Types and Semantics for Extensible Data Types

Cas van der Rest[a,1] Casper Bach Poulsen[a,2]

[a] *Department of Software Technology*
Technische Universiteit Delft
Delft, The Netherlands

Abstract

Developing and maintaining software commonly requires (1) adding new data type constructors to existing applications, but also (2) adding new functions that work on existing data. Most programming languages have native support for defining data types and functions in a way that supports either (1) or (2), but not both. This lack of native support makes it difficult to use and extend libraries. For this reason, there is a proliferation of work that describes how to encode data types and functions in a way that works around the lack of native support. Many of these approaches encode *initial algebra semantics*. While initial algebra encodings solve the modularity problem, they are also syntactically noisy, add interpretive overhead, and commonly fail to take advantage of the map and fold fusion laws of initial algebras which compilers could exploit to generate more efficient code. In this paper, we present a type discipline and core calculus for a language with native support for initial algebra semantics. We develop its formal semantics, and illustrate how the calculus provides native support for state of the art type safe modular programming techniques.

Keywords: Type Systems, Modularity, Language Design, Categorical Semantics

1 Introduction

A common litmus test for a programming language's capability for modularity is whether a programmer is able to extend existing data with new ways to construct it as well as to add new functionality for this data. All in a way that preserves static type safety; a conundrum which Wadler [32] dubbed the *expression problem*. When working in pure functional programming languages, another modularity question is how to model side effects modularly using, e.g., *monads* [23]. Ideally, we would keep the specific monad used to model the effects of a program abstract and program against an *interface* of effectful operations instead, defining the syntax and implementation of such interfaces separately and in a modular fashion.

The traditional approach for tackling these modularity questions in pure functional programming languages is by embedding the *initial algebra semantics* [13] of inductive data types in the language's type system. By working with such embeddings in favor of the language's built-in data types we gain modularity without sacrificing type safety. This approach was popularized by Swierstra's *Data Types à la Carte* [28] as a solution to the expression problem, where it was used to derive modular interpreters for a small expression language. In later work, similar techniques were applied to define the syntax and implementation of a large class of monads using (algebraic) effects and handlers based on different flavors of inductively defined *free monads*. This was shown to be an effective technique for modularizing both first order [17] and higher order [34,26,30] effectful computations.

[1] Email: c.r.vanderrest@tudelft.nl
[2] Email: c.b.poulsen@tudelft.nl

The key idea that unifies these techniques is the use of *signature functors*, which act as a de facto syntactic representation of an inductive data type or inductively defined free monad. Effectively, this allows us to define a "generic" inductive data type or free monad that takes its constructors as a parameter. The crucial benefit of this setup is that we can compose data types and effects by taking the coproduct of signature functors, and we can compose function cases defined over these signature functors in a similarly modular way. Inductive data types and functions in mainstream functional programming languages generally do not support these kinds of composition.

While embedding signature functors has proven itself as a tremendously useful technique for enhancing functional languages with a high degree of type safe modularity, the approach has some downsides:

- By working with an encoding the initial algebra semantics of data types, we lose some syntactic convenience compared to native data types, especially when it comes to constructing and pattern matching on values, making programs harder to read. Further encoding overhead arises from the lack of interoperability with native data types, which is typically only possible by defining specific isomorphisms for each data type/encoding pair.
- The encodings are only implicitly connected to the mathematical concepts that justify them. It is only on the meta-level (and often informally) that we can argue that our definitions correspond to the initial algebra semantics of inductive data types. This has two drawbacks. First, the programmer is required to supply boilerplate definitions to witness that definitions have the right structure, for example, by defining type class instances to establish that signatures are functorial. Second, the compiler cannot benefit from specific properties associated with this mathematical structure, such as by implementing (provably correct) optimizations based on the well-known map and fold fusion laws.

1.1 Contributions

In this paper, we advocate an alternative approach by making the functional programmer's modularity toolkit—e.g., functors, folds, fixpoints, etc.—part of a language's design. This approach has the potential to address both issues outlined above: by designing languages with type-safe modularity in mind we have the opportunity to present the programmer with more convenient syntax for working with extensible data types (see, for example, the language design proposed by Van der Rest and Bach Poulsen [31]). Simultaneously, by incorporating the mathematical structures that drive type safe modularity into a language's design we create the possibility for compilers to benefit from its properties, for example by applying fusion based optimizations.

We investigate this approach by developing a core calculus for programming with extensible data types that is intended as minimal basis for capturing the programming patterns for type safe modularity discussed here in the introduction. The goal of designing this calculus is to provide a firm foundation for understanding the typing and semantics of programming languages that incorporate type safe modularity as part of their design, and to serve as a stepping stone for the development of new programming languages with more extensive facilities for type safe modularity grounded in a solid theoretical understanding of their semantics.

More specifically, we make the following technical contributions:

- We show (in Section 2) how, using our calculus, one can capture modular functions over algebraic data types in the style of Data Types à la Carte, and modular definitions of first-order and higher-order (algebraic) effects and handlers based on inductively defined free monads.
- We present (in Section 3) a formal definition of the syntax and type system.
- We give (in Section 4) a formal semantics for our calculus.

Section 5 discusses related work, and Section 6 concludes.

2 Programming with Extensible Data Types, by Example

The basis of our calculus is the polymorphic λ-calculus extended with kinds and restricted to rank-1 polymorphism, allowing the definition of many familiar polymorphic functions, such as $(id : \forall\alpha.\alpha \Rightarrow \alpha) = \lambda x.x$ or $(const : \forall\alpha.\forall\beta.\alpha \Rightarrow \beta \Rightarrow \alpha) = \lambda x.\lambda y.x$. The set of types is closed under products and coproducts,

and includes a unit type ($\mathbb{1}$) and empty type ($\mathbb{0}$) acting as their respective units. Furthermore, we include a type-level fixpoint (μ), which can be used to encode many well-known algebraic data types. For example, the familiar type of lists is encoded as $\mathit{List} \triangleq \lambda\alpha.\mu(\lambda X.\mathbb{1} \oplus (\alpha \otimes X))$. A key feature of the calculus is that all higher-order types (i.e., that have one or more type argument) are, by construction, functorial in all their arguments. While this imposes some restrictions on the types we can define, it also means that the programmer gets access to primitive mapping and folding operations that they would otherwise have to define themselves. For the type *List*, for example, this means that we get both the usual mapping operation transforming its elements, as well as an operation corresponding to Haskell's *foldr*, for free.

Although the mapping and folding primitives for first-order type constructors (i.e., those taking arguments of kind $\star$ and producing a type of kind $\star$) are already enough to solve the expression problem for regular algebraic data types (Section 2.1) and encode modular algebraic effects (Section 2.2), they can readily be generalized to higher-order type constructors. That is, type constructors that construct higher-order types from higher-order types. The benefit of this generalization is that our calculus can also capture the definition of so-called *nested data types* [5], which arise as the fixpoint of a *higher-order functor*. We make essential use of the calculus' higher-order capabilities in Section 2.3 to define modular handlers for scoped effects [35] and modular elaborations for higher-order effects [26], as in both cases the type of effect trees that represents monadic programs with higher-order operations is defined as a nested data type.

2.1 Modular Interpreters in the style of Data Types à la Carte

We consider how to define a modular interpreter for a small expression language of simple arithmetic operations. For starters, we just include literals and addition. The corresponding BNF equation and signature functor are given below:

$$e := \mathbb{N} \mid e + e \qquad\qquad \mathit{ExprF} \triangleq \lambda X.\mathbb{N} \oplus (X \otimes X)$$

Now, we can define an *eval* that maps expressions—given by the fixpoint of *ExprF*—to their result:

$$\begin{aligned} &\mathit{evalAlg} : \mathbb{N} \oplus (\mathbb{N} \otimes \mathbb{N}) \Rightarrow \mathbb{N} \\ &\mathit{evalAlg} = (\lambda x.x) \blacktriangledown (\lambda x.\boldsymbol{\pi_1}\ x + \boldsymbol{\pi_2}\ x) \end{aligned} \qquad\qquad \begin{aligned} &\mathit{eval} : \mu(\mathit{ExprF}) \Rightarrow \mathbb{N} \\ &\mathit{eval} = (\!|\ \mathit{evalAlg}\ |\!)^{\mathit{ExprF}} \end{aligned}$$

Terms typeset in **purple** are built-in operations. $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ are the usual projection functions for products, and $- \blacktriangledown -$ is an eliminator for coproducts. Following Meijer et al. [22], we write $(\!|\ \mathit{alg}\ |\!)^{\tau}$ (i.e., "banana brackets") to denote a fold over the type $\mu(\tau)$ with an *algebra* of type $\mathit{alg} : \tau\ \tau' \Rightarrow \tau'$. The calculus does not include a general term level fixpoint; the only way to write a function that recurses on the substructures of a μ-type is by using the built-in folding operation. While this limits the operations we can define for a given type, it also ensures that all well-typed terms in the calculus have a well-defined semantics.

