of indirection that causes some encoding overhead due to a lack of interoperability with built-in data types. Furthermore, the connection with the underlying categorical concepts that motivate these embeddings remains implicit. By keeping the motivating concepts implicit, our programs lack a rigorously defined formal semantics, but we also introduce further encoding overhead. That is, we usually have to define typeclass instances or work with a *universe construction* [1] to ensure that signatures are indeed functorial.

This work. We advocate an alternative approach that makes the functional programmer's modularity toolkit—e.g., functors, folds, fixpoints, etc.—part of the language's design. We believe that this has the potential to address the issues outlined above. By incorporating these elements into a language's design we have the opportunity to develop more convenient syntax for working with extensible data types (see e.g. the authors' previous work [13]), and by defining a formal semantics we maintain a tight connection between the used modularity abstractions and the concepts that motivate these constructs. The aim of this work is to present a core calculus that acts as a minimal basis for capturing the modularity abstractions discussed here, as well as to develop a formal categorical semantics for this calculus.

Calculus Design and Semantics. We present a λ-calculus with kinds and Hindley-Milner style polymorphism. Types are restricted such that any higher-order type expression is by construction a functor in all its arguments, effectively making the concept of functors first-class in the language's design. By imposing this additional structure, we can provide the programmer with several additional primitives that can be used to capture the aforementioned modularity abstractions, while simultaneously keeping a closer connection to the categorical semantics of these abstractions. Well-formedness of types is defined as usual for the first-order fragment of System F_ω, the only salient difference being that we maintain a separate context, Φ, containing the free variables that a type expression is intended to be functorial in. Type-level λ-abstraction adds a new binding to Φ, and we discard all functorial variables in the domain of a function to enforce that the variables in Φ are only used covariantly:

$$\frac{\Delta \mid \Phi, (X \mapsto k_1) \vdash \tau : k_2}{\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2} \qquad \frac{\Delta \mid \emptyset \vdash \tau_1 : \star \qquad \Delta \mid \Phi \vdash \tau_2 : \star}{\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star}$$

This ensures that all higher-order types have a semantics as objects in an appropriate functor category. The variables in Δ have mixed variance and are bound by universal quantification.

The functor semantics of a type $\tau : k_1 \rightsquigarrow k_2$ guarantees that we can map over values of type τ, provided we have a way to transform the argument type. We expose this ability to the programmer by adding a general mapping primitive to the calculus:

$$\frac{\Delta \mid \epsilon \vdash \tau : k_1 \rightsquigarrow k_2 \qquad \Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2}{\Gamma \vdash \mathbf{map}^{\tau}(M) : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2} \qquad \begin{aligned} \sigma \xrightarrow{\star} \tau &= \sigma \Rightarrow \tau \\ \sigma \xrightarrow{k_1 \rightsquigarrow k_2} \tau &= \forall \alpha.\ \sigma(\alpha) \xrightarrow{k_2} \tau(\alpha) \end{aligned}$$

We use the syntax $\tau_1 \xrightarrow{k} \tau_2$ to denote a (polymorphic) function that universally closes over all type arguments of τ_1 and τ_2, provided that they have the same kind.

Generally speaking, the intended semantics of a terms is a natural transformation between functors over a bicartesian closed category $\mathcal{C}$. We reify this underlying categorical structure through primitives such as **map**. Other examples of such primitives are operations for destructing fixpoints or co-products:

$$\text{T-FOLD}\ \frac{\Gamma \vdash M : \tau_1(\tau_2) \xrightarrow{k} \tau_2}{\Gamma \vdash \mathbf{fold}^{\tau_1}(M) : \mu(\tau_1) \xrightarrow{k} \tau_2} \qquad \text{T-JOIN}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau \qquad \Gamma \vdash M : \tau_2 \xrightarrow{k} \tau}{\Gamma \vdash M \blacktriangledown N : \tau_1 \oplus \tau_2 \xrightarrow{k} \tau}$$

To justify these operations we must argue that terms of type $\tau_1 \xrightarrow{k} \tau_2$ represent morphisms in the (functor) category associated with k.

As an example, we compare definitions of the free monad in our calculus (l) and Haskell (r):

$$\mathit{Free} \triangleq \lambda F.\lambda A.\mu X.A \oplus F(X) \qquad \mathbf{data}\ \mathit{Free}\ f\ a = \mathit{Pure}\ a \mid \mathit{In}\ (f\ (\mathit{Free}\ f\ a))$$

Free is well-formed with kind $(\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star$. Consequently, it is by construction a functor in both type arguments, guaranteeing that we can always map over either of its arguments:

$$\mathbf{map}^{\mathit{Free}(\gamma)}(f) : \forall\alpha.\forall\beta.\forall\gamma.\mathit{Free}(\gamma)(\alpha) \Rightarrow \mathit{Free}(\gamma)(\beta) \qquad \text{where } f : \alpha \Rightarrow \beta$$
$$\mathbf{map}^{\mathit{Free}}(f) : \forall\alpha.\forall\gamma_1.\forall\gamma_2.\mathit{Free}(\gamma_1)(\alpha) \Rightarrow \mathit{Free}(\gamma_2)(\alpha) \qquad \text{where } f : \forall\alpha.\gamma_1(\alpha) \Rightarrow \gamma_2(\alpha)$$

In Haskell, we would require dedicated instances to witness that *Free* is a (higher-order) functor.

Existing Work. There is some previous work that attacks similar problems [9, 3], but to the best of our knowledge no existing language design can capture the modularity abstractions discussed in this abstract and has a clearly defined categorical semantics. Closest to our work, and a major source of inspiration, is a calculus developed by Johann al. [6, 5] for studying parametricity for nested data types [2]. Still, there are some key differences: in their setting universal quantification is limited to zero-argument types, and the semantics is tied to the category of sets, and relies on an additional interpretation of types as relations.

Conclusion. We have designed a calculus that demonstrates how support for type-safe modularity can be integrated into a programming language's design in a principled way, which we intend as a stepping stone for designing functional languages with better facilities for type-safe modularity. We are finalizing the semantic model that relates this support for modularity to the categorical concepts that motivate it.

References

[1] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[2] Richard S. Bird and Lambert G. L. T. Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[3] Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System f-omega with equirecursive types for datatype-generic programming. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 30–43. ACM, 2016.

[4] Joseph A Goguen. An intial algebra approach to the specification, correctness and implementation of abstract data types. *IBM Research Report*, 6487, 1976.

[5] Patricia Johann and Enrico Ghiorzi. Parametricity for nested types and gadts. *Log. Methods Comput. Sci.*, 17(4), 2021.

[6] Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. Parametricity for primitive nested types. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 2021.

[7] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013.

[8] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[9] J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, 2019.

[10] Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL):1801–1831, 2023.

[11] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[12] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent effects for reusable language components. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 182–201. Springer, 2021.

[13] Cas van der Rest and Casper Bach Poulsen. Towards a language for defining reusable programming language components - (project paper). In Wouter Swierstra and Nicolas Wu, editors, *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*, volume 13401 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2022.

[14] Phil Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998. Accessed: 2020-07-01.

[15] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 1–12. ACM, 2014.