Types and Semantics for Extensible Data Types (Extended Version)

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

[1]Delft University of Technology, Delft, The Netherlands
{c.r.vanderrest,c.b.poulsen}@tudelft.nl

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. A theoretically well-studied solution is to define data types and functions using *initial algebra semantics*. While it is possible to encode this solution in existing programming languages, such encodings add syntactic and 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. A solution to these is to provide native support for initial algebra semantics. In this paper, we develop such a solution and present a type discipline and core calculus for a language with native support for initial algebra semantics.

Keywords: Type systems · Modularity · Programming 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 [37] 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* [28]. 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* [18] 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 [35] 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 [23] and higher order [39, 31, 7] effectful computations.

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 defines 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 higher degree of type safe modularity, the approach has some downsides:

- Encodings of a data type's initial algebra semantics lacks the syntactic convenience of native data types, especially when it comes to constructing and pattern matching on values. Further overhead is introduced by their limited interoperability, which is typically relies on user-defined isomorphisms.
- The connection between initial algebra semantics encodings of data types, and the mathematical concepts that motivate them remains implicit. This has two drawbacks: (1) the programmer has to write additional code witnessing that their definitions possess the required structure (e.g., by defining instances of the `Functor` typeclass), and (2) a compiler cannot leverage the properties of this structure, such as by implementing (provably correct) optimizations based on the well-known map and fold fusion laws.

In this paper, we explore an alternative perspective by making type-safe modularity part of the language's design, by including built-in primitives for the functional programmer's modularity toolkit—e.g., functors, folds, fixpoints, etc. We believe that this approach has the potential 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 [32]). Furthermore, by supporting type-safe modularity through dedicated language primitives, we open the door for compilers to benefit from their properties, for example by applying fusion based optimizations.

1.1 Contributions

The semantics of (nested) algebraic data types has been studied extensively in the literature (e.g., by Johann et al. [21, 22, 20], and Abel et al. [2–4]) resulting in the development of various calculi with the purpose of studying different aspects of the semantics of programming with algebraic data types. In this paper, we build on these works to develop a core calculus that seeks to distill the essential language features needed for developing programming languages with built-in support for type-safe modularity while retaining the same formal foundations. Although the semantic ideas that we build on to develop our calculus are generally well-known, their application to improving the design of functional programming languages has yet to be explored in depth. It is still future work to leverage the insights gained by developing this calculus in the design of programming language that provide better ergonomics for working with extensible data types, but we believe the development of a core calculus capturing the essentials of programming with extensible data types to be a key step for achieving this goal. To bridge from the calculus presented in this paper to a practical language design, features such as *smart constructors*, *row types*, and *(functor) subtyping* (as employed, for example, by Morris and McKinna [29] and Hubers and Morris [19]) would be essential. We make the following technical contributions:

- We show (in Section 2) how 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 can be captured in the calculus.
- We present (in Section 3) a formal definition of the syntax and type system.
- We give (in Section 4) a categorical semantics for our calculus.
- We present (in Section 5) an operational semantics for our calculus, and discuss how it relates to the categorical semantics.

Section 6 discusses related work, and Section 7 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 $(const : \forall\alpha.\forall\beta.\alpha \Rightarrow \beta \Rightarrow \alpha) = \lambda x.\lambda y.x$. Types are closed under products and coproducts, with the 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 $List \triangleq \lambda\alpha.\mu(\lambda X.\mathbb{1}+(\alpha\times X))$. 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 to 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* [8], 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 [40] and modular elaborations for higher-order effects [31], as in both cases effect trees that represents monadic programs with higher-order operations is defined as a nested data type.

Notation. All code examples in this section directly correspond to programs in our calculus, but we take some notational liberty to simplify the exposition. Abstraction and application of type variables is left implicit. Similarly, we omit first-order universal quantifications. By convention, we denote type variables bound by type-level λ-abstraction using capital letters (e.g., X), and those bound by universal quantification using Greek letters (e.g., α,β).

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:

$$e ::= \mathbb{N} \mid e + e \qquad\qquad \mathit{Expr} \triangleq \lambda X.\mathbb{N} + (X \times X)$$

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

$$\begin{aligned} &\mathit{expr} : \mathbb{N} + (\mathbb{N} \times \mathbb{N}) \Rightarrow \mathbb{N} && \mathit{eval} : \mu(\mathit{Expr}) \Rightarrow \mathbb{N} \\ &\mathit{expr} = (\lambda x.x) \blacktriangledown (\lambda x.\boldsymbol{\pi_1}\ x + \boldsymbol{\pi_2}\ x) && \mathit{eval} = (\!|\ \mathit{expr}\ |\!)^{\mathit{Expr}} \end{aligned}$$

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. [27], 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 $\mathit{Mul} \triangleq \lambda X.X \times X$:

$$\begin{aligned} &\mathit{mul} : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} && \mathit{eval} : \mu(\mathit{Expr} + \mathit{Mul}) \Rightarrow \mathbb{N} \\ &\mathit{mul} = \lambda x.\boldsymbol{\pi_1}\ x * \boldsymbol{\pi_2}\ x && \mathit{eval} = (\!|\ \mathit{expr} \blacktriangledown \mathit{mul}\ |\!)^{\mathit{Expr}+\mathit{Mul}} \end{aligned}$$

2.2 Modular Algebraic Effects using the Free Monad

As our second example we consider how to define modular algebraic effects and handlers [30] in terms of the free monad following Swierstra [35]. 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.

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

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.

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

Here, we use higher order instances at kind $\star \rightsquigarrow \star$ of the coproduct eliminator $- \blacktriangledown -$, the coproduct injection functions ι_1, ι_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:

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

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:

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

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.

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

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. [40], 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.

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

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 [30] original strategy for capturing higher-order operations. In recent work, Bach Poulsen and Van der Rest [31] demonstrated how such abbreviations can be made modular and reusable 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.

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

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. [39]:

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

$$\alpha, \beta, \gamma, X, Y \;\in\; \text{String}$$

$$\begin{array}{rcl}
\mathit{Kind} & \ni & k ::= \star \mid k \rightsquigarrow k \\
\mathit{KindEnv} & \ni & \Delta, \Phi ::= \emptyset \mid \Delta, \alpha : k \\
\\
\mathit{Type} & \ni & \tau ::= \alpha \mid X \mid \tau\ \tau \mid \lambda X.\tau \mid \mu(\tau) \mid \tau \Rightarrow \tau \\
 & & \quad\mid\ \mathbb{0} \mid \mathbb{1} \mid \tau \times \tau \mid \tau + \tau \\
\mathit{Scheme} & \ni & \sigma ::= \forall \alpha.\sigma \mid \tau
\end{array}$$

Fig. 1. Type syntax

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

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

That is, we show that *Prog h* is a *modular carrier* for *Maybe* [34].