Now, we can extend this expression language with support for a multiplication operation as follows, where $\mathit{MulF} \triangleq \lambda X.X \otimes X$:

$$\begin{aligned} &\mathit{mulAlg} : \mathbb{N} \otimes \mathbb{N} \\ &\mathit{mulAlg} = \lambda x.\boldsymbol{\pi_1}\ x * \boldsymbol{\pi_2}\ x \end{aligned} \qquad\qquad \begin{aligned} &\mathit{eval} : \mu(\mathit{ExprF} \oplus \mathit{MulF}) \Rightarrow \mathbb{N} \\ &\mathit{eval} = (\!|\ \mathit{evalAlg} \blacktriangledown \mathit{mulAlg}\ |\!)^{\mathit{ExprF} \oplus \mathit{MulF}} \end{aligned}$$

2.2 Modular Algebraic Effects using the Free Monad

As our second example we consider how to define modular algebraic effects and handlers [25] in terms of the free monad following Swierstra [28]. First, we define the *Free* type which constructs a free monad for a given signature functor f. We can think of a term with type $\mathit{Free}\ f\ \alpha$ as a syntactic representation of a monadic program producing a value of type α with f describing the operations which we can use to interact with the monadic context.

$$\mathit{Free} : (\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star \quad \triangleq \quad \lambda f.\lambda\alpha.\mu(\lambda X.\alpha \oplus f X)$$

Note that the type *Free* is actually a functor in both its arguments, and thus there are two ways to "map over" a value of type $\mathit{Free}\ f\ \alpha$; we can transform the values at the leaves using a function $\alpha \Rightarrow \beta$, or the

shape of the nodes using a natural transformation $\forall\alpha.f\ \alpha \Rightarrow g\ \alpha$. The higher order map can be used, for example, for defining function that reorders the operations of effect trees with a composite signature.

$$\begin{aligned} &\mathit{reorder} : \forall f.\forall g.\forall\alpha.\mathit{Free}\ (f \oplus g)\ \alpha \Rightarrow \mathit{Free}\ (g \oplus f)\ \alpha \\ &\mathit{reorder} = \mathbf{map}\langle \boldsymbol{\iota_2} \blacktriangledown \boldsymbol{\iota_1} \rangle^{\mathit{Free}} \end{aligned}$$

Here, we use higher order instances at kind $\star \rightsquigarrow \star$ of the coproduct eliminator $- \blacktriangledown -$, the coproduct injection functions $\boldsymbol{\iota_1}$, $\boldsymbol{\iota_2}$, and the functorial map operation $\mathbf{map}\langle - \rangle^{-}$.

Effect handlers can straightforwardly be implemented as folds over *Free*. In fact, the behavior of a handler is entirely defined by the algebra that we use to fold over the effect tree, allowing us write a generic *handle* function:

$$\begin{aligned} &\mathit{handle} : \forall f.\forall g.\forall\alpha.\forall\beta.(\alpha \Rightarrow \beta) \Rightarrow (f\ (\mathit{Free}\ g\ \beta) \Rightarrow \mathit{Free}\ g\ \beta) \Rightarrow \mathit{Free}\ (f \oplus g)\ \alpha \Rightarrow \mathit{Free}\ g\ \beta \\ &\mathit{handle} = \lambda h.\lambda i.(\!|\ (\mathbf{in} \circ \boldsymbol{\iota_1} \circ h) \blacktriangledown i \blacktriangledown (\mathbf{in} \circ \boldsymbol{\iota_2})\ |\!)^{\alpha \oplus (f X) \oplus (g X)} \end{aligned}$$

Here, $\mathbf{in}$ is the constructor of a type-level fixpoint (μ). The fold above distinguishes three cases: (1) pure values, in which case we return it again using the function h; (2) an operation of the signature f which is handled using the function i; or (3) an operation of the signature g which is preserved by reconstructing the effect tree and doing nothing.

As an example, we consider how to implement a handler for the *Abort* effect, which has a single operation indicating abrupt termination of a computation. We define its signature functor as follows:

$$\mathit{Abort} : \star \rightsquigarrow \star \quad \triangleq \quad \lambda X.\mathbb{1}$$

The definition of *Abort* ignores its argument, X, which is the type of the continuation. After aborting a computation, there is no continuation, thus the *Abort* effect does not need to store one. A handler for *Abort* is then defined like so, invoking the generic *handle* function defined above:

$$\begin{aligned} &\mathit{hAbort} : \forall f.\forall\alpha.\mathit{Free}\ (\mathit{Abort} \oplus f)\ \alpha \Rightarrow \mathit{Free}\ f\ (\mathit{Maybe}\ \alpha) \\ &\mathit{hAbort} = \mathit{handle}\ \mathit{Just}\ (\lambda x.\mathbf{in}\ (\boldsymbol{\iota_1}\ \mathit{Nothing})) \end{aligned}$$

2.3 Modular Higher-Order Effects

To describe the syntax of computations that interact with their monadic context through higher-order operations—that is, operations whose arguments can themselves also be monadic computations—we need to generalize the free monad as follows.

$$\mathit{Prog} : ((\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star \quad \triangleq \quad \lambda f.\mu(\lambda X.\lambda\alpha.\alpha \oplus (f\ X\ \alpha))$$

Note that, unlike the *Free* type, *Prog* is defined as the fixpoint of a higher-order functor. This generalization allows for signature functors to freely choose the return type of continuations. Following Yang et al. [35], we use this additional expressivity to describe the syntax of higher-order operations by nesting continuations. For example, the following defines the syntax of an effect for exception catching, that we can interact with by either throwing an exception, or by declaring an exception handler that first executes its first argument, and only runs the second computation if an exception was thrown.

$$\mathit{Catch} : (\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star \quad \triangleq \quad \lambda X.\lambda\alpha.\mathbb{1} \oplus (X(X\alpha) \otimes (X(X\alpha))$$

A value of type *Prog Catch* α is then a syntactic representation of a monadic program that can both throw and catch exceptions. From this syntactic representation we can proceed in two different ways. The first option is to replace exception catching with an application of the *hAbort* handler, in line with Plotkin and Pretnar's [25] original strategy for capturing higher-order operations. In recent work, Bach Poulsen and Van der Rest [26] demonstrated how such abbreviations can be made modular and reusable

$$
\begin{array}{rcll}
\alpha, \beta, \gamma, X, Y & \in & \text{String} & \text{(Type variables)} \\
\\
k & := & \star \mid k \rightsquigarrow k & \text{(Kinds)} \\
\Delta, \Phi & := & \emptyset \mid \Delta, (\alpha \mapsto k) & \text{(Kind environments)} \\
\\
\tau & := & \alpha \mid X \mid \tau\ \tau \mid \lambda X.\tau \mid \mu(\tau) \mid \tau \Rightarrow \tau & \text{(Types)} \\
 & \mid & \mathbb{0} \mid \mathbb{1} \mid \tau \otimes \tau \mid \tau \oplus \tau & \\
\sigma & := & \forall \alpha.\sigma \mid \tau & \text{(Type Schemes)}
\end{array}
$$

Fig. 1. Type syntax

by implementing them as algebras over the *Prog* type. Following their approach, we define the following elaboration of exception catching into a first-order effect tree.

$$
\begin{aligned}
&\mathit{eCatch} : \forall \alpha.\mathit{Prog}\ \mathit{Catch}\ \alpha \Rightarrow \mathit{Free}\ \mathit{Abort}\ \alpha \\
&\mathit{eCatch} = (\!|\ (\mathbf{in} \circ \boldsymbol{\iota_1})\ \blacktriangledown\ (\mathbf{in} \circ \boldsymbol{\iota_2})\ \blacktriangledown\ (\lambda x.\mathit{hAbort}\ (\boldsymbol{\pi_1}\ x) \gg\!= \mathit{maybe}\ (\mathit{join}\ (\boldsymbol{\pi_2}\ x))\ \mathit{id})\ |\!)^{\alpha \oplus \mathit{Catch}\ X\ \alpha}
\end{aligned}
$$

Here, the applications of monadic bind ($\gg\!=$) and *join* refer to the monadic structure of *Free*. Alternatively, we can define a handler for exception catching directly by folding over the *Prog* type, following the *scoped effects* approach by Wu et al. [34]:

$$
\begin{aligned}
&\mathit{hCatch} : \forall h.\forall \alpha.\mathit{Prog}\ (\mathit{Catch} \oplus h)\ \alpha \Rightarrow \mathit{Prog}\ h\ (\mathit{Maybe}\ \alpha) \\
&\mathit{hCatch} = (\!|\ (\mathbf{in} \circ \boldsymbol{\iota_1} \circ \mathit{Just})\ \blacktriangledown\ (\lambda x.\mathbf{in}\ (\boldsymbol{\iota_1}\ \mathit{Nothing})) \\
&\qquad \blacktriangledown\ (\lambda x.\boldsymbol{\pi_1}\ x \gg\!= \mathit{maybe}\ (\boldsymbol{\pi_2}\ x \gg\!= \mathit{fwd})\ \mathit{id}))\ \blacktriangledown\ (\mathbf{in} \circ \boldsymbol{\iota_2})\ |\!)^{\alpha \oplus (\mathit{Catch}\ X\ \alpha) \oplus (h\ X\ \alpha)}
\end{aligned}
$$

Where the function *fwd* establishes that *Maybe* commutes with the *Prog* type in a suitable way:

$$
\mathit{fwd} : \forall h.\forall \alpha.\mathit{Maybe}\ (\mathit{Prog}\ h\ (\mathit{Maybe}\ \alpha)) \Rightarrow \mathit{Prog}\ h\ (\mathit{Maybe}\ \alpha)
$$

In other words, we show that *Prog h* is what Schrijvers et al. [27] call a *modular carrier* for *Maybe*.

3 The Calculus

The previous section demonstrated how a language with built-in support for functors, folds, and fixpoints provides support for defining and working with state of the art techniques for type safe modular programming techniques. In this section we present a calculus for such a language.

The basis of our calculus is the first-order fragment of System F^{ω}—i.e., the polymorphic λ-calculus with kinds, where universal quantification is restricted to prenex normal form à la Hindley-Milner; i.e., universal quantification only occurs at the top-level of a type ($\overline{\forall \alpha}.\tau$). Additionally, the syntax of types, shown in Figure 1, includes primitives for constructing recursive types ($\mu(-)$), products ($\otimes$) and coproducts ($\oplus$), as well as a unit type ($\mathbb{1}$) and empty type ($\mathbb{0}$). Occurrences of universal quantifications (i.e., $\forall$-types) are syntactically restricted by stratifying the syntax of types into two layers, types and type schemes. This implies that our calculus is, by design, *predicative*: $\forall$-types can quantify over any type but not over type schemes, precluding programs with higher-rank polymorphism where type variables stand for $\forall$-types.

