FP 2.0 [1][2][3]

Fall 2025

Contents

[1] Author Andrey Stoyan (`andrey.stoyan.csam@gmail.com`).

[2] Thanks to Ilya Kolegov for the first careful read and many comments. Thanks to my students whose interest in the subject made this work possible.

[3] Version 1 мая 2026 г., 18:10.

Introduction

Many complex concepts in language design and programming can be understood as special cases of some simple fundamental principles that are usually treated as common folklore and do not require extra explanation. However, the difficulty is that this knowledge is scattered across books, papers, and "cult" blog posts, and it takes a lot of time and effort to reconstruct the whole picture.

The goal of this course is to gather such folklore knowledge in one place and organize it into a coherent system. The course will explicitly rely on classical works studying the principles of language design and will help in studying them. Reading the referenced papers is an important part of independent work in the course.

By functional programming, as highlighted in the course title, we mean a careful attitude towards the notion of effect, which in FP — unlike in other schools of thought — is not considered an axiomatic given but a subject for study, conscious construction, and careful handling. This approach turns out to be very useful for learning languages and for building powerful language constructs; it is also the foundation of a productive programming style. Moreover, functional languages are relatively simple, which means new ideas and approaches often originate there and spread further.

Haskell is chosen as the main language of the course because, on the one hand, it embodies many concepts that are often brought to a certain logical completion and is powerful enough to encode others. On the other hand, it remains a practical industrial programming language.

Because of the breadth of the topic, this course is not always deep. Thus, implementation details in GHC or the category-theoretic foundations of some concepts may be given in general terms, without going into specifics. At the same time, in the realm of language design through the lens of functional programming, the course tries to be as detailed as possible.

Thus, this course may be useful for those who are interested in language design and elegant generalizations of programming concepts, want to improve their API design skills, or plan to work practically with functional languages.

A prerequisite for the course is knowledge of the basics of functional programming: algebraic data types, pattern matching, folds, parametric polymorphism, type classes, and basic monads. It is also useful to be able to read type judgments, and to be familiar with polymorphic lambda calculus and Church encoding.

1 FP Recollections

In this section we will recall the main concepts of functional programming and the Haskell language.

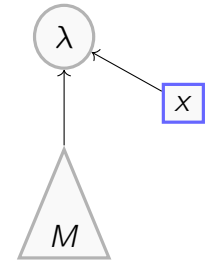
1.1 Terms and Reduction

In FP, programs are expressions. Program execution is the reduction of those expressions to simpler ones. Expressions can be represented either as a linear sequence of symbols or as a tree, which makes them easier to understand without requiring knowledge of auxiliary associativity rules and so on.

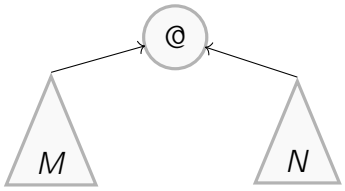
The simplest functional language is the lambda calculus. Expressions in it are called lambda terms, which consist of nodes of three kinds (V is the set of valid identifiers, Λ is the set of lambda terms):

Variables $x \in \Lambda$, if $x \in V$

Abstraction $(\lambda x.\, M) \in \Lambda$, if $x \in V, M \in \Lambda$



Application $(M\ N) \in \Lambda$, if $M \in \Lambda, N \in \Lambda$



In an arbitrary expression, some subterm can be replaced with a formal parameter that must be declared higher in the tree by a special lambda node. Instead of the formal parameter, we can later substitute various concrete arguments using the application node @ and reuse the expression for different purposes (for example, see Fig. 1). Reduction is defined by the following rewriting rule: an application of a lambda function to an argument is found, and the argument is substituted for all free occurrences of the variable bound by the lambda in the function body (Fig. 2).

1.2 Types

Software is hard. Therefore, bugs constantly and inevitably appear in programs. They can be found, among other ways, statically, that is, without running the program. One form of static analysis is type checking.

The idea of type analysis is that we attempt to assign a syntactic label to every node of the program tree according to certain rules. If every node can be labeled that way, then we say the