As demonstrated, our calculus supports defining higher-order effects and their interpretations. To conveniently sequence higher-order computations we typically also want to use a monadic bind function, such as $\gg\!= : \mathit{Prog}\ h\ \alpha \rightarrow (\alpha \rightarrow \mathit{Prog}\ h\ \beta) \rightarrow \mathit{Prog}\ h\ \beta$. While it is possible to define monadic bind for *Free* from Section 2.2 in terms of a plain fold, defining the monadic bind for *Prog* generally requires a *generalized fold* [9, 40]. Adding this and other recursion principles [27] to our calculus is future work.

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. In this section we present a core 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 limited to prenex normal form à la Hindley-Milner. Additionally, the syntax of types, defined in Figure 1, includes primitives for constructing recursive types ($\mu(-)$), products ($\times$) and coproducts ($+$), as well as a unit type ($\mathbb{1}$) and empty type ($\mathbb{0}$). In the definition of the syntax of types, the use of $\forall$-types is restricted by stratifying the syntax into two layers, types and type schemes. Consequently, our calculus is, by design, *predicative*: $\forall$-types can quantify over types but not type schemes.

The motivation for this predicative design is that it permits a relatively straightforward categorical interpretation of $\forall$-types in terms of *ends* (see Section 4.2). Whereas the restriction of universal quantification to prenex normal form is usually imposed to facilitate type inference, our calculus does not support inference in its current form due to the structural treatment of data types.

In a structural setting, inference requires the reconstruction of (recursive) data type definitions from values, which is, in general, not possible.

We remark that the current presentation of the type system is *declarative*, meaning certain algorithmic aspects crucial for type checking, such as normalization and equality checking of types, are not covered in the current exposition. Regarding decidability of the type system: our system is a subset of System F_ω, whose Church-style formulation is decidable while its Curry-style formulation is not. As such, we expect our type system to inherit these properties. Since we are restricting ourselves to a predicative subset of F_ω, we are optimistic that the Curry-style formulation of our type system will be decidable too, but verifying this expectation is future work.

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

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

$$\textsc{K-Abs}\ \frac{\Delta \mid \Phi, (X \mapsto k_1) \vdash \tau : k_2}{\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2} \qquad \textsc{K-Fix}\ \frac{\Delta \mid \Phi \vdash \tau : k \rightsquigarrow k}{\Delta \mid \Phi \vdash \mu(\tau) : k} \qquad \textsc{K-Fun}\ \frac{\Delta \mid \emptyset \vdash \tau_1 : \star \qquad \Delta \mid \Phi \vdash \tau_2 : \star}{\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star}$$

$$\textsc{K-Empty}\ \frac{}{\Delta \mid \Phi \vdash \mathbb{0} : k} \qquad \textsc{K-Unit}\ \frac{}{\Delta \mid \Phi \vdash \mathbb{1} : k} \qquad \textsc{K-Product}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k \qquad \Delta \mid \Phi \vdash \tau_2 : k}{\Delta \mid \Phi \vdash \tau_1 \times \tau_2 : k}$$

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

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

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

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

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. Well-formedness of types is defined using the judgment $\Delta \mid \Phi \vdash \tau : k$, stating that the type τ has kind k under contexts Δ and Φ. Similarly, well-formedness of type schemes is defined by the judgment $\Delta \vdash \sigma$, stating that the type scheme σ is well-formed with respect to the context Δ.

Following Johann et al. [21], well-formedness of types is defined with respect to two contexts, one containing functorial variables (Φ), and one containing variables with mixed variance (Δ). Specifically, the variables in the context Φ are restricted to occur only in *strictly positive* [1, 13] 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. While it may seem overly restrictive to require type expressions to be strictly positive—rather than merely positive—in Φ, this is necessary to ensure that μ-types, as well as its introduction and elimination forms, have a well-defined semantics (see Section 4.2). Variables in Φ are bound by type-level λ-abstraction, meaning that any type former with kind $k_1 \rightsquigarrow k_2$ is functorial in its argument. In contrast, the variables in Δ are bound by $\forall$-quantification.

Products ($\times$), coproducts ($+$), units ($\mathbb{1}$) and empty types ($\mathbb{0}$) can be constructed at any kind, reflecting the fact that the corresponding categorical (co)limits can be lifted from Set to its functor categories by computing them pointwise. This pointwise lifting of these (co)limits to functor categories is reflected in the β equalities for these type formers (shown in Figure 5), which allow an instance at kind $k_1 \rightsquigarrow k_2$, when applied with a type argument, to be replaced with an instance at kind k_2.

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 approach 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 scheme. Section 3.3 discusses type equivalence in more detail.

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**/**unin**), 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*. In spirit of the syntactic notion of natural transformations used by Abel et al. [2–4] to study generalized (Mendler) iteration, 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. Arrow types are defined by induction over k, since the precise meaning of morphism for any pair of types 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,

$$x, y \in \text{String}$$

$$\begin{array}{lrl} \mathit{Env} & \ni & \Gamma ::= \emptyset \mid \Gamma, x : \sigma \\ \mathit{Term} & \ni M, N ::= & x \mid M\ N \mid \lambda x.M \mid \mathbf{let}\ (x : \sigma) = M\ \mathbf{in}\ N \\ & \mid & \Lambda\alpha.M \mid M\ @\tau \mid \mathbf{in} \mid \mathbf{unin} \mid \mathbf{map}\langle M \rangle^{\tau} \mid (\!|\ M\ |\!)^{\tau} \\ & \mid & \boldsymbol{\pi_1} \mid \boldsymbol{\pi_2} \mid M \blacktriangle N \mid \boldsymbol{\iota_1} \mid \boldsymbol{\iota_2} \mid M \blacktriangledown N \mid \mathbf{tt} \mid \mathbf{absurd} \end{array}$$

$$\tau_1 \xrightarrow{\star} \tau_2 \triangleq \tau_1 \Rightarrow \tau_2 \qquad \text{(Arrow Types)}$$
$$\tau_1 \xrightarrow{(k_1 \rightsquigarrow k_2)} \tau_2 \triangleq \forall\alpha.\ \tau_1\ \alpha \xrightarrow{k_2} \tau_2\ \alpha$$
$$\textit{where} \quad \Delta \vdash \tau_1 \xrightarrow{k} \tau_2 \quad \textit{if} \quad \Delta \mid \emptyset \vdash \tau_1, \tau_2 : k$$

Fig. 3. Term syntax

they are to be interpreted as objects in a suitable functor category, meaning that their morphisms are natural transformations. 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 intuition that polymorphic functions cor respond to natural transformations.[1] For instance, we would type the inorder traversal of binary trees as $\mathit{inorder} : \mathit{Tree} \xrightarrow{\star \rightsquigarrow \star} \mathit{List}$ ($\triangleq \forall\alpha.\mathit{Tree}\ \alpha \Rightarrow \mathit{List}\ \alpha$), describing a natural transformation between the *Tree* and *List* functors.