3.1 Well-Formed Types

Types are well-formed with respect to a kind k, describing the arity of a type's parameters, if it has any. The set of all kinds is defined by iterating over the base kind $\star$—the kind of types that have no arguments. Well-formedness of types is defined using the judgment $\Delta \mid \Phi \vdash \tau : k$, stating that the type τ has kind k

$\boxed{\Delta \mid \Phi \vdash \tau : k}$

$$\textsc{K-Var}\ \frac{\Delta(\alpha) \mapsto k}{\Delta \mid \Phi \vdash \alpha : k} \qquad \textsc{K-Fvar}\ \frac{\Phi(X) \mapsto k}{\Delta \mid \Phi \vdash X : k} \qquad \textsc{K-App}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k_1 \rightsquigarrow k_2 \qquad \Delta \mid \Phi \vdash \tau_2 : k_1}{\Delta \mid \Phi \vdash \tau_1\ \tau_2 : k_2} \qquad \textsc{K-Abs}\ \frac{\Delta \mid \Phi, (X \mapsto k_1) \vdash \tau : k_2}{\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2}$$

$$\textsc{K-Fix}\ \frac{\Delta \mid \Phi \vdash \tau : k \rightsquigarrow k}{\Delta \mid \Phi \vdash \mu(\tau) : k} \qquad \textsc{K-Fun}\ \frac{\Delta \mid \emptyset \vdash \tau_1 : \star \qquad \Delta \mid \Phi \vdash \tau_2 : \star}{\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star} \qquad \textsc{K-Empty}\ \frac{}{\Delta \mid \Phi \vdash \mathbb{0} : \star} \qquad \textsc{K-Unit}\ \frac{}{\Delta \mid \Phi \vdash \mathbb{1} : \star}$$

$$\textsc{K-Product}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k \qquad \Delta \mid \Phi \vdash \tau_2 : k}{\Delta \mid \Phi \vdash \tau_1 \otimes \tau_2 : k} \qquad \textsc{K-Sum}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k \qquad \Delta \mid \Phi \vdash \tau_2 : k}{\Delta \mid \Phi \vdash \tau_1 \oplus \tau_2 : k}$$

$\boxed{\Delta \vdash \sigma}$

$$\textsc{SC-Forall}\ \frac{\Delta, (\alpha \mapsto k) \vdash \sigma}{\Delta \vdash \forall \alpha.\sigma} \qquad \textsc{SC-Type}\ \frac{\Delta \mid \emptyset \vdash \tau : \star}{\Delta \vdash \tau}$$

Fig. 2. Well-formedness rules for types and type schemes

under contexts Δ and Φ. Similarly, well-formedness of type schemes is defined in terms of the judgment $\Delta \vdash \sigma$, stating that the type scheme σ is well-formed with respect to the context Δ. We maintain two contexts when defining well-formedness for types in order to distinguish between those free variables a type expression is intended to be functorial in, and those that can be used freely. More specifically, the variables in the context Φ are restricted to occur only in *strictly positive* [1,10] positions (i.e., they can never appear to the left of a function arrow), while the variables in Δ can have mixed variance. This restriction on the occurrence of the variables in Φ is enforced in the well-formedness rule for function types, K-Fun, which requires that its domain is typed under an empty context of functorial variables, preventing the domain type from dereferencing any functorial variables bound in the surrounding context. Variables in Φ are bound by type-level λ-abstraction, meaning that any higher-order type (that is, of the form $k_1 \rightsquigarrow k_2$) is functorial in its argument of kind k_1. In contrast, the variables in Δ are bound by $\forall$-quantification.

The well-formed judgements for types effectively define a (simply typed) type level λ-calculus with base "type" $\star$. Consequently, the same type has multiple equivalent representations in the presence of β-redexes, raising the question of how we should deal with type normalization. The solution we adopt here is to add a non-syntactic conversion rule to the definition of our type system that permits any well-formed term to be typed under an "equivalent" type. A precise definition of equivalence for types and type schemes is beyond the scope of the current paper, but crucially it should be defined such that syntactic type equivalence is reflected in the semantic domain as well. Looking ahead to the semantics of types define in Section 4.2, in our case this means that equivalent types should be interpreted as naturally isomorphic functors. This way, for example, we can warrant the addition of a β-conversion rule to our definition of type equivalence, motivated by the conventional β-equality associated with Cartesian closed categories (in this case it is the Cartesian closed structure of the category of categories that supplies the required equality).

3.2 Well-Typed Terms

Figure 3 shows the term syntax of our calculus. Along with the standard syntactic forms of the polymorphic λ-calculus we include explicit type abstraction and application, as well as introduction and elimination forms for recursive types (**in**/**out**), products ($\boldsymbol{\pi_1}/\boldsymbol{\pi_2}/- \blacktriangle -$), coproducts ($\boldsymbol{\iota_1}/\boldsymbol{\iota_2}/- \blacktriangledown -$), and the unit (**tt**) and empty (**absurd**) types. Furthermore, the calculus includes dedicated primitives for mapping ($\mathbf{map}\langle - \rangle^-$) and folding ($(\!| - |\!)^-$) over a type.

Figure 3 also includes the definition of *arrow types*. An arrow type of the form $\tau_1 \xrightarrow{k} \tau_2$ (where $\tau_1, \tau_2 : k$) defines the type of *morphisms* between the objects that interpret τ_1 and τ_2. They are defined

$$
\begin{array}{rcll}
x, y & \in & \text{String} & \text{(Variables)} \\
\\
\Gamma & := & \emptyset \mid \Gamma, (x \mapsto \sigma) & \text{(Type environments)} \\
M, N & := & x \mid M\ N \mid \lambda x.M \mid \textsf{let}\ (x : \sigma) = M\ \textsf{in}\ N \mid \Lambda\alpha.M \mid M\ @\tau & \text{(Terms)} \\
& \mid & \textsf{in} \mid \textsf{out} \mid \textsf{map}\langle M \rangle^{\tau} \mid (\!|\ M\ |\!)^{\tau} & \\
& \mid & \pi_1 \mid \pi_2 \mid M \blacktriangle N \mid \iota_1 \mid \iota_2 \mid M \blacktriangledown N \mid \textsf{tt} \mid \textsf{absurd} &
\end{array}
$$

$$
\begin{array}{rcll}
\tau_1 \xrightarrow{\star} \tau_2 & \triangleq & \tau_1 \Rightarrow \tau_2 & \text{(Arrow Types)} \\
\tau_1 \xrightarrow{(k_1 \rightsquigarrow k_2)} \tau_2 & \triangleq & \forall\alpha.\ \tau_1\ \alpha \xrightarrow{k_2} \tau_2\ \alpha & \\
\multicolumn{2}{r}{\textit{where}} & \Delta \vdash \tau_1 \xrightarrow{k} \tau_2 \quad \textit{if} \quad \Delta \mid \emptyset \vdash \tau_1, \tau_2 : k &
\end{array}
$$

Fig. 3. Term syntax

by induction over k, since the precise meaning of "morphism" for any pair of type depends on their kind. If $k = \star$, then a morphism between τ_1 and τ_2 is simply a function type. However, if τ_1 and τ_2 have one or more type argument, they are to be interpreted as objects in a suitable functor category, meaning that their morphisms are the natural transformations between the corresponding functors. This is reflected in the definition of arrow types by unfolding an arrow $\tau_1 \xrightarrow{k} \tau_2$ to a $\forall$-type that closes over all type arguments of τ_1 and τ_2, capturing the usual intuition that polymorphic functions correspond to natural transformations. For instance, we would type the inorder traversal of binary trees as $\mathit{inorder} : \mathit{Tree} \xrightarrow{\star \rightsquigarrow \star} \mathit{List}$ ($\triangleq \forall\alpha.\mathit{List}\ \alpha \Rightarrow \mathit{Tree}\ \alpha$), describing a natural transformation between the *Tree* and *List* functors.

The typing rules of our calculus are shown in shown in Figure 4. The rules use arrow types extensively for introduction and elimination forms. For example, since products can be constructed at any kind (following rule K-Product in Figure 2), the rules for terms that operate on these, such as T-Fst, T-Snd, and T-Fork also use arrow types at any kind k. For this reason, arrow types should correspond to morphisms in a suitable category, such that the semantics of a product type and its introduction/elimination forms can be expressed in terms of morphisms in this category.

4 Semantics

To define a formal semantics for our calculus, we interpret well-formed types and type schemes as functors from a category whose objects correspond to their contexts into a category whose objects correspond to kinds. We start by picking suitable base categories $\boldsymbol{\mathcal{C}}$ (large) and $\boldsymbol{\mathcal{D}}$[3] (very large) such that $\boldsymbol{\mathcal{C}}$ is a *full subcategory* of $\boldsymbol{\mathcal{D}}$, as witnessed by a (fully faithful) inclusion functor I.

$$\boldsymbol{\mathcal{C}} \overset{I}{\hookrightarrow} \boldsymbol{\mathcal{D}}$$

We require that $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{D}}$ are both *complete* and *cocomplete*, and that $\boldsymbol{\mathcal{C}}$ is *cartesian closed*. Importantly, since I is fully faithful, the cartesian closed structure of $\boldsymbol{\mathcal{C}}$ is reflected in $\boldsymbol{\mathcal{D}}$ for those objects that lie in the image of I. Additional conditions on $\boldsymbol{\mathcal{C}}$ arise when considering the requirement that the functors corresponding to higher-order types should have initial algebras; we discuss this in Section 4.3. The intuition behind our setup is that well-formed types are interpreted as objects in $\boldsymbol{\mathcal{C}}$ and its endofunctor categories (depending on their kind), and well-formed type schemes as objects in $\boldsymbol{\mathcal{D}}$. The subcategory relation between $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{D}}$ reflects the syntactic restriction of types to rank-1 polymorphism: all objects in $\boldsymbol{\mathcal{C}}$ can be found in $\boldsymbol{\mathcal{D}}$, but $\boldsymbol{\mathcal{D}}$ is sufficiently larger than $\boldsymbol{\mathcal{C}}$ that it also includes objects modelling quantification over objects in $\boldsymbol{\mathcal{C}}$. This intuition is embodied by fact that every functor $\mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \boldsymbol{\mathcal{D}}$, where $\mathcal{C}$ is smaller

[3] We write boldface caligraphic letters $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{D}}$ to refer to the specific base categories used to interpret respectively types with kind $\star$ and type schemes, while other caligraphic letters $(\mathcal{C}, \mathcal{D}, \mathcal{E})$ range over arbitrary categories.