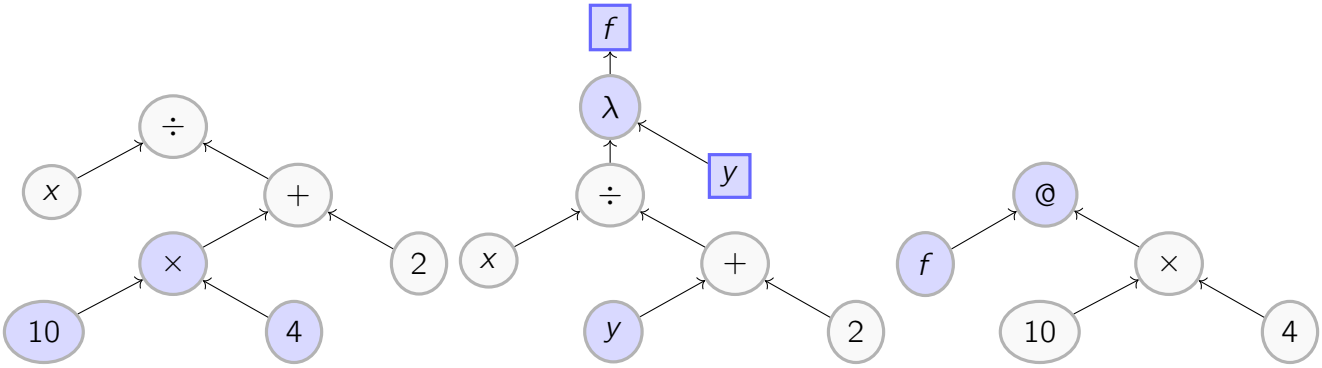


Figure 1: The expression represented with a lambda node is transformed into a one-argument function.

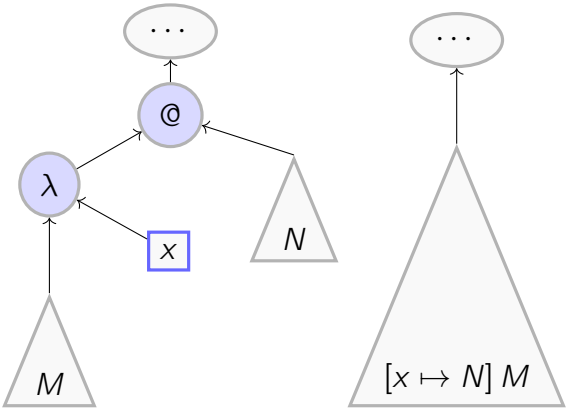


Figure 2: Reduction rewrites the tree by substituting a concrete argument for the formal parameter.

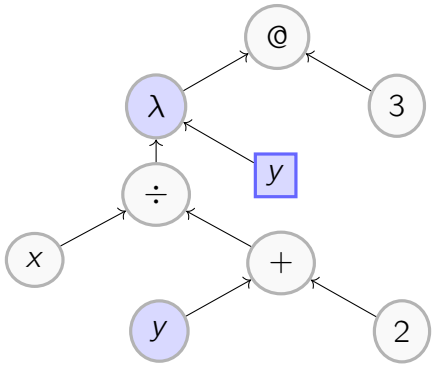


Figure 3: The tree corresponding to the expression $(\lambda y.\, x \div (y + 2))\ 3$.

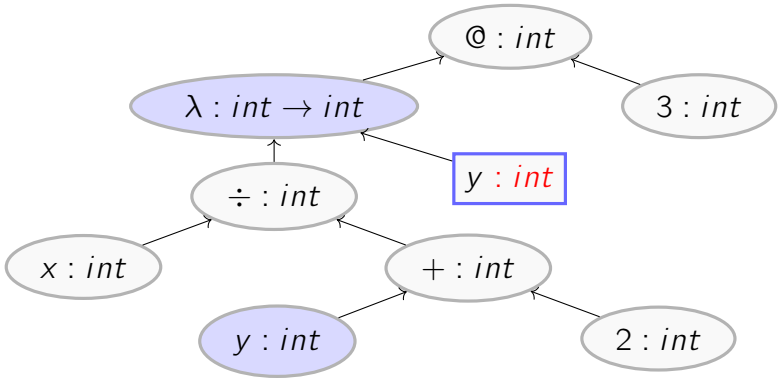


Figure 4: The tree for $(\lambda y.\, x \div (y + 2))\ 3$ after assigning type labels.

program type-checks and is "good". For example, Fig. 3 shows an expression, and Fig. 4 shows the same tree with labels assigned consistently with a type system.

A type system defines the syntax of type labels and the rules by which they can be assigned. Syntax is usually described in a classical BNF-like notation, and rules are expressed as typing judgments. For example, the rules for simply-typed lambda calculus look like this:

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ ctx \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ elim \to \qquad \frac{\{x : \sigma\} \cup \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, M : \sigma \to \tau}\ intro \to$$

Type labels are purely syntactic, but they can be interpreted. The most popular interpretation is to treat a type label as a set. Thus, the label $int \to int$ can correspond to the set of functions between the sets of integers $\mathbb{Z} \to \mathbb{Z}$.

1.3 Functions in Haskell

At its core, Haskell is an extended typed lambda calculus enriched with primitive types, the ability to declare new names, data structures, and type classes.