The typing rules are shown in shown in Figure 4. The rules rely on arrow types for introduction and elimination forms. For example, Products can be constructed at any kind (following rule K-Product in Figure 2), so the rules for terms that operate on these (i.e., T-Fst, T-Snd, and T-Fork) use arrow types at any kind k. Consequently, 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 as morphisms in this category.

3.3 Type Equivalence

In the presence of type level λ-abstraction and application, the same type can have multiple representations. For this reason, the type system defined in Figure 4 includes a non-syntactic conversion rule that allows a well-typed term to be re-typed under any equivalent type scheme. The relevant equational theory for types is defined in Figure 5, and includes the customary β and η equivalences for λ-terms, as well as β rules for product, sum, unit, and empty types. The equations shown in Figure 5 are motivated by the semantic model we discuss in Section 4, in the sense that equivalent types are interpreted to naturally isomorphic functors. The relation is also reflexive and transitive, motivated by respectively the identity and composition of natural isomorphisms. Viewing the equalities in Figure 5 left-to-right provides us with a basis for a normalization strategy for types, which would be required for implementing the type system.

[1] This intuition is made formal by Theorem 1 in Section 4.4.

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

$$\text{T-VAR}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{T-APP}\ \frac{\Gamma \vdash M : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M N : \tau_2} \qquad \text{T-ABS}\ \frac{\Gamma, (x : \tau_1) \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \Rightarrow \tau_2}$$

$$\text{T-LET}\ \frac{\Gamma \vdash M : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash N : \sigma_2}{\Gamma \vdash \textbf{let}\ (x : \sigma_1) = M\ \textbf{in}\ N : \sigma_2} \qquad \text{T-TYPEABS}\ \frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathsf{freevars}(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma}$$

$$\text{T-TYPEAPP}\ \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\ @\tau : \sigma[\tau/\alpha]} \qquad \text{T-IN}\ \frac{}{\Gamma \vdash \textbf{in} : \tau\ \mu(\tau) \xrightarrow{k} \mu(\tau)} \qquad \text{T-OUT}\ \frac{}{\Gamma \vdash \textbf{unin} : \mu(\tau) \xrightarrow{k} \tau\ \mu(\tau)}$$

$$\text{T-MAP}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2}{\Gamma \vdash \textbf{map}\langle M\rangle^{\tau} : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2} \qquad \text{T-FOLD}\ \frac{\Gamma \vdash M : \tau_1\ \tau_2 \xrightarrow{k} \tau_2}{\Gamma \vdash (\!|\ M\ |\!)^{\tau_1} : \mu(\tau_1) \xrightarrow{k} \tau_2}$$

$$\text{T-FST}\ \frac{}{\Gamma \vdash \boldsymbol{\pi_1} : \tau_1 \times \tau_2 \xrightarrow{k} \tau_1} \qquad \text{T-SND}\ \frac{}{\Gamma \vdash \boldsymbol{\pi_2} : \tau_1 \times \tau_2 \xrightarrow{k} \tau_2}$$

$$\text{T-FORK}\ \frac{\Gamma \vdash M : \tau \xrightarrow{k} \tau_1 \qquad \Gamma \vdash N : \tau \xrightarrow{k} \tau_2}{\Gamma \vdash M \blacktriangle N : \tau \xrightarrow{k} \tau_1 \times \tau_2} \qquad \text{T-INL}\ \frac{}{\Gamma \vdash \boldsymbol{\iota_1} : \tau_1 \xrightarrow{k} \tau_1 + \tau_2}$$

$$\text{T-INR}\ \frac{}{\Gamma \vdash \boldsymbol{\iota_2} : \tau_2 \xrightarrow{k} \tau_1 + \tau_2} \qquad \text{T-JOIN}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau \qquad \Gamma \vdash M : \tau_2 \xrightarrow{k} \tau}{\Gamma \vdash M \blacktriangledown N : \tau_1 + \tau_2 \xrightarrow{k} \tau} \qquad \text{T-UNIT}\ \frac{}{\Gamma \vdash \textbf{tt} : \mathbb{1}}$$

$$\text{T-EMPTY}\ \frac{}{\Gamma \vdash \textbf{absurd} : \mathbb{0} \xrightarrow{k} \tau} \qquad \text{T-CONV}\ \frac{\Gamma \vdash M : \sigma_1 \qquad \sigma_1 \equiv \sigma_2}{\Gamma \vdash M : \sigma_2}$$

Fig. 4. Well-formed terms

4 Categorical Semantics

In this section, we consider how to define a categorical semantics for our calculus, drawing inspiration from the semantics defined by Johann and Polonsky [22] and Johann et al. [21, 20]. To define this semantics, we must show that each type in our calculus corresponds to a functor, and that all such functors have initial algebras. In Section 4.3 we discuss the requirements for these initial algebras to exist, and argue informally why they should exist for the functors interpreting our types. Although Johann and Polonsky [22] present a detailed argument for

$$(\lambda X.\tau_1)\ \tau_2 \equiv \tau_1[\tau_2/X]$$
$$(\lambda X.\tau\ X) \equiv \tau$$
$$(\tau_1 \times \tau_2)\ \tau \equiv (\tau_1\ \tau) \times (\tau_2\ \tau)$$
$$(\tau_1 + \tau_2)\ \tau \equiv (\tau_1\ \tau) + (\tau_2\ \tau)$$
$$\mathbb{1}\ \tau \equiv \mathbb{1}$$
$$\mathbb{0}\ \tau \equiv \mathbb{0}$$

$$T := [] \mid T\ \tau \mid \tau\ T \mid \mu(T) \mid T \Rightarrow \tau \mid \tau \Rightarrow T$$
$$\mid\ T \times \tau \mid \tau \times T \mid T + \tau \mid \tau + T$$

$$\frac{\tau_1 \equiv \tau_2}{T[\ \tau_1\ ] \equiv T[\ \tau_2\ ]}$$

Fig. 5. Equational theory for types

the existence of initial algebras of the functors underlying nested data types, it is still future work to adapt this argument to our setting.