$\boxed{\Gamma \vdash M : \sigma}$

$$\textsc{T-Var}\ \frac{\Gamma(x) \mapsto \sigma}{\Gamma \vdash x : \sigma} \qquad \textsc{T-App}\ \frac{\Gamma \vdash M : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2} \qquad \textsc{T-Abs}\ \frac{\Gamma, (x : \tau_1) \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \Rightarrow \tau_2}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash M : \sigma_1 \qquad \Gamma, (x \mapsto \sigma_1) \vdash N : \sigma_2}{\Gamma \vdash \textsf{let}\ (x : \sigma_1) = M\ \textsf{in}\ N : \sigma_2} \qquad \textsc{T-TypeAbs}\ \frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathsf{freevars}(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma} \qquad \textsc{T-TypeApp}\ \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\ @\tau : \sigma[\tau/\alpha]}$$

$$\textsc{T-In}\ \frac{}{\Gamma \vdash \textsf{in} : \tau\ \mu(\tau) \xrightarrow{k} \mu(\tau)} \qquad \textsc{T-Out}\ \frac{}{\Gamma \vdash \textsf{out} : \mu(\tau) \xrightarrow{k} \tau\ \mu(\tau)} \qquad \textsc{T-Map}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2}{\Gamma \vdash \textsf{map}\langle M \rangle^{\tau} : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2}$$

$$\textsc{T-Fold}\ \frac{\Gamma \vdash M : \tau_1\ \tau_2 \xrightarrow{k} \tau_2}{\Gamma \vdash (\!|\ M\ |\!)^{\tau_1} : \mu(\tau_1) \xrightarrow{k} \tau_2} \qquad \textsc{T-Fst}\ \frac{}{\Gamma \vdash \boldsymbol{\pi_1} : \tau_1 \otimes \tau_2 \xrightarrow{k} \tau_1} \qquad \textsc{T-Snd}\ \frac{}{\Gamma \vdash \boldsymbol{\pi_2} : \tau_1 \otimes \tau_2 \xrightarrow{k} \tau_2}$$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : \tau \xrightarrow{k} \tau_1 \qquad \Gamma \vdash N : \tau \xrightarrow{k} \tau_2}{\Gamma \vdash M \blacktriangle N : \tau \xrightarrow{k} \tau_1 \otimes \tau_2} \qquad \textsc{T-Inl}\ \frac{}{\Gamma \vdash \boldsymbol{\iota_1} : \tau_1 \xrightarrow{k} \tau_1 \oplus \tau_2} \qquad \textsc{T-Inr}\ \frac{}{\Gamma \vdash \boldsymbol{\iota_2} : \tau_2 \xrightarrow{k} \tau_1 \oplus \tau_2}$$

$$\textsc{T-Join}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau \qquad \Gamma \vdash M : \tau_2 \xrightarrow{k} \tau}{\Gamma \vdash M \blacktriangledown N : \tau_1 \oplus \tau_2 \xrightarrow{k} \tau} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash \textsf{tt} : \mathbb{1}} \qquad \textsc{T-Empty}\ \frac{}{\Gamma \vdash \textsf{absurd} : \mathbb{0} \xrightarrow{k} \tau} \qquad \textsc{T-Conv}\ \frac{\Gamma \vdash M : \sigma_1 \qquad \sigma_1 \equiv \sigma_2}{\Gamma \vdash M : \sigma_2}$$

Fig. 4. Well-formed terms

than $\mathcal{D}$, has an end in $\mathcal{D}$ (resulting from completeness [20, p. 224, corollary 2]). We discuss the use of ends to model universal quantification in Section 4.2.3. An example pair of categories that satisfy the assumptions outlined here are the categories of large respectively very large sets. [4]

4.1 Interpreting Kinds and Kind Environments

We associate with each kind k a category whose objects interpret the types of that kind. The semantics of kinds is defined by induction over k, where we map the base kind $\star$ to the category $\mathcal{C}$, and kinds of the form $k_1 \rightsquigarrow k_2$ to the category of functors between their domain and codomain.

$$\begin{aligned} [\![-]\!] &: \mathit{Kind} \to \mathbf{CAT} \\ [\![\star]\!] &= \mathcal{C} \\ [\![k_1 \rightsquigarrow k_2]\!] &= [\, [\![k_1]\!] , [\![k_2]\!] \,] \end{aligned}$$

This setup cements the idea that higher-order types correspond to functors, as their semantics is given by objects in a functor category. The interpretation of contexts is then defined component-wise in terms of

[4] We consider *large* categories (i.e., both the set of objects and morphisms is large) instead of locally small categories because the category of functors between two locally small categories is itself not locally small. Constructing the functor category of two large categories lacks this size escalation, and the resulting category remains large. This ensures that sizes are uniform between the interpretation of both the base kind and arrow kinds, and thus $[\![k]\!] \in \mathbf{CAT}$ for any k. We justify instantiating the base category $\mathcal{C}$ with the (locally small) category of sets by remarking that, assuming cumulative universes, we can view any locally small category as a large category.

$$
\begin{aligned}
[\![\Delta \mid \Phi \vdash \alpha : \tau]\!] &= \mathbf{lookup}^{\Delta}_{\alpha} \circ \pi_2 \circ \pi_1 \\
[\![\Delta \mid \Phi \vdash X : \tau]\!] &= \mathbf{lookup}^{\Phi}_{X} \circ \pi_2 \\
[\![\Delta \mid \Phi \vdash \tau_1\ \tau_2 : k_2]\!] &= \mathsf{eval} \circ \langle\ [\![\Delta \mid \Phi \vdash \tau_1 : k_1 \rightsquigarrow k_2]\!], [\![\Delta \mid \Phi \vdash \tau_2 : k_1]\!]\ \rangle \\
[\![\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2]\!] &= \mathsf{curry}([\![\Delta \mid \Phi, (X \mapsto k_1) \vdash \tau : k_2]\!]) \\
[\![\Delta \mid \Phi \vdash \mu(\tau) : k]\!] &= \boldsymbol{\mu}([\![\Delta \mid \Phi \vdash \tau : k \rightsquigarrow k]\!]) \\
[\![\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star]\!] &= \mathbf{exp}([\![\Delta \mid \emptyset \vdash \tau_1 : \star]\!], [\![\Delta \mid \Phi \vdash \tau_2 : \star]\!]) \\
[\![\Delta \mid \Phi \vdash \mathbb{0} : \star]\!] &= \bot \\
[\![\Delta \mid \Phi \vdash \mathbb{1} : \star]\!] &= \top \\
[\![\Delta \mid \Phi \vdash \tau_1 \otimes \tau_2 : k]\!] &= [\![\Delta \mid \Phi \vdash \tau_1 : k]\!] \times [\![\Delta \mid \Phi \vdash \tau_2 : k]\!] \\
[\![\Delta \mid \Phi \vdash \tau_1 \oplus \tau_2 : k]\!] &= [\![\Delta \mid \Phi \vdash \tau_1 : k]\!] + [\![\Delta \mid \Phi \vdash \tau_2 : k]\!] \\
\\
[\![\Delta \vdash \forall \alpha.\sigma]\!] &= \mathbf{end}(\mathsf{curry}([\![\Delta, (\alpha \mapsto k \vdash \sigma]\!] \circ \mathbf{sift})) \\
[\![\Delta \vdash \tau]\!] &= I \circ [\![\Delta \mid \emptyset \vdash \tau : \star]\!]
\end{aligned}
$$

Fig. 5. Semantics of well-formed types and type schemes

the categories that interpret their elements.

$$
\begin{aligned}
[\![-]\!] &: \mathit{Context} \to \mathbf{CAT} \\
[\![\ \emptyset\ ]\!] &= \bullet \\
[\![\ \Delta, \alpha \mapsto k\ ]\!] &= [\![\ \Delta\ ]\!] \times [\![\ k\ ]\!]
\end{aligned}
$$

Here, $\bullet$ denotes the *trivial category* (with only one object, $*$, and its identity morphism, id_*) and $\times$ the usual product of two categories. It is worth mentioning that $\bullet$, $- \times -$, and $[-, -]$ define a cartesian closed structure for **CAT**, the very large (bi)category of large categories, which will prove useful for interpreting the fragment of well-formed types that corresponds to the simply-typed λ-calculus.