Examples of lambda abstractions in the GHCi REPL:

```
ghci> (\x -> x + 1) 4
5
```

You can learn the type of a function in the interpreter (in reality numbers are polymorphic, but we will discuss that later):

```
ghci> :t \x -> x + 1
\x -> x + 1 :: Int -> Int
```

Functions can be given names. Names can be annotated with types, and it is recommended to do so explicitly for top-level declarations in source files:

```
f :: Int -> Int
f x = x + 1
```

If a type name begins with a lowercase letter, it is not a concrete type but a type variable that can take different values depending on the call site. This capability is called **parametric polymorphism**. Thus, a function that simply returns its argument does not constrain the argument type. At the same time, the result type must match the argument type:

```
id :: a -> a
id x = x

ghci> :t id 5
id 5 :: Int
```

Functions can take other functions as arguments (such functions are called **higher-order functions**). A function name can consist of special symbols, in which case it is treated as an operator and can be used in infix form:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Example of a recursive function using guards to distinguish the base case:

```
factorial :: Int -> Int
factorial n
  | n < 1 = 1
  | otherwise = n * factorial (n - 1)
```

Exercise 1 *What does the command* `ghci> :t uncurry (flip const)` *return?*

Exercise 2 *What does* `ghci> :t first . first` *return given*

```
first :: (a -> a') -> (a, b) -> (a', b)
```

Exercise 3 *Implement factorial using the accumulator pattern.*

1.4 Data in Haskell

Haskell has built-in support for declaring new data types from other types and constructing values of those types.

Define a data type describing animals:

```
data Animal
  = Cat String Int
  | Dog String
```

We defined the data type `Animal` and two ways to create values of this type: one for cats and one for dogs. `Cat` and `Dog` are **data constructors**. They are functions whose implementation is provided by the language. They allocate memory for instances of the type and place components positionally. Cats are described by a name and the remaining number of lives, while dogs are described only by a name.

```
Cat :: String -> Int -> Animal
Dog :: String -> Animal
```

To use the information stored in a data structure, it must be deconstructed using pattern matching. We match the value against a pattern. If the pattern resembles the way the value was constructed, it is selected among other alternatives and the variables declared in it bind to the corresponding positional contents of the data structure:

```
show :: Animal -> String
show animal = case animal of
  Cat name nLives -> "This is cat " ++ name ++ " " ++ show nLives
  Dog name -> "This is dog " ++ name
```

Haskell offers special syntax for declaring record fields with named labels.

```
data Penguin = Penguin { getName :: String, getAge :: Int }
penguin = Penguin { getName = "Andrey", getAge = 500 }
```

Haskell generates accessor functions for field access:

```
ghci> :t getName :: Penguin -> String
```

Often functions in programming are partial — for some argument values they return a result, and for others they do not. Let us model this with a dedicated data type. If there is a real result, we return it. If there is none, we return a specially reserved constant value of that type.

```
data MaybeD = NothingD | JustD Double
sqrt :: Double -> MaybeD
sqrt x = if x < 0 then NothingD else JustD (calcSqrt x)
```

One can notice that this would require defining a separate type `MaybeT` for each type `T`. Therefore Haskell allows type-level abstraction, just as we can abstract over values in terms.

```haskell
data Maybe a = Nothing | Just a
sqrt :: Double -> Maybe Double
sqrt x = if x < 0 then Nothing else Just (calcSqrt x)
```

Note that `Maybe` is not a concrete type itself; it requires a type parameter to produce a concrete type. `Maybe` is called a **type constructor**.

With abstraction at the type level also comes application of types to types. What if a type constructor is given fewer parameters than expected? What if it is given more? Correctness of type applications is enforced by the **kind system**[4]. Kinds are the simplest "types for types", syntactic labels that control whether the types written by the programmer are valid. Ordinary types have kind `*`. Type constructors have arrow kinds. For example, `Maybe :: * -> *`. Applying a type constructor to a type of the appropriate kind consumes one arrow:

```haskell
ghci> :k Int
Int :: *
ghci> :k Maybe
Maybe :: * -> *
ghci> :k Maybe Int
Maybe Int :: *
```

In addition to brand-new data types, Haskell allows type synonyms. These are names that can be used instead of other types when the original type expression is too long to write everywhere:

```haskell
type T a = VeryLongType Int (a -> AnotherLongType a)
```

If a data type has only one constructor and only one field, there is no need to allocate runtime memory containing a constructor tag and field references. In that case, the value of the new type can be represented by the wrapped value itself, leaving the new type present only at compile time and reducing runtime overhead. To declare such wrapper types, use `newtype` instead of `data`:

```haskell
newtype CourseId = CourseId Int64
newtype ModuleId = ModuleId Int64
```

Exercise 4 *Define the kind of the type constructor*

```haskell
data Free f a = Pure a | Free (f (Free f a))
```

1.5 Type Classes in Haskell

Parametric polymorphism allows the same code to work for different input types. Type classes allow one identifier to refer to different implementations for different data types (which is analogous to overloading in other languages). Type classes are, as they say, the mechanism of **ad-hoc polymorphism**. For example, we can declare the symbol ==, whose implementation depends on the chosen type `a`:

[4] In Russian literature, kinds are sometimes called "genera of types", but we will not use that terminology.

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

For each type, one can declare its own implementation of `Eq`:

```haskell
instance Eq CourseId where
  CourseId x == CourseId y = x == y

instance Eq a => Eq [a] where
  [] == [] = True
  x:xs == y:ys = x == y && xs == ys
```

Now, depending on the concrete type `a` at the call site, the appropriate implementation for that type will be selected:

```
ghci> CourseId 1 == CourseId 2