The general setup of our semantics is to interpret types of kind $\star$ as objects in Set (the category of sets), higher-order types as functors on Set, and type schemes as objects in Set_2 (the category of large sets). This size bump is necessary to model the universal quantification over types in type schemes. Crucially, Set is a *full subcategory* of Set_1, as witnessed by the existence of a fully faithful inclusion functor I:

$$\text{Set} \overset{I}{\hookrightarrow} \text{Set}_1$$

Assuming cumulative universes (i.e., the collection of all large sets also includes all small sets), I is just the identity functor. We remark that both Set and Set_1 are *complete* and *cocomplete* and *cartesian closed*. Importantly, since I is fully faithful, the cartesian closed structure of Set is reflected in Set_1 for those objects that lie in the image of I.

The subcategory relation between Set and Set_1 reflects the syntactic restriction of types to rank-1 polymorphism: all objects in Set can also be found in Set_1, but Set_1 is sufficiently larger than Set that it also includes objects modelling quantification over objects in Set. This intuition is embodied by fact that every functor $F : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \text{Set}_1$, where $\mathcal{C}$ is smaller than Set_1 (which includes Set), has an end in Set_1. This follows from completeness of Set_1 [26, p. 224, corollary 2]. We discuss the use of ends for modelling universal quantification in more detail in Section 4.2.

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 Set, and kinds of the form $k_1 \rightsquigarrow k_2$ to the category of functors between their domain and codomain.[2]

$$[\![-]\!]\ :\ \mathit{Kind} \to \mathbf{CAT}$$
$$[\![\ \star\ ]\!] = \text{Set}$$
$$[\![\ k_1 \rightsquigarrow k_2\ ]\!] = [\ [\![\ k_1\ ]\!]\ ,\ [\![\ k_2\ ]\!]\ ]$$

[2] Here, **CAT** denotes the (very large) category of large categories. Although Set itself is locally small, its functor categories have a large set of morphisms.

By interpreting types of kind $k_1 \rightsquigarrow k_2$ as objects in a functor category, we formalize the intuition that higher-order types correspond to functors. The semantics of kind contexts is then defined on a per-entry basis, as a chain of products of 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*, which has a single object, $*$, together with its identity morphism, id_*. It is worth mentioning that $\bullet$ and $- \times -$, together with the operation of constructing a functor category, $[-,-]$, imply that $\mathbf{CAT}$ is a cartesian closed category. We will use this cartesian closed structure to give a semantics to 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 [36], or the object calculus semantics by Glimming and Ghani [17]). Well-formed types (left) and type schemes (right) are interpreted as a functors over their contexts of the following form:

$$[\![\Delta \mid \Phi \vdash \tau : k]\!] : ([\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!]) \times [\![\Phi]\!] \to [\![k]\!] \qquad [\![\Delta \vdash \sigma]\!] : [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!] \to \textsc{Set}_1$$

Ultimately, the goal of this setup is to interpret $\forall$-types as *ends* in $\textsc{Set}_1$, 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, and give a more precise account of the relation between arrow types and natural transformations in Section 4.4.

Figure 6 defines the semantics of well-formed types and type schemes. The interpretation of the empty type, unit type, and (co)product types follow immediately from (co)completeness of $\textsc{Set}$. Since they can be constructed at any kind, the semantics of (co)product types depends crucially on the fact that functor

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

$$
\begin{aligned}
[\![\Delta \vdash \forall \alpha.\sigma]\!] &= \mathsf{\textbf{end}}(\mathsf{curry}([\![\Delta, \alpha : k \vdash \sigma]\!] \circ \mathsf{\textbf{sift}})) \\
[\![\Delta \vdash \tau]\!] &= I \circ [\![\Delta \mid \emptyset \vdash \tau : \star]\!]
\end{aligned}
$$

Fig. 6. Semantics of well-formed types and type schemes

categories preserve all (co)limits of their codomain category, which implies 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.

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

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 specifically-defined functors. Although their definitions are typical examples of how (co)limits are lifted to functor categories by computing them pointwise, we discuss the definition of these functors separately and in more detail respectively in Section 4.2 (recursive types), Section 4.2 (function types), and Section 4.2 ($\forall$-types).

Recursive Types Following the usual categorical interpretation of inductive data types [18], the semantics of recursive types is given by an *initial algebras*. We summarize the 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 $F(X) \stackrel{.}{\to} F(Y)$, 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 iterating F and showing that it converges, applying the classic theorem by Adámek [5]. 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.

Function Types The functor $\mathbf{exp}(-)$ is defined by mapping onto exponential objects in Set. 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 co-variant 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.

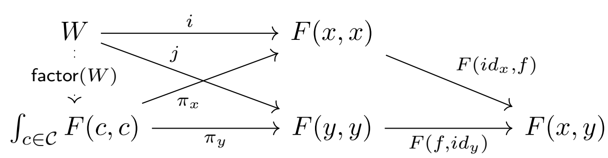
$$\begin{array}{ll}
\mathbf{exp}(F, G)(-) & : (\mathcal{C}^{\mathsf{op}} \times \mathcal{C}) \times \mathcal{D} \to \mathcal{E} \\
\mathbf{exp}(F, G)((x, y), z) & \mapsto G((x, y), z)^{F(y,x)} \\
\mathbf{exp}(F, G)((f, g), h) & \mapsto \mathsf{curry}(G((f, g), h) \circ \mathsf{eval} \circ (id_{\mathbf{exp}(F,G)((x,y),z)}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

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 SET.

Universal quantification The semantics of universal quantifications is expressed in terms of ends in the category SET_1. 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:

$$F(x, x) \xrightarrow{F(id_x, f)} F(x, y) \xleftarrow{F(f, id_y)} F(y, y)$$

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.

$$G(Y)(x, x) \xrightarrow{G(Y)(id_x, f)} G(Y)(x, y) \xleftarrow{G(Y)(f, id_y)} G(Y)(y, y)$$

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 $\mathbf{end}(\mathsf{f})$.

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

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 SET_1.[3] 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.

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

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 [5]. 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 $F : \mathcal{C} \rightarrow \mathcal{C}$ converge after infinite iterations, reaching a fixpoint. If $\mathcal{C}$ has an initial object and *ω-colimits*,[4] we can define the initial algebra of F as the ω-colimit of the following chain:

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

Where $\bot$ is the initial object in $\mathcal{C}$ and $!_X : \bot \rightarrow X$ 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 6 actually inhabits this refined semantics.

Johann and Polonsky [22] 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

[3] See Mac Lane [26] chapter 9.5 corollary 2.