4.2 Interpreting Types

Since a well-formed type $\Delta \mid \Phi \vdash \tau : k$ is intended to be functorial in all variables in Φ, it is clear that its semantics should be a functor over the category associated with Φ (i.e., $[\![\Phi]\!]$). But what about the variables in Δ, which can occur both in covariant and contravariant positions? For example, in the type of the identity function, $\forall\alpha.\alpha \Rightarrow \alpha$, we cannot interpret the sub-derivation for $\alpha \Rightarrow \alpha$ as a functor over the category interpreting its free variables since there would not be a sensible way to define its action on morphisms due to the negative occurence of α. To account for the mixed variance of universally quantified type variables, we instead adopt a *difunctorial semantics*, interpreting types as a functor on the product category $[\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!]$ (similar representations of type expressions with mixed variance appear, for example, when considering Mendler-style inductive types [29], or the object calculus semantics by Glimming and Ghani [12]). Well-formed types (left) and type schemes (right) are interpreted as a functors over their contexts of the following form:

$$
[\![\ \Delta \mid \Phi \vdash \tau : k\ ]\!] : ([\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!]) \times [\![\Phi]\!] \to [\![k]\!] \qquad\qquad [\![\ \Delta \vdash \sigma\ ]\!] : [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!] \to \boldsymbol{\mathcal{D}}
$$

Ultimately, the goal of this setup is to interpret universal quantifications as *ends* in $\boldsymbol{\mathcal{D}}$, which allows us to formally argue that terms that are well-formed with an arrow type of the form $\tau_1 \xrightarrow{k} \tau_2$ (which unfolds to $\forall\bar{\alpha}.\tau_1\ \bar{\alpha} \Rightarrow \tau_2\ \bar{\alpha}$) correspond, in a suitable sense, to the natural transformations between the functors interpreting τ_1 and τ_2. Or, put differently, terms with an arrow type define a morphism between the interpretation of their domain and codomain. We discuss the semantics of universal quantification further in Section 4.2.3, and give a more precise account of the relation between arrow types and natural transformations in Section 4.4.

Figure 5 defines the semantics of well-formed types and type schemes. The interpretation of the empty type, unit type, and (co)product types follow straightforwardly from (co)completeness of the base category $\mathcal{C}$. Since they can be constructed at any kind, the semantics of (co)product types depends crucially on the fact that functor categories preserve all (co)limits of their codomain category, which entails that $[\![k]\!]$ is (co)complete for any k. To interpret variables, we utilize the cartesian closed structure of **CAT** to compute an appropriate projection based on the position of the variable in the environment.

$$\begin{aligned}
\mathbf{lookup}_{\alpha}^{\Delta} &: [\![\ \Delta\ ]\!] \to [\![\ k\ ]\!] \\
\mathbf{lookup}_{\alpha}^{\Delta,(\alpha\mapsto k)} &\mapsto \pi_2 \\
\mathbf{lookup}_{\alpha}^{\Delta,(\beta\mapsto k)} &\mapsto \mathbf{lookup}_{\alpha}^{\Delta} \circ \pi_1 \qquad (\text{where } \alpha \neq \beta)
\end{aligned}$$

Similarly, the cartesian closed structure of **CAT** also implies the existence of functors $\mathsf{eval} : [\mathcal{C}, \mathcal{D}] \times \mathcal{C} \to \mathcal{D}$ and $\mathsf{curry}(F) : \mathcal{C} \to [\mathcal{D}, \mathcal{E}]$, for any $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$, which immediately provide a semantics for type-level application and abstraction respectively. The remaining type and type scheme constructors are interpreted using specially-defined functors. Although their definitions are typical in the sense that objects are computed pointwise from the associated (co)limits in the codomain category, with the action on morphisms arising from their respective universal property, we discuss the construction of these functors separately and in more detail in Section 4.2.1 (recursive types), Section 4.2.2 (function types), and Section 4.2.3 (∀-types).

4.2.1 Recursive Types

Following the usual initial semantics of inductive data types [13], the semantics of recursive types is given by an *initial algebras*. We summarize the categorical setup here. An *F-algebra* for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is defined as a tuple (A, α) of an object $A \in C$ (called the *carrier*), and a morphism $\alpha : FA \to A$. An *algebra homomorphism* between F-algebras (A, α) and (B, β) is given by a morphism $f : A \to B$ such that the following diagram commutes.

$$\begin{array}{ccc}
FA & \xrightarrow{Ff} & FB \\
{\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\
A & \xrightarrow[f]{} & B
\end{array}$$

F-algebras and their homomorphisms form a category. If F is an endofunctor, we denote the initial object of the category of F-algebras (which, if it exists, we refer to as the initial algebra) as $(\mu F, \mathsf{in})$. Initial algebras give a semantics to inductive data types, with their universal property providing an induction principle. Given an F-algebra (A, α), we denote unique F-algebra homomorphism that factors through A by $\mathsf{cata}(\alpha) : \mu F \to A$. Instantiating the diagram above with $\mathsf{cata}(\alpha)$ gives us the familiar universal property of folds, $\mathsf{cata}(\alpha) \circ \mathsf{in} = \alpha \circ F(\mathsf{cata}(\alpha))$, which defines their computational behavior.

To interpret recursive types in our calculus, we construct the functor $\boldsymbol{\mu}(F)$, which sends objects pointwise to the initial algebras of a functor $F : \mathcal{C} \to [\mathcal{D}, \mathcal{D}]$. For a morphism $f : X \to Y$, the action of $\boldsymbol{\mu}(F)$ on f is defined by factoring through the algebra defined by precomposing the initial algebra of $F(Y)$ with the action of F on f, which defines a natural transformation $F(X) \dot{\to} F(Y)$, at component $\mu(F(Y))$.

$$\begin{aligned}
\boldsymbol{\mu}(F)(-) &: \mathcal{C} \to \mathcal{D} \\
\boldsymbol{\mu}(F)(x) &\mapsto \mu(F(x)) \\
\boldsymbol{\mu}(F)(f) &\mapsto \mathsf{cata}(\mathsf{in} \circ F(f)_{\mu(F(Y))})
\end{aligned}$$

In general, it is not guaranteed that an initial algebra exists for any endofunctor $F : \mathcal{C} \to \mathcal{C}$. Typically, the existence of an initial algebras is shown by (infinitely) iterating F and showing that it converges, applying the classic theorem by Adámek [3]. This approach imposes some additional requirements on the functor F and underlying category $\mathcal{C}$, which we discuss in more detail in Section 4.3.

4.2.2 Function Types

The functor $\mathbf{exp}(-)$ is defined by mapping onto exponential objects in $\boldsymbol{\mathcal{C}}$. But we have to take some additional care to ensure that we can still define its action on morphism, as the polarity of free variables is reversed in domain of a function type. Indeed, when computed pointwise exponential objects give rise to a bifunctor of the form $\mathcal{C}^{\mathsf{op}} \to \mathcal{C} \to \mathcal{C}$, meaning that functors are not, in general, closed under exponentiation. To some extent we anticipated this situation already in the design of our type system by defining the well-formedness rule for function types such that the context of functorial variables, Φ, is discarded in its domain. Of course, the variables in Δ can occur both in covariant and contravariant positions, but by adopting a difunctorial semantics we limit ourselves to a specific class of functors that is closed under exponentiation. The key observation is that constructing the opposite category of the product of a category and its opposite is an idempotent (up to isomorphism) operation. That is, we have the following equivalence of categories: $(C^{\mathsf{op}} \times \mathcal{C})^{\mathsf{op}} \simeq C^{\mathsf{op}} \times \mathcal{C}$. As a result, a pointwise mapping of difunctors to exponential objects does give rise to a new difunctor. We use this fact to our advantage to define the following functor $\mathbf{exp}(F, G)$ for functors $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{E}$ and $G : (C^{\mathsf{op}} \times \mathcal{C}) \times \mathcal{D} \to \mathcal{E}$, of which the interpretation of function types is an instance.

$$
\begin{array}{lcl}
\mathbf{exp}(F,G)(-) & : & (\mathcal{C}^{\mathsf{op}} \times \mathcal{C}) \times \mathcal{D} \to \mathcal{E} \\
\mathbf{exp}(F,G)((x,y),z) & \mapsto & G((x,y),z)^{F(y,x)} \\
\mathbf{exp}(F,G)((f,g),h) & \mapsto & \mathsf{curry}(G((f,g),h) \circ \mathsf{eval} \circ (id_{\mathbf{exp}(F,G)((x,y),z)} \times F(g,f))
\end{array}
$$

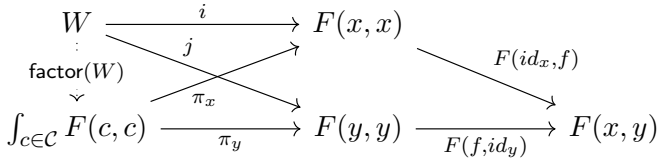
We remark that $\mathbf{exp}(\mathsf{F},\mathsf{G})$ does not define an exponential object in the functor category $[(\mathcal{C}^{\mathsf{op}} \times \mathcal{C}) \times \mathcal{D}, \mathcal{E}]$. Fortunately, for defining the semantics of term level λ-abstraction or application it is sufficient that the action on objects maps to exponentials in $\boldsymbol{\mathcal{C}}$.

4.2.3 Universal quantification

The semantics of universal quantifications is captured by ends in the category $\boldsymbol{\mathcal{D}}$. If $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ is a functor, then an *end* of F is an object $\int_{x \in \mathcal{C}} F(x,x) \in \mathcal{D}$ equipped with a projection map given by an extranatural transformation $\pi_x : \int_{c \in \mathcal{C}} F(c,c) \to F(x,x)$. Formally, and end of the F is defined as the universal wedge of the following diagram:

$$
F(x,x) \overset{F(id_x,f)}{\longrightarrow} F(x,y) \overset{F(f,id_y)}{\longleftarrow} F(y,y)
$$

For all $x, y \in \mathcal{C}$ and $f : x \to y$. The universal property of ends then states that any other wedge $W \in \mathcal{D}$ with maps $i : W \to F(x,x)$ and $j : W \to F(y,y)$ uniquely factors through $\int_{c \in \mathcal{C}} F(c,c)$.



