Types and Semantics for Extensible Data Types

Cas van der Rest Casper Bach Poulsen

June 15, 2023

Modularity is a key concern in functional programming languages

Modularity is a key concern in functional programming languages

- The Expression Problem: how to extend recursive data and functions in a type-safe way

Modularity is a key concern in functional programming languages

- The Expression Problem: how to extend recursive data and functions in a type-safe way
- Defining monadic effects in a modular way

In practice, these modularity concerns are often addressed by embedding the *Initial Semantics* of inductive data types

```
data Fix (F : Set → Set) : Set where
  In : F (Fix F) → Fix F
```

```
data Free (F : Set → Set) A : Set where
  Pure   : A → Free F A
  Impure : F (Free F A) → Free F A
```

The Same ideas work for *higher-order* functors

```
data Fix (H : (Set → Set) → Set → Set) A : Set where
  In : H (Fix H) A → Fix H A
```

```
data Prog (H : (Set → Set) → Set → Set) A : Set where
  Pure   : A → Prog H A
  Impure : H (Prog H) A → Prog H A
```

Such structures are used e.g. to encode *nested data types*[1] or Scoped Effects[2]

[1] Bird and Meertens, 1998
[2] Wu et al., 2014

These techniques are tremendously useful, but it's unfortunate that we have to rely on embeddings to use them

- Adds some noise compared to built-in data types, interoperability through isomorphisms
- Connection to the underlying theory remains implicit

How would we *design a programming language* that has type safe modularity for data types built in?

This work

Here, we work towards laying the groundwork for developing programming languages with built-in support for type-safe modularity

How? By designing a calculus that *captures the essential features* of type-safe modularity for data types

Calculus Design

A λ-calculus with rank-1 polymorphism and kinds

Well-kindedness for types is defined such that *all* type formers with kind $k_1 \rightarrow k_2$ are a functor

Built-in primitives for mapping and folding

Types

$$k := \star \mid k \to k$$
$$\tau := \alpha \mid \tau\ \tau \mid \lambda\alpha.\tau \mid \mu(\tau) \mid \tau \Rightarrow \tau \mid \top \mid \bot \mid \tau \otimes \tau \mid \tau \oplus \tau$$
$$\sigma := \forall\alpha.\sigma \mid \tau$$

Types

$$k := \star \mid k \to k$$
$$\tau := \alpha \mid \tau\ \tau \mid \lambda\alpha.\tau \mid \mu(\tau) \mid \tau \Rightarrow \tau \mid \top \mid \bot \mid \tau \otimes \tau \mid \tau \oplus \tau$$
$$\sigma := \forall\alpha.\sigma \mid \tau$$

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

K-Abs
$$\frac{\Delta \mid \Phi, (\alpha \mapsto k_1) \vdash \tau : k_2}{\Delta \mid \Phi \vdash \lambda\alpha.\tau : k_1 \to k_2}$$

K-Fun
$$\frac{\Delta \mid \emptyset \vdash \tau_1 : \star \qquad \Delta \mid \Phi \vdash \tau_2 : \star}{\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star}$$

Example: Free Monad

$$Free \triangleq \lambda f.\lambda a.\mu(\lambda X.a \oplus f\,X) : (\star \to \star) \to \star \to \star$$

By construction, *Free* is a functor in both *f* and *a*.

The former means that we can apply natural transformations to change the monad's signature.

Terms

$$
\begin{array}{lll}
M & := & \ldots \mid \textbf{in} \mid \textbf{out} \mid \textbf{map}\langle M \rangle^{\tau} \mid (\!|\ M\ |\!)^{\tau} \\
 & \mid & \boldsymbol{\pi_1} \mid \boldsymbol{\pi_2} \mid M \blacktriangle M \mid \boldsymbol{\iota_1} \mid \boldsymbol{\iota_2} \mid M \blacktriangledown M \mid \textbf{tt} \mid \textbf{absurd}
\end{array}
$$

Terms

$$
\begin{array}{rcl}
M & := & \ldots \mid \mathbf{in} \mid \mathbf{out} \mid \mathbf{map}\langle M \rangle^{\tau} \mid (\!| M |\!)^{\tau} \\
 & \mid & \boldsymbol{\pi_1} \mid \boldsymbol{\pi_2} \mid M \blacktriangle M \mid \boldsymbol{\iota_1} \mid \boldsymbol{\iota_2} \mid M \blacktriangledown M \mid \mathbf{tt} \mid \mathbf{absurd}
\end{array}
$$

These primitives have both first and higher order instances

How to type the higher-order instances of these language primitives?