[4] That is, colimits over diagrams defined as a functor on the thin category generated from the poset of natural numbers.

a ω-cocontinuous functor. For λ-abstraction, we transport along the currying isomorphism, which should preserve ω-cocontinuity. For function types, we require that the functor $(-)^X : \textsc{Set} \to \textsc{Set}$ is ω-cocontinuous for all X, which, as Johann and Polonsky [22] point out, is indeed the case. Expanding this proof sketch into a full proof of the existence of initial algebras is future work.

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 $[\![k]\!]$. 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 $[\![k]\!]$, which implies the existence of a *morphism* $\pi_1 : [\![k]\!](x \times y, x)$ for $x, y \in [\![k]\!]$. However, the rule T-FST implies that $\boldsymbol{\pi_1}$ should be related to an *object* in $\textsc{Set}_1$, i.e., $[\![\tau_1 \times \tau_2 \xrightarrow{k} \tau_1]\!]$. To mediate between morphisms in $[\![k]\!]$ and objects in $\textsc{Set}_1$ 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 1. *Given a kind k, morphisms of the category $[\![k]\!]$ are internalized as objects in $\textsc{Set}_1$ through the following bijection between hom-sets:*

$$[\![k]\!](F(\boldsymbol{\delta}) \times [\![\tau_1]\!](\boldsymbol{\delta}^\circ), [\![\tau_2]\!](\boldsymbol{\delta})) \quad \simeq \quad \textsc{Set}_1(F(\boldsymbol{\delta}), [\![\tau_1 \xrightarrow{k} \tau_2]\!](\boldsymbol{\delta})) \qquad (1)$$

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 $F : [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!] \to \textsc{Set}_1$ 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 $k = k_1 \rightsquigarrow \cdots \rightsquigarrow k_n \rightsquigarrow \star$:*

$$\begin{aligned}
& [\![k]\!](F(\boldsymbol{\delta}) \times [\![\tau_1]\!](\boldsymbol{\delta}^\circ), [\![\tau_2]\!](\boldsymbol{\delta})) \\
= \quad & \int_{x_1 \in [\![k_1]\!]} \cdots \int_{x_n \in [\![k_n]\!]} \textsc{Set}_1(F(\boldsymbol{\delta}) \times [\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n), [\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n)) \\
\simeq \quad & \int_{x_1 \in [\![k_1]\!]} \cdots \int_{x_n \in [\![k_n]\!]} \textsc{Set}_1(F(\boldsymbol{\delta}), [\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n)^{[\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n)}) \\
\simeq \quad & \textsc{Set}_1(F(\boldsymbol{\delta}), \int_{x_1} \cdots \int_{x_n} [\![\tau_2]\!](\boldsymbol{\delta})(x_1) \cdots (x_n \in [\![k_n]\!])^{[\![\tau_1]\!](\boldsymbol{\delta}^\circ)(x_1) \cdots (x_n)}) \\
\simeq \quad & \textsc{Set}_1(F(\boldsymbol{\delta}), [\![\tau_1 \xrightarrow{k} \tau_2]\!](\boldsymbol{\delta}))
\end{aligned}$$

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 $\textsc{Set}_1$ *in the subsequent step. This is justified by cartesian closedness of* $\textsc{Set}$*, 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*. Next, we use the fact that the covariant hom-functor* $\textsc{Set}_1(x,-)$ *is continuous and thus preserves ends:*[5]

$$\int_{y\in\mathcal{C}} \textsc{Set}_1(x, G(y,y)) \quad\simeq\quad \textsc{Set}_1(x, \int_{y\in\mathcal{C}} G(y,y)) \tag{2}$$

By repeatedly applying the identity above, we can distribute the aforementioned sequence of ends over the functor $\textsc{Set}_1(F(\boldsymbol{\delta}),-)$*. Intuitively, this corresponds to distributing universal quantification over logical implication in the scenario that the quantified variable does not occur freely in the antecedent, 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 cartesian closedness of* $\mathbf{CAT}$*.*

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

4.5 Interpreting Terms

Well-typed terms, of the form $\Gamma \vdash M : \sigma$, are interpreted as natural transformations from the interpretation their context, $[\![\Gamma]\!]$, to the interpretation of their type, $[\![\sigma]\!]$. At component $\boldsymbol{\delta} \in [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!]$ this transformation is given by a function with 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 7 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 $\textsc{Set}$, which is preserved by its inclusion in $\textsc{Set}_1$. 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

[5] See Mac Lane [26], page 225 Equation 4.

$$
\begin{aligned}
[\![\Gamma \vdash x : \sigma]\!]_{\boldsymbol{\delta}} &= \mathsf{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 : \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 : \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{unin} : \mu(\tau) \xrightarrow{k} \tau\ \mu(\tau)]\!]_{\boldsymbol{\delta}} &= \Uparrow(\mathsf{unin} \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 \times \tau_2 \xrightarrow{k} \tau_1]\!]_{\boldsymbol{\delta}} &= \Uparrow(\pi_1 \circ \pi_2) \\
[\![\Gamma \vdash \boldsymbol{\pi_2} : \tau_1 \times \tau_2 \xrightarrow{k} \tau_2]\!]_{\boldsymbol{\delta}} &= \Uparrow(\pi_2 \circ \pi_2) \\
[\![\Gamma \vdash M \blacktriangle N : \tau \xrightarrow{k} \tau_1 \times \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 + \tau_2]\!]_{\boldsymbol{\delta}} &= \Uparrow(\iota_1 \circ \pi_2) \\
[\![\Gamma \vdash \boldsymbol{\iota_2} : \tau_2 \xrightarrow{k} \tau_1 + \tau_2]\!]_{\boldsymbol{\delta}} &= \Uparrow(\iota_2 \circ \pi_2) \\
[\![\Gamma \vdash M \blacktriangledown N : \tau_1 + \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}} &= !\quad \text{(the unique morphism to the terminal object)} \\
[\![\Gamma \vdash \mathbf{absurd} : \mathbb{0} \Rightarrow \tau]\!]_{\boldsymbol{\delta}} &= \mathsf{curry}(h \circ \pi_2)
\end{aligned}
$$

Fig. 7. Semantics of Well-Typed Terms.

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 Set_1, 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).

Both the denotation function $[\![-]\!]$ as well as the function it computes are total. Consequently, a well-typed value can be computed from every well-typed term. In this sense, the categorical model provides us with a sound computational model of the calculus, which we could implement by writing a definitional interpreter [33]. In the next section, we will discuss how a more traditional small-step operational semantics can be derived from the same categorical model.

5 Operational Semantics

The previous section gave an overview of a categorical semantics of our calculus. In this section, we define a small-step operational semantics for our calculus, and discuss how it relates to the categorical model.