To model the more general situation where a $\forall$-quantified type can contain free variables that are bound by another quantifier above it in the lexical hierarchy, we define the semantics of universal quantification in terms of the *end functor*, $\mathbf{end}(-)$, which for a functor $G : \mathcal{C} \to [\mathcal{D}^{\mathsf{op}} \times \mathcal{D}, \mathcal{E}]$ defines a functor $\mathbf{end}(G) : \mathcal{C} \to E$ whose object action is computed pointwise from ends in $\mathcal{E}$. Its action on morphisms, $\mathbf{end}(f) : \int_{d \in \mathcal{D}} G(X)(d,d) \to \int_{d \in \mathcal{D}} G(Y)(d,d)$, follows from the universal property stated above. To define the action on morphisms, we observe that the object $\int_{d \in \mathcal{D}} G(X)(d,d)$ is a wedge of the following diagram.

$$
G(Y)(x,x) \overset{G(Y)(id_x,f)}{\longrightarrow} G(Y)(x,y) \overset{G(Y)(f,id_y)}{\longleftarrow} G(Y)(y,y)
$$

Where the vertices of the cone are constructed by composing the projection map with the action of G on f, i.e., $G(f)(x,x) \circ \pi_x$. By universality, this wedge uniquely factors through the end $\int_{d \in \mathcal{D}} G(Y)(d,d)$. This factorization defines the morphism action $\mathbf{end}(\mathsf{f})$.

$$\begin{array}{lcl} \textbf{end}(G)(-) & : & \mathcal{C} \to \mathcal{E} \\ \textbf{end}(G)(x) & \mapsto & \int_{d \in \mathcal{D}} G(x)(d,d) \\ \textbf{end}(G)(f) & \mapsto & \mathsf{factor}(\int_{d \in \mathcal{D}} G(x)(d,d)) \end{array}$$

An important subtlety here is that $F(X)$ should have an end in $\mathcal{E}$ for every X. In our case, this is a consequence of completeness of $\boldsymbol{\mathcal{D}}$.[5] To actually use the functor **end** to define the semantics of universal quantifications, we need to precompose the semantics of its body with the **sift** functor to separate the quantified variable from the remainder of the context.

$$\textbf{sift} : (\llbracket\Delta\rrbracket \times \llbracket k\rrbracket)^{\mathsf{op}} \times (\llbracket\Delta\rrbracket \times \llbracket k\rrbracket)) \times \llbracket\Phi\rrbracket \to ((\llbracket\Delta\rrbracket^{\mathsf{op}} \times \llbracket\Delta\rrbracket) \times \llbracket\Phi\rrbracket) \times (\llbracket k\rrbracket^{\mathsf{op}} \times \llbracket k\rrbracket)$$

We note that **sift** defines an isomorphism in **CAT**.

4.3 On the Existence of Initial Algebras

In general, it is not the case that any endofunctor has an initial algebra. For certain classes of endofunctors, it can be shown that an initial algebra exists by means of Adámek's theorem [3]. Here, we present a condensed argument for why we expect that functors interpreting well-formed types of kind $k \rightsquigarrow k$ (for any k) have initial algebras; a more thorough formal treatment of the construction of initial algebras is a subject of further study.

The intuition behind Adámek's construction is that repeated applications of an endofunctor $F : \mathcal{C} \to \mathcal{C}$ converge after infinite iterations, reaching a fixpoint. If $\mathcal{C}$ has an initial object and *ω-colimits*,[6] we can define the initial algebra of F as the ω-colimit of the following chain:

$$\bot \xrightarrow{!} F\bot \xrightarrow{F!} FF\bot \xrightarrow{FF!} FFF\bot \xrightarrow{FFF!} \ldots$$

Where $\bot$ is the initial object in $\mathcal{C}$ and $!_X : \bot \to X$ the unique map from $\bot$ to X. A crucial stipulation is that F should be *ω-cocontinuous*, meaning that it preserves ω-colimits.

Thus, for the functors interpreting higher-order types to have an initial algebra, we must argue that all higher-order types are interpreted to a ω-cocontinuous functor. This prompts a refinement of the semantics for kinds discussed in Section 4.1, where we impose the additional restriction that the interpretation of a kind of the form $k_1 \rightsquigarrow k_2$ is a ω-cocontinuous functor from $\llbracket k_1\rrbracket$ to $\llbracket k_2\rrbracket$. Subsequently, we must show that Figure 5 actually inhabits this refined semantics.

Johann and Polonsky [16] present an inductive argument showing the existence of initial algebras for a universe of higher-kinded data types is similar to our definition of well-formed terms in Figure 2. While their proof establishes the more general property of λ-cocontinuity (for an arbitrary limit ordinal λ) for the functors interpreting higher-kinded types, we expect that the relevant cases of their inductive proof—specifically the cases for products, coproducts, type application, and the $\boldsymbol{\mu}$ functor—can be adapted to our setting. What remains is to show that the semantics of type level λ-abstraction and function types is a ω-cocontinuous functor. For λ-abstraction, we transport along the currying isomorphism, which should preserve ω-cocontinuity. For function types, we would need the additional assumption that the functor $(-)^X : \boldsymbol{\mathcal{C}} \to \boldsymbol{\mathcal{C}}$ is ω-cocontinuous for all X, which, as Johann and Polonsky [16] point out, at least holds for the category of sets.

4.4 Arrow Types Correspond to Morphisms

To define the semantics of well-typed terms, it is crucial that we can relate arrow types—i.e., of the form $\tau_1 \xrightarrow{k} \tau_2$—to morphisms in the category $\llbracket k\rrbracket$. To make this more precise, consider the typing rule for left projections. To define its semantics, we would like to use the cartesian structure of the category $\llbracket k\rrbracket$, which implies the existence of a *morphism* $\pi_1 : \llbracket k\rrbracket(x \times y, x)$ for $x, y \in \llbracket k\rrbracket$. However, the rule T-Fst implies that

[5] See Mac Lane [20] chapter 9.5 corollary 2.
[6] That is, colimits over diagrams defined as a functor on the thin category generated from the poset of natural numbers.

π_1 should be related to an *object* in the category $\mathcal{D}$, i.e., $[\![\tau_1 \otimes \tau_2 \xrightarrow{k} \tau_1]\!]$. To mediate between morphisms in $[\![k]\!]$ and objects in $\mathcal{D}$ calls for a suitable currying/uncurrying isomorphism for arrow types, though we highlight that the required isomorphism is different from the usual currying isomorphism arising from the existence of right adjoints for the tensor product in closed monoidal catetegories, in the sense that $[\![\tau_1 \xrightarrow{k} \tau_2]\!]$ does not define an internal hom for the objects $[\![\tau_1]\!], [\![\tau_2]\!]$ but rather inernalizes the morphisms between these objects in a *different* category.

Theorem 4.1 *Given a kind k, morphisms of the category $[\![k]\!]$ are internalized as objects in $\mathcal{D}$ through the following bijection between hom-sets:*

$$[\![k]\!](F(\boldsymbol{\delta}) \times [\![\tau_1]\!](\boldsymbol{\delta}^\circ), [\![\tau_2]\!](\boldsymbol{\delta})) \quad \simeq \quad \mathcal{D}(F(\boldsymbol{\delta}), [\![\tau_1 \xrightarrow{k} \tau_2]\!](\boldsymbol{\delta})) \tag{1}$$

Where $\boldsymbol{\delta} \in [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!]$ and $\boldsymbol{\delta}^\circ \in ([\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!])^{\mathsf{op}}$ its complement, which is defined by swapping the objects representing contravariant respectively covariant occurrences of the variables in Δ. Let $F : [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!] \to \mathcal{D}$ be a functor. In a slight abuse of notation, we also write $F(\boldsymbol{\delta})$ for the "lifting" of F to an object in the (functor) category $[\![k]\!]$ that ignores all the additional variables on which $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ depend.

Proof. *We compute the isomorphism as follows, where $k = k_1 \rightsquigarrow \cdots \rightsquigarrow k_n \rightsquigarrow \star$:*

$$\begin{aligned}
[\![k]\!](F(\boldsymbol{\delta}) \times [\![\tau_1]\!](\boldsymbol{\delta}^\circ), [\![\tau_2]\!](\boldsymbol{\delta})) &= \int_{x_1 \in [\![k_1]\!]} \cdots \int_{x_n \in [\![k_n]\!]} \mathcal{D}(F(\boldsymbol{\delta}) \times [\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n), [\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n)) \\
&\simeq \int_{x_1 \in [\![k_1]\!]} \cdots \int_{x_n \in [\![k_n]\!]} \mathcal{D}(F(\boldsymbol{\delta}), [\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n)^{[\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n)}) \\
&\simeq \mathcal{D}(F(\boldsymbol{\delta}), \int_{x_1} \cdots \int_{x_n} [\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n \in [\![k_n]\!])^{[\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n)}) \\
&\simeq \mathcal{D}(F(\boldsymbol{\delta}), [\![\tau_1 \xrightarrow{k} \tau_2]\!](\boldsymbol{\delta}))
\end{aligned}$$

The first step of the derivation rewrites the left-hand side of the isomorphism to a sequence of zero or more ends in the category of very large sets, allowing us to apply currying for exponentials in $\mathcal{D}$ in the subsequent step. This is justified by cartesian closedness of $\mathcal{C}$, because the objects $[\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n)$ and $[\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n)$ are included in the image of the fully faithful inclusion functor $I : \mathcal{C} \to \mathcal{D}$. Next, we use the fact that the covariant hom-functor $\mathcal{D}(x, -)$ is continuous and thus preserves ends:[7]

$$\int_{y \in \mathcal{C}} \mathcal{D}(x, G(y, y)) \quad \simeq \quad \mathcal{D}(x, \int_{y \in \mathcal{C}} G(y, y)) \tag{2}$$

By repeatedly applying the identity above, we can distribute the aforementioned sequence of ends over the functor $\mathcal{D}(F(\boldsymbol{\delta}), -)$. Intuitively, this corresponds to distributing universal quantification over logical implication in the scenario that the quantified variable does not occur freely in the anteceden, which is axiomatized in some flavors of first-order logic though we apply a much more general instance of the same principle here. The final step then follows from the standard definition of η-equivalence implied by the cartesian closed structure of **CAT**. □

We write $\uparrow(-)/\downarrow(-)$ for the functions that transport along the isomorphism defined in Equation (1).