Arrow Types

$$\tau_1 \xrightarrow{\star} \tau_2 \quad \triangleq \quad \tau_1 \Rightarrow \tau_2$$
$$\tau_1 \xrightarrow{k_1 \to k_2} \tau_2 \quad \triangleq \quad \forall \alpha . \tau_1 \ \alpha \xrightarrow{k_2} \tau_2 \ \alpha$$

An *arrow type*, $\tau_1 \xrightarrow{k} \tau_2$ at kind k describes a *morphism* between the "objects" of kind k

I.e., for $k = \star$ it's functions, and for $k = \star \to \star$ it's natural transformations between functors on $\textsc{Set}$, etcetera ...

$$\frac{\tau_1, \tau_2 : k}{\Gamma \vdash \boldsymbol{\pi_1} : \tau_1 \xrightarrow{k} \tau_2} \; \text{T-FST}$$

T-Fst

$$\frac{\tau_1, \tau_2 : k}{\Gamma \vdash \boldsymbol{\pi_1} : \tau_1 \xrightarrow{k} \tau_2}$$

T-Map

$$\frac{\tau : k_1 \to k_2 \qquad \tau_1, \tau_2 : k_1 \qquad \Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2}{\Gamma \vdash \mathbf{map}\langle M \rangle^{\tau} : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2}$$

Semantics

- Interpret types as objects in Set and its functor categories
- Interpret terms as natural transformations from a the functor interpreting its context to a functor interpreting its type

Kinds:

$$\begin{array}{rcl} [\![-]\!] & : & \mathit{Kind} \to \mathit{Cat} \\ [\![\star]\!] & = & \textsc{Set} \\ [\![k_1 \to k_2]\!] & = & [[\![k_1]\!], [\![k_2]\!]] \end{array}$$

Kinds:

$$\begin{array}{rcl} [\![-]\!] & : & \mathit{Kind} \to \mathit{Cat} \\ [\![\star]\!] & = & \textsc{Set} \\ [\![k_1 \to k_2]\!] & = & [[\![k_1]\!], [\![k_2]\!]] \end{array}$$

Types:

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

$$[\![\Delta \vdash \sigma]\!] : [\![\Delta]\!]^{\mathsf{op}} \times [\![\Delta]\!] \to \textsc{Set}$$

Kinds:

$$\begin{array}{rcl} [\![-]\!] & : & \mathit{Kind} \to \mathit{Cat} \\ [\![\star]\!] & = & \textsc{Set} \\ [\![k_1 \to k_2]\!] & = & [[\![k_1]\!], [\![k_2]\!]] \end{array}$$

Types:

$$[\![\Delta \mid \Phi \vdash \tau : k]\!] : ([\![\Delta]\!]^{op} \times [\![\Delta]\!]) \times [\![\Phi]\!] \to [\![k]\!]$$

$$[\![\Delta \vdash \sigma]\!] : [\![\Delta]\!]^{op} \times [\![\Delta]\!] \to \textsc{Set}$$

Terms:

$$[\![\Gamma \vdash M : \sigma]\!] : \mathit{Nat}([\![\Gamma]\!], [\![\tau]\!])$$

Arrow Type Semantis

Function types are interpreted as *exponential objects* and universal quantifications as *ends*

Consequently:

$$\llbracket \tau_1 \xrightarrow{k} \tau_2 \rrbracket(X, Y) \mapsto \int_{X_1} \dots \int_{X_n} \llbracket \tau_2 \rrbracket(X, Y)(X_1, \dots, X_n)^{\llbracket \tau_1 \rrbracket(X,Y)(X_1,\dots,X_n)}$$

Take the primitive $\boldsymbol{\pi_1}$ at kind k. To define its semantics, we would like to appeal to the cartesian structure of $[\![k]\!]$

But $\boldsymbol{\pi_1}$ has an arrow type, whose semantics is a functor into $\textsc{Set}$

Thus, we must show that the semantics of an arrow type $\tau_1 \xrightarrow{k} \tau_2$ internalizes the morphisms between $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ in $[\![k]\!]$ as an object in $\textsc{Set}$

"Currying" for Arrow Types

$$[\![k]\!](X \times [\![\tau_1]\!], [\![\tau_2]\!]) \simeq \mathrm{S{\small ET}}(X, [\![\tau_1 \xrightarrow{k} \tau_2]\!])$$

Operational Model

We can extract an operational model from the categorical model by looking the equations we get on the categorical side

Operational Model

We can extract an operational model from the categorical model by looking the equations we get on the categorical side

$$(\!|\ M\ |\!)^{\tau}\ (\mathbf{in}\ N) \longrightarrow M\ (\mathbf{map}\langle(\!|\ M\ |\!)^{\tau}\rangle^{\tau}\ N)$$

$$(M_1 \blacktriangledown M_2)\ (\boldsymbol{\iota_1}\ N) \longrightarrow M_1\ N$$

Operational Model

We can extract an operational model from the categorical model by looking the equations we get on the categorical side

$$(\!|\, M \,|\!)^\tau \; (\mathbf{in}\ N) \longrightarrow M\ (\mathbf{map}\langle (\!|\, M \,|\!)^\tau \rangle^\tau\ N)$$

$$(M_1 \blacktriangledown M_2)\ (\boldsymbol{\iota_1}\ N) \longrightarrow M_1\ N$$

Can we say something about how "good" this operational model is (e.g., preservation, progress, ...)

Conclusion

We designed a calculus that can capture many existing programming patterns for modularity

But, there's plenty to do still:

- Categorical model is tied to $\textsc{Set}$, would like to generalize
- More formal connection between categorical and operational model
- Increased expressiveness using generalized or adjoint folds