5.1 Reduction Rules

$$
\begin{aligned}
v := {} & \lambda x.M \mid \Lambda\alpha.M \mid \textbf{in}\ \overline{\tau}\ v \mid \textbf{unin}\ \overline{\tau}\ v \mid (v_1 \blacktriangle v_2)\ \overline{\tau}\ v & \text{(Values)}\\
\mid {} & \boldsymbol{\iota_1}\ \overline{\tau}\ v \mid \boldsymbol{\iota_2}\ \overline{\tau}\ v \mid \textbf{map}\langle v\rangle^{\tau'}\ \overline{\tau} \mid (\!|\ v\ |\!)^{\tau'}\ \overline{\tau} \mid \boldsymbol{\pi_1}\ \overline{\tau} \mid \boldsymbol{\pi_2}\ \overline{\tau} & \\
\mid {} & (v_1 \blacktriangledown v_2)\ \overline{\tau} \mid \textbf{tt}\ \overline{\tau} \mid \textbf{absurd}\ \overline{\tau}\ v &
\end{aligned}
$$

$$
\begin{aligned}
E := {} & [] \mid E\ M \mid v\ E \mid E\ \tau \mid \textbf{let}\ (x:\sigma) = E\ \textbf{in}\ M \mid \textbf{let}\ (x:\sigma) = v\ \textbf{in}\ E & \text{(Contexts)}\\
\mid {} & \textbf{map}\langle E\rangle^{\tau} \mid (\!|\ E\ |\!)^{\tau} \mid E \blacktriangle M \mid v \blacktriangle E \mid E \blacktriangledown M \mid v \blacktriangledown E &
\end{aligned}
$$

Fig. 8. Values and Evaluation Contexts. Highlights indicate optional occurrences of (type) arguments

We define our operational semantics as a reduction semantics in the style of Felleisen and Hieb [16]. Figure 8 shows the definition of values and evaluation contexts. In our definition of values, we must account for the fact that language primitives can exist at any kind. For example, the primitive $\boldsymbol{\iota_1}$ by itself is a value of type $\tau_1 \xrightarrow{k} \tau_1 + \tau_2$. Simultaneously, applying $\boldsymbol{\iota_1}$ with a value and/or a sequence of type arguments (the number of which depends on the kind of its arrow type), also yields a value. In fact, all the *partial applications* of $\boldsymbol{\iota_1}$ with only some of its type arguments, or all type arguments but no value argument, are also values. We use gray highlights to indicate such an optional application with type and/or value arguments in the definition of values.

Figure 9 defines the reduction rules. We split the rules in two categories: the first set describes β-reduction[6] for the various type formers, while the second set determines how the $\textbf{map}\langle - \rangle^{-}$ primitive computes. Similar to the definition of values and contexts in Figure 8, we use the notation $\overline{\tau}$ to depict a sequence of zero or more type applications. Unlike for values, these type arguments are not optional; terms typed by an arrow types must be fully applied with all their type arguments before they reduce. The notation $N \bullet M$ is used as a syntactic shorthand for the composition of two arrow types, which is defined through η-expansion of all its type arguments and the term argument. The reduction rules for the $\textbf{map}\langle\tau\rangle^{M}$ primitive are type directed, in the sense that the selected reduction depends on τ. This is necessary, because in an application of $\textbf{map}\langle - \rangle^{-}$

[6] Here, we mean "β-reduction" in the more general sense of simplifying an application of an elimination form to an introduction form.

to a value, there is no way to decide whether to apply the function or to push the $\textbf{map}\langle - \rangle^{-}$ further inwards by only looking at the value.

$$
\begin{aligned}
((\lambda x.M)\ v &\longrightarrow M[v/x] && (1)\\
\textbf{let}\ (x : \sigma) = v\ \textbf{in}\ M &\longrightarrow M[v/x] && (2)\\
(\Lambda\alpha.M)\ \tau &\longrightarrow M[\tau/\alpha] && (3)\\
\textbf{unin}\ \overline{\tau}\ (\textbf{in}\ \overline{\tau}\ v) &\longrightarrow v && (4)\\
(\!|\ v_1\ |\!)^{\tau'}\ \overline{\tau}\ (\textbf{in}\ \overline{\tau}\ v_2) &\longrightarrow v_1\ \overline{\tau}\ (\textbf{map}\langle (\!|\ v_1\ |\!)^{\tau'} \rangle^{\tau'}\ \overline{\tau}\ v_2) && (5)\\
\boldsymbol{\pi_1}\ \overline{\tau}\ ((v_1\ \blacktriangle\ v_2)\ \overline{\tau}\ v) &\longrightarrow v_1\ \overline{\tau}\ v && (6)\\
\boldsymbol{\pi_2}\ \overline{\tau}\ ((v_1\ \blacktriangle\ v_2)\ \overline{\tau}\ v) &\longrightarrow v_2\ \overline{\tau}\ v && (7)\\
(v_1\ \blacktriangledown\ v_2)\ \overline{\tau}\ (\boldsymbol{\iota_1}\ \overline{\tau}\ v)) &\longrightarrow v_1\ \overline{\tau}\ v && (8)\\
(v_1\ \blacktriangledown\ v_2)\ \overline{\tau}\ (\boldsymbol{\iota_2}\ \overline{\tau}\ v)) &\longrightarrow v_2\ \overline{\tau}\ v && (9)
\end{aligned}
$$

$$
\begin{aligned}
\textbf{map}\langle v_1 \rangle^{(\lambda X.X)}\ \overline{\tau}\ v_2 &\longrightarrow v_1\ \overline{\tau}\ v_2 && (10)\\
\textbf{map}\langle v_1 \rangle^{\mu(\tau')}\ \overline{\tau}\ (\textbf{in}\ \overline{\tau}\ v_2) &\longrightarrow \textbf{in}\ \overline{\tau}\ (\textbf{map}\langle v_1 \rangle^{(\tau'\ \mu(\tau'))}\ \overline{\tau}\ v_2) && (11)\\
\textbf{map}\langle v \rangle^{\tau_1 \times \tau_2}\ \overline{\tau}\ ((v_1\ \blacktriangle\ v_2)\ \overline{\tau}\ v_3) &\longrightarrow ((\textbf{map}\langle v \rangle^{\tau_1} \bullet v_1)\ \blacktriangle\ (\textbf{map}\langle v \rangle^{\tau_2} \bullet v_2))\ \overline{\tau}\ v_3 && (12)\\
\textbf{map}\langle v_1 \rangle^{\tau_1 + \tau_2}\ \overline{\tau}\ (\boldsymbol{\iota_1}\ \overline{\tau}\ v_2) &\longrightarrow \boldsymbol{\iota_1}\ \overline{\tau}\ (\textbf{map}\langle v_1 \rangle^{\tau_1}\ \overline{\tau}\ v_2) && (13)\\
\textbf{map}\langle v_1 \rangle^{\tau_1 + \tau_2}\ \overline{\tau}\ (\boldsymbol{\iota_2}\ \overline{\tau}\ v_2) &\longrightarrow \boldsymbol{\iota_2}\ \overline{\tau}\ (\textbf{map}\langle v_1 \rangle^{\tau_2}\ \overline{\tau}\ v_2) && (14)\\
\textbf{map}\langle v \rangle^{\mathbf{1}}\ \overline{\tau}\ (\textbf{tt}\ \overline{\tau}) &\longrightarrow \textbf{tt}\ \overline{\tau} && (15)
\end{aligned}
$$