4.5 Interpreting Terms

To define the semantics of well-typed terms, we specialize the semantics of types described in this section to the category of large and very large sets. A well-typed term $\Gamma \vdash M : \sigma$ is then interpreted as a natural transformation from the interpretation its context, $[\![\Gamma]\!]$, to the interpretation of its type, $[\![\sigma]\!]$. At

[7] See Mac Lane [20], page 225 Equation 4.

$$
\begin{aligned}
[\![\Gamma \vdash x : \sigma]\!]_{\boldsymbol{\delta}} &= \mathbf{lookup}^{\Gamma}_{x} \\
[\![\Gamma \vdash M\ N : \tau_2]\!]_{\boldsymbol{\delta}} &= \mathsf{eval} \circ \langle [\![\Gamma \vdash M : \tau_1 \Rightarrow \tau_2]\!]_{\boldsymbol{\delta}}, [\![\Gamma \vdash N : \tau_1]\!]_{\boldsymbol{\delta}} \rangle \\
[\![\Gamma \vdash \lambda x.M : \tau_1 \Rightarrow \tau_2]\!]_{\boldsymbol{\delta}} &= \mathsf{curry}([\![\Gamma, (x \mapsto \tau_1) \vdash M : \tau_2]\!]_{\boldsymbol{\delta}}) \\
[\![\Gamma \vdash \mathbf{let}\ (x : \sigma_1) = M\ \mathbf{in}\ N : \sigma_2]\!]_{\boldsymbol{\delta}} &= \mathsf{eval} \circ \langle \mathsf{curry}([\![\Gamma, (x \mapsto \sigma_1) \vdash N : \sigma_2]\!]_{\boldsymbol{\delta}}), [\![\Gamma \vdash N : \sigma_1]\!]_{\boldsymbol{\delta}} \rangle \\
[\![\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma]\!]_{\boldsymbol{\delta}} &= [\![\Gamma \vdash M : \sigma]\!]_{\boldsymbol{\delta}} \quad \text{(isomorphic per Equation (2))} \\
[\![\Gamma \vdash M@\tau : \sigma[\tau/\alpha]]\!]_{\boldsymbol{\delta}} &= \pi_{[\![\tau]\!]} \circ [\![\Gamma \vdash M : \forall\alpha.\sigma]\!]_{\boldsymbol{\delta}} \\
[\![\Gamma \vdash \mathbf{in} : \tau\ \mu(\tau) \xrightarrow{k} \mu(\tau)]\!]_{\boldsymbol{\delta}} &= \uparrow(\mathsf{in} \circ \pi_2) \\
[\![\Gamma \vdash \mathbf{out} : \mu(\tau) \xrightarrow{k} \tau\ \mu(\tau)]\!]_{\boldsymbol{\delta}} &= \uparrow(\mathsf{out} \circ \pi_2) \\
[\![\Gamma \vdash \mathbf{map}\langle M \rangle^{\tau} : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2]\!]_{\boldsymbol{\delta}} &= \uparrow(\lambda(\gamma, x).[\![\tau]\!](\boldsymbol{\delta})(\lambda y.\ \downarrow([\![\Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2]\!]_{\boldsymbol{\delta}})(\gamma, y))) \\
[\![\Gamma \vdash (\!|\ M\ |\!)^{\tau_1} : \mu(\tau_1) \xrightarrow{k} \tau_2]\!]_{\boldsymbol{\delta}} &= \uparrow(\lambda(\gamma, x).\mathsf{cata}(\lambda y.\ \downarrow([\![\Gamma \vdash M : \tau_1\ \tau_2 \xrightarrow{k} \tau_2]\!]_{\boldsymbol{\delta}})(\gamma, y))) \\
[\![\Gamma \vdash \boldsymbol{\pi_1} : \tau_1 \otimes \tau_2 \xrightarrow{k} \tau_1]\!]_{\boldsymbol{\delta}} &= \uparrow(\pi_1 \circ \pi_2) \\
[\![\Gamma \vdash \boldsymbol{\pi_2} : \tau_1 \otimes \tau_2 \xrightarrow{k} \tau_2]\!]_{\boldsymbol{\delta}} &= \uparrow(\pi_2 \circ \pi_2) \\
[\![\Gamma \vdash M \blacktriangle N : \tau \xrightarrow{k} \tau_1 \otimes \tau_2]\!]_{\boldsymbol{\delta}} &= \uparrow(\ \langle\ \downarrow([\![\Gamma \vdash M : \tau \xrightarrow{k} \tau_1]\!]_{\boldsymbol{\delta}})\ ,\ \downarrow([\![\Gamma \vdash N : \tau \xrightarrow{k} \tau_2]\!]_{\boldsymbol{\delta}})\ \rangle\ ) \\
[\![\Gamma \vdash \boldsymbol{\iota_1} : \tau_1 \xrightarrow{k} \tau_1 \oplus \tau_2]\!]_{\boldsymbol{\delta}} &= \uparrow(\iota_1 \circ \pi_2) \\
[\![\Gamma \vdash \boldsymbol{\iota_2} : \tau_2 \xrightarrow{k} \tau_1 \oplus \tau_2]\!]_{\boldsymbol{\delta}} &= \uparrow(\iota_2 \circ \pi_2) \\
[\![\Gamma \vdash M \blacktriangledown N : \tau_1 \oplus \tau_2 \xrightarrow{k} \tau]\!]_{\boldsymbol{\delta}} &= \uparrow(\ [\ \downarrow([\![\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau]\!]_{\boldsymbol{\delta}})\ ,\ \downarrow([\![\Gamma \vdash N : \tau_2 \xrightarrow{k} \tau]\!]_{\boldsymbol{\delta}})\ ]\ ) \\
[\![\Gamma \vdash \mathbf{tt} : \mathbb{1}]\!]_{\boldsymbol{\delta}} &= h \\
[\![\Gamma \vdash \mathbf{absurd} : \mathbb{0} \Rightarrow \tau]\!]_{\boldsymbol{\delta}} &= \mathsf{curry}(h \circ \pi_2)
\end{aligned}
$$

Fig. 6. Semantics of Well-Typed Terms. The gray highlights indicate the cases where we rely on working with sets.

component $\boldsymbol{\delta} \in [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!]$ this transformation is given by a function of the following type:

$$[\![\ \Gamma \vdash M : \sigma\ ]\!]_{\boldsymbol{\delta}} : [\![\ \Gamma\ ]\!](\boldsymbol{\delta}) \to [\![\ \sigma\ ]\!](\boldsymbol{\delta})$$

Here, $[\![\Gamma]\!]$ is defined componentwise by mapping contexts to a left-associated product of its elements, analogous to how we defined the interpretation of kind contexts in Section 4.1. Figure 6 shows the interpretation of well-typed terms in its entirety.

The interpretation of λ-abstraction and application is defined in terms of the cartesian closed structure of $\boldsymbol{\mathcal{C}}$, which is preserved by its inclusion in $\boldsymbol{\mathcal{D}}$. For a type abstractions of the form $\Lambda\alpha.M$, its semantics follows from the fact that hom-functors preserves ends (see Equation (2)), which implies a bijection between the set of morphisms that interprets the type abstraction and the set of morphisms into which we interpret its body. We remark that this only works because α does not occur free in Γ, meaning that we know that $[\![\Gamma]\!]$ does not depend on α in $[\![\Gamma \vdash M : \sigma]\!]_{\boldsymbol{\delta},(\alpha,\alpha)} : [\![\Gamma]\!](\boldsymbol{\delta}, (\alpha, \alpha)) \to [\![\sigma]\!](\boldsymbol{\delta}, (\alpha, \alpha))$, and thus we can view $[\![\Gamma]\!]$ as a constant when applying the isomorphism. The semantics of a type application $M\ @\tau$ is then given by the projection map at component $[\![\tau]\!]$ of the end interpreting the type of M. For the introduction and elimination forms of (co)product types, and the unit and empty type, we define the semantics in terms of the corresponding (co)limits in $\boldsymbol{\mathcal{D}}$, applying the currying isomorphism defined in Equation (1) to mediate with arrow types. Similarly, a semantics for the mapping and folding primitives also follows from the currying isomorphism defined in Equation (1), but in these cases we unfortunately have to be explicit about working in the category of sets to distribute the object interpreting Γ over the functions we are respectively mapping and folding with. As of yet, we do not know how to define these cases in a way that does not require us to be explicit about working in the category of sets.

5 Related Work

The problem of equipping functional languages with better support for modularity as been studied extensively in the literature. One of the earlier instances is the *Algebraic Design Language* (ADL) by Kieburtz and Lewis [18], which features language primitives for specifying computable functions in terms of algebras. ADL overlaps to a large extent with the first-order fragment of our calculus, but lacks support for defining nested data types. Zhang et al. [36] recently proposed a calculus and language for *compositional programming*, called CP. Their language design is inspired by *object algebras*, which in turn is based on the *tagless final* approach [8,19] and *final algebra semantics* [33], which, according to Wand [33, §7], is an extension of *initial algebra semantics*. These lines of work thus provide similar modularity as initial algebra semantics, but in a way that does not require *tagged values*. While the categorical foundations of Zhang et al.'s CP language seems to be an open question, the language provides flexible support for modular programming, in part due to its powerful notion of subtyping. We are not aware of attempts to model (higher-order) effects and handlers using CP. In contrast, our calculus is designed to have a clear categorical semantics. This semantics makes it straightforward to define state of the art type safe modular (higher-order) effects and handlers. Morris and McKinna [24] define a language that has built-in support for *row types*, which supports both extensible records and variants. While their language captures many known flavors of extensibility, due to parameterizing the type system over a so-called *row theory* describing how row types behave under composition, rows are restricted to first order types. Consequently, they cannot describe any modularity that hinges on the composition of (higher-order) signature functors.