$$N \bullet M \ \triangleq\ \overline{\Lambda\alpha}.\lambda x.N\ \overline{\alpha}\ (M\ \overline{\alpha}\ x)$$

Fig. 9. Reduction rules

5.2 Relation to the Denotational Model

The reduction rules shown in Figure 9 define a computational model for our calculus. We now discuss how this model arises from the denotational model discussed in Section 4. Informally speaking, reducing a term should not change its meaning. This intuition is reflected by the following implication, which states if M reduces N, their semantics should be equal.[7]

$$M \longrightarrow N \implies [\![M]\!] = [\![N]\!] \tag{3}$$

While we do not give a formal proof of the implication above, by relying on the categorical model to inform how terms compute we can be reasonably confident that our semantics does not contain any reductions that violate this property. That is, all the reductions shown in Figure 9 are supported by an equality of morphisms in the categorical model.

What does this mean, specifically? The semantics of well-typed terms is given by a natural transformation, so if $M \longrightarrow N$, M and N should be interpreted as

[7] This property implies what Devesas Campos and Levy [15] call *soundness* of the denotational model with respect to the operational model. Their soundness property is about a big-step relation; ours is small-step.

the same natural transformation. Equivalence of natural transformations is defined pointwise in terms of the equality relation for morphisms in the underlying category. In our case, this is the category SET, as terms are interpreted as natural transformations between functors into SET. By studying the properties—expressed as equalities between morphisms—of the constructions that give a semantics to the different type formers, and reifying these equalities as syntactic reduction rules, we obtain an operational model that we conjecture respects the denotational model by construction.

Let us illustrate this principle with a concrete example. The semantics of a sum type $\tau_1 + \tau_2 : k$ is given by a coproduct in the category $[\![k]\!]$. The universal property of coproducts tells us that $[f, g] \circ \iota_1 = f$ and $[f, g] \circ \iota_2 = g$, or in other words, constructing and then immediately deconstructing a coproduct is the same as doing nothing. Rules (8) and (9) in Figure 9 reflect these equations. That is, since the $\boldsymbol{\iota_1}$, $\boldsymbol{\iota_2}$, and $- \blacktriangledown -$ primitives are interpreted as the injections ι_1, ι_2, and unique morphism $[-,-]$ respectively, the universal property of coproducts tells us that the left-hand side and right-hand side of rule (8) and (9) in Figure 9 are interpreted to equal morphism in the categorical domain.

The remaining reduction rules are justified by the categorical model in a similar fashion. More specifically:

- Rules (1,2) follow from the β-law for exponential objects, which states that $\mathsf{eval} \circ \langle \mathsf{curry}(f), id\rangle = f$.
- Rule (3) holds definitionally, assuming type substitution is appropriately defined such that it corresponds to functor application.
- Rule (4) follows from Lambek's lemma, which states that the component of an initial algebra is always an isomorphism. That is, there exists a morphism unin such that $\mathsf{unin} \circ \mathsf{in} = id$.
- Rule (5) reflects the universal property of folds, i.e., $\mathsf{cata}(f) \circ \mathsf{in} = f \circ F(\mathsf{cata}(f))$.
- Rules (6,7) follow from the universal property of products, which states that $\pi_1 \circ \langle f, g\rangle = f$ and $\pi_2 \circ \langle f, g\rangle = g$.
- Rule (10) mirrors the identity law for functors, i.e. $F(id) = id$.
- Rule (11) is derived from naturality of the component of the initial algebra of higher-order functors, which states that $\boldsymbol{\mu}(F)(f) \circ \mathsf{in} = \mathsf{in} \circ F(\boldsymbol{\mu}(F))(f)$.
- Rule (12,13,14,15) are derived from the way (co)-limits are computed pointwise in functor categories. For example, the morphism action of the product of two functors F and G is defined as $(F \times G)(f) = \langle F(f) \circ \pi_1, G(f) \circ \pi_2\rangle$, which gives rise to rule (12).

6 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 [24], 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. [41] 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 [11, 25] and *final algebra semantics* [38], which, according to Wand [38, §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 [29] 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. [10] 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. [4] 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 [22], Johann et al. [21], and Johann and Ghiorzi [20], 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 ∀-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 uses a *universe of descriptions* [6], 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 [14]. In fact, Chapman et al. [12] 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.

7 Conclusion and Future work

In this paper, we presented the design and semantics of a calculus with support for modularity. We demonstrated it can serve as a 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.

Acknowledgements. This research was partially funded by the NWO VENI Composable and Safe-by-Construction Programming Language Definitions project (VI.Veni.192.259).

References

1. Abbott, M.G., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theor. Comput. Sci. **342**(1), 3–27 (2005), https://doi.org/10.1016/j.tcs.2005.06.002
2. Abel, A., Matthes, R.: (co-)iteration for higher-order nested datatypes. In: Geuvers, H., Wiedijk, F. (eds.) Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers. Lecture Notes in Computer Science, vol. 2646, pp. 1–20. Springer (2002). https://doi.org/10.1007/3-540-39185-1_1, https://doi.org/10.1007/3-540-39185-1_1
3. Abel, A., Matthes, R., Uustalu, T.: Generalized iteration and coiteration for higher-order nested datatypes. In: Gordon, A.D. (ed.) Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2620, pp. 54–69. Springer (2003). https://doi.org/10.1007/3-540-36576-1_4, https://doi.org/10.1007/3-540-36576-1_4
4. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theor. Comput. Sci. **333**(1-2), 3–66 (2005). https://doi.org/10.1016/j.tcs.2004.10.017, https://doi.org/10.1016/j.tcs.2004.10.017

5. Adámek, J.: Free algebras and automata realizations in the language of categories. Commentationes Mathematicae Universitatis Carolinae **15**(4), 589–602 (1974)
6. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nord. J. Comput. **10**(4), 265–289 (2003)
7. van den Berg, B., Schrijvers, T., Poulsen, C.B., Wu, N.: Latent effects for reusable language components. In: Oh, H. (ed.) Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13008, pp. 182–201. Springer (2021), https://doi.org/10.1007/978-3-030-89051-3_11
8. Bird, R.S., Meertens, L.G.L.T.: Nested datatypes. In: Jeuring, J. (ed.) Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1422, pp. 52–67. Springer (1998), https://doi.org/10.1007/BFb0054285
9. Bird, R.S., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects Comput. **11**(2), 200–222 (1999), https://doi.org/10.1007/s001650050047
10. Cai, Y., Giarrusso, P.G., Ostermann, K.: System f-omega with equirecursive types for datatype-generic programming. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 30–43. ACM (2016), https://doi.org/10.1145/2837614.2837660
11. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009), https://doi.org/10.1017/S0956796809007205
12. Chapman, J., Dagand, P., McBride, C., Morris, P.: The gentle art of levitation. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 3–14. ACM (2010), https://doi.org/10.1145/1863543.1863547
13. Coquand, T., Paulin, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings. Lecture Notes in Computer Science, vol. 417, pp. 50–66. Springer (1988), https://doi.org/10.1007/3-540-52335-9_47
14. Dagand, P.: A cosmology of datatypes : reusability and dependent types. Ph.D. thesis, University of Strathclyde, Glasgow, UK (2013), http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713
15. Devesas Campos, M., Levy, P.B.: A syntactic view of computational adequacy. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10803, pp. 71–87. Springer (2018), https://doi.org/10.1007/978-3-319-89366-2_4
16. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2), 235–271 (1992), https://doi.org/10.1016/0304-3975(92)90014-7
17. Glimming, J., Ghani, N.: Difunctorial semantics of object calculus. In: Bono, V., Bugliesi, M., Drossopoulou, S. (eds.) Proceedings of the Second Workshop on Object Oriented Developments, WOOD 2004, London, UK, August 30, 2004. Electronic Notes in Theoretical Computer Science, vol. 138, pp. 79–94. Elsevier (2004), https://doi.org/10.1016/j.entcs.2005.09.012

18. Goguen, J.A.: An intial algebra approach to the specification, correctness and implementation of abstract data types. IBM Research Report **6487** (1976)
19. Hubers, A., Morris, J.G.: Generic programming with extensible data types: Or, making ad hoc extensible data types less ad hoc. Proc. ACM Program. Lang. **7**(ICFP) (aug 2023), https://doi.org/10.1145/3607843
20. Johann, P., Ghiorzi, E.: Parametricity for nested types and gadts. Log. Methods Comput. Sci. **17**(4) (2021), https://doi.org/10.46298/lmcs-17(4:23)2021
21. Johann, P., Ghiorzi, E., Jeffries, D.: Parametricity for primitive nested types. In: Kiefer, S., Tasson, C. (eds.) 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, vol. 12650, pp. 324–343. Springer (2021), https://doi.org/10.1007/978-3-030-71995-1_17
22. Johann, P., Polonsky, A.: 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. pp. 1–13. IEEE (2019), https://doi.org/10.1109/LICS.2019.8785657
23. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013. pp. 145–158. ACM (2013), https://doi.org/10.1145/2500365.2500590
24. Kieburtz, R.B., Lewis, J.: Programming with algebras. In: Jeuring, J., Meijer, E. (eds.) 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, vol. 925, pp. 267–307. Springer (1995), https://doi.org/10.1007/3-540-59451-5_8
25. Kiselyov, O.: Typed tagless final interpreters. In: Gibbons, J. (ed.) Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures. Lecture Notes in Computer Science, vol. 7470, pp. 130–174. Springer (2010), https://doi.org/10.1007/978-3-642-32202-0_3
26. MacLane, S.: Categories for the Working Mathematician. Springer-Verlag, New York (1971), graduate Texts in Mathematics, Vol. 5
27. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. Lecture Notes in Computer Science, vol. 523, pp. 124–144. Springer (1991), https://doi.org/10.1007/3540543961_7
28. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991), https://doi.org/10.1016/0890-5401(91)90052-4
29. Morris, J.G., McKinna, J.: Abstracting extensible data types: or, rows by any other name. Proc. ACM Program. Lang. **3**(POPL), 12:1–12:28 (2019), https://doi.org/10.1145/3290325
30. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) 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, vol. 5502, pp. 80–94. Springer (2009), https://doi.org/10.1007/978-3-642-00590-9_7

31. Poulsen, C.B., van der Rest, C.: Hefty algebras: Modular elaboration of higher-order algebraic effects. Proc. ACM Program. Lang. **7**(POPL), 1801–1831 (2023), https://doi.org/10.1145/3571255
32. van der Rest, C., Poulsen, C.B.: Towards a language for defining reusable programming language components - (project paper). In: Swierstra, W., Wu, N. (eds.) Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13401, pp. 18–38. Springer (2022), https://doi.org/10.1007/978-3-031-21314-4_2
33. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. High. Order Symb. Comput. **11**(4), 363–397 (1998), https://doi.org/10.1023/A:1010027404223
34. Schrijvers, T., Piróg, M., Wu, N., Jaskelioff, M.: Monad transformers and modular algebraic effects: what binds them together. In: Eisenberg, R.A. (ed.) Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019. pp. 98–113. ACM (2019), https://doi.org/10.1145/3331545.3342595
35. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008), https://doi.org/10.1017/S0956796808006758
36. Uustalu, T., Vene, V.: Mendler-style inductive types, categorically. Nord. J. Comput. **6**(3), 343 (1999)
37. Wadler, P.: The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (1998), accessed: 2020-07-01
38. Wand, M.: Final algebra semantics and data type extensions. J. Comput. Syst. Sci. **19**(1), 27–44 (1979), https://doi.org/10.1016/0022-0000(79)90011-4
39. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Swierstra, W. (ed.) Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014. pp. 1–12. ACM (2014), https://doi.org/10.1145/2633357.2633358
40. Yang, Z., Paviotti, M., Wu, N., van den Berg, B., Schrijvers, T.: Structured handling of scoped effects. In: Sergey, I. (ed.) 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, vol. 13240, pp. 462–491. Springer (2022), https://doi.org/10.1007/978-3-030-99336-8_17
41. Zhang, W., Sun, Y., d. S. Oliveira, B.C.: Compositional programming. ACM Trans. Program. Lang. Syst. **43**(3), 9:1–9:61 (2021), https://doi.org/10.1145/3460228