The question of including nested data types in a language's support for modularity has received some attention as well. For example, Cai et al. [7] develop an extension of F_ω with equirecursive types tailored to describe patterns from datatype generic programming. Their calculus is expressive enough to capture the modularity abstractions discussed in this paper, including those requiring nested data types, but lacks a denotational model; a correspondence between a subset of types in their calculus and (traversable) functors is discussed informally. Similarly, Abel et al. [2] consider an operational perspective of traversals over nested datatypes by studying several extensions of F_ω with primitives for *(generalized) Mendler iteration and coiteration*. Although these are expressive enough to describe modular higher-order effects and handlers, their semantic foundation is very different from the semantics of the primitive fold operation in our calculus. It is future work to investigate how our calculus can be extended with support for codata.

A major source of inspiration for the work in this paper are recent works by Johann and Polonsky [16], Johann et al. [15], and Johann and Ghiorzi [14], which respectively study the semantics and parametricity of nested data types and GADTs. For the latter, the authors develop a dedicated calculus with a design and semantics that is very similar to ours. Still, there are some subtle but key differences between the designs; for example, their calculus does not include general notions of $\forall$-types and function types, but rather integrates these into a single type representing natural transformations between type constructors. While their setup does not require the same stratification of the type syntax we adopt here, it is also slightly less expressive, as the built-in type of transformations is restricted to closing over 0-arity arguments.

Data type generic programming commonly relies on a *universe of descriptions* [4], which is a data type whose inhabitants correspond to a signature functor. Generic functions are commonly defined by induction over these descriptions, ranging over a semantic reflection of the input description in the type system of a dependently-typed host language [11]. In fact, Chapman et al. [9] considered the integration of descriptions in a language's design by developing a type theory with native support for generic programming. We are, however, not aware of any notion of descriptions that corresponds to our syntax of well-formed types.

6 Conclusion and Future work

In this paper, we presented the design and semantics of a calculus with support for modularity. We demonstrated how this calculus is a minimal basis for capturing several well-known programming patterns for retrofitting type-safe modularity to functional languages, such as modular interpreters in the style of Data Types à la Carte, and modular (higher-order) algebraic effects. The formal semantics associates these patterns with their motivating concepts, creating the possibility for a compiler to benefit from their properties such as by performing fusion-based optimizations. In future work, we hope to extend our calculus with generalized folds [6,21] to allow the definition of a broader class of recursive functions.

References

[1] Abbott, M. G., T. Altenkirch and N. Ghani, *Containers: Constructing strictly positive types*, Theor. Comput. Sci. **342** (2005), pp. 3–27.
URL https://doi.org/10.1016/j.tcs.2005.06.002

[2] Abel, A., R. Matthes and T. Uustalu, *Iteration and coiteration schemes for higher-order and nested datatypes*, Theor. Comput. Sci. **333** (2005), pp. 3–66.
URL https://doi.org/10.1016/j.tcs.2004.10.017

[3] Adámek, J., *Free algebras and automata realizations in the language of categories*, Commentationes Mathematicae Universitatis Carolinae **15** (1974), pp. 589–602.

[4] Benke, M., P. Dybjer and P. Jansson, *Universes for generic programs and proofs in dependent type theory*, Nord. J. Comput. **10** (2003), pp. 265–289.

[5] Bird, R. S. and L. G. L. T. Meertens, *Nested datatypes*, in: J. Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, Lecture Notes in Computer Science **1422** (1998), pp. 52–67.
URL https://doi.org/10.1007/BFb0054285

[6] Bird, R. S. and R. Paterson, *Generalised folds for nested datatypes*, Formal Aspects Comput. **11** (1999), pp. 200–222.
URL https://doi.org/10.1007/s001650050047

[7] Cai, Y., P. G. Giarrusso and K. Ostermann, *System f-omega with equirecursive types for datatype-generic programming*, in: R. Bodík and R. 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* (2016), pp. 30–43.
URL https://doi.org/10.1145/2837614.2837660

[8] Carette, J., O. Kiselyov and C. Shan, *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages*, J. Funct. Program. **19** (2009), pp. 509–543.
URL https://doi.org/10.1017/S0956796809007205

[9] Chapman, J., P. Dagand, C. McBride and P. Morris, *The gentle art of levitation*, in: P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010* (2010), pp. 3–14.
URL https://doi.org/10.1145/1863543.1863547

[10] Coquand, T. and C. Paulin, *Inductively defined types*, in: P. Martin-Löf and G. Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, Lecture Notes in Computer Science **417** (1988), pp. 50–66.
URL https://doi.org/10.1007/3-540-52335-9_47

[11] Dagand, P., "A cosmology of datatypes : reusability and dependent types," Ph.D. thesis, University of Strathclyde, Glasgow, UK (2013).
URL http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713

[12] Glimming, J. and N. Ghani, *Difunctorial semantics of object calculus*, in: V. Bono, M. Bugliesi and S. Drossopoulou, editors, *Proceedings of the Second Workshop on Object Oriented Developments, WOOD 2004, London, UK, August 30, 2004*, Electronic Notes in Theoretical Computer Science **138** (2004), pp. 79–94.
URL https://doi.org/10.1016/j.entcs.2005.09.012

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

[14] Johann, P. and E. Ghiorzi, *Parametricity for nested types and gadts*, Log. Methods Comput. Sci. **17** (2021).
URL https://doi.org/10.46298/lmcs-17(4:23)2021

[15] Johann, P., E. Ghiorzi and D. Jeffries, *Parametricity for primitive nested types*, in: S. Kiefer and C. 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*, Lecture Notes in Computer Science **12650** (2021), pp. 324–343.
URL https://doi.org/10.1007/978-3-030-71995-1_17

[16] Johann, P. and A. Polonsky, *Higher-kinded data types: Syntax and semantics*, in: *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019* (2019), pp. 1–13.
URL https://doi.org/10.1109/LICS.2019.8785657

[17] Kammar, O., S. Lindley and N. Oury, *Handlers in action*, in: G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013* (2013), pp. 145–158.
URL https://doi.org/10.1145/2500365.2500590

[18] Kieburtz, R. B. and J. Lewis, *Programming with algebras*, in: J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, Lecture Notes in Computer Science **925** (1995), pp. 267–307.
URL https://doi.org/10.1007/3-540-59451-5_8

[19] Kiselyov, O., *Typed tagless final interpreters*, in: J. Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Lecture Notes in Computer Science **7470** (2010), pp. 130–174.
URL https://doi.org/10.1007/978-3-642-32202-0_3

[20] MacLane, S., "Categories for the Working Mathematician," Springer-Verlag, New York, 1971, ix+262 pp., graduate Texts in Mathematics, Vol. 5.

[21] Martin, C. E., J. Gibbons and I. Bayley, *Disciplined, efficient, generalised folds for nested datatypes*, Formal Aspects Comput. **16** (2004), pp. 19–35.
URL https://doi.org/10.1007/s00165-003-0013-6

[22] Meijer, E., M. M. Fokkinga and R. Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire*, in: J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, Lecture Notes in Computer Science **523** (1991), pp. 124–144.
URL https://doi.org/10.1007/3540543961_7

[23] Moggi, E., *Notions of computation and monads*, Inf. Comput. **93** (1991), pp. 55–92.
URL https://doi.org/10.1016/0890-5401(91)90052-4

[24] Morris, J. G. and J. McKinna, *Abstracting extensible data types: or, rows by any other name*, Proc. ACM Program. Lang. **3** (2019), pp. 12:1–12:28.
URL https://doi.org/10.1145/3290325

[25] Plotkin, G. D. and M. Pretnar, *Handlers of algebraic effects*, in: G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, Lecture Notes in Computer Science **5502** (2009), pp. 80–94.
URL https://doi.org/10.1007/978-3-642-00590-9_7

[26] Poulsen, C. B. and C. van der Rest, *Hefty algebras: Modular elaboration of higher-order algebraic effects*, Proc. ACM Program. Lang. **7** (2023), pp. 1801–1831.
URL https://doi.org/10.1145/3571255

[27] Schrijvers, T., M. Piróg, N. Wu and M. Jaskelioff, *Monad transformers and modular algebraic effects: what binds them together*, in: R. A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019* (2019), pp. 98–113.
URL https://doi.org/10.1145/3331545.3342595

[28] Swierstra, W., *Data types à la carte*, J. Funct. Program. **18** (2008), pp. 423–436.
URL https://doi.org/10.1017/S0956796808006758

[29] Uustalu, T. and V. Vene, *Mendler-style inductive types, categorically*, Nord. J. Comput. **6** (1999), p. 343.

[30] van den Berg, B., T. Schrijvers, C. B. Poulsen and N. Wu, *Latent effects for reusable language components*, in: H. Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, Lecture Notes in Computer Science **13008** (2021), pp. 182–201.
URL https://doi.org/10.1007/978-3-030-89051-3_11

[31] van der Rest, C. and C. B. Poulsen, *Towards a language for defining reusable programming language components - (project paper)*, in: W. Swierstra and N. Wu, editors, *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*, Lecture Notes in Computer Science **13401** (2022), pp. 18–38.
URL https://doi.org/10.1007/978-3-031-21314-4_2

[32] Wadler, P., *The expression problem*, http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (1998), accessed: 2020-07-01.

[33] Wand, M., *Final algebra semantics and data type extensions*, J. Comput. Syst. Sci. **19** (1979), pp. 27–44.
URL https://doi.org/10.1016/0022-0000(79)90011-4

[34] Wu, N., T. Schrijvers and R. Hinze, *Effect handlers in scope*, in: W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014* (2014), pp. 1–12.
URL https://doi.org/10.1145/2633357.2633358

[35] Yang, Z., M. Paviotti, N. Wu, B. van den Berg and T. Schrijvers, *Structured handling of scoped effects*, in: I. Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, Lecture Notes in Computer Science **13240** (2022), pp. 462–491.
URL https://doi.org/10.1007/978-3-030-99336-8_17

[36] Zhang, W., Y. Sun and B. C. d. S. Oliveira, *Compositional programming*, ACM Trans. Program. Lang. Syst. **43** (2021), pp. 9:1–9:61.
URL https://doi.org/10.1145/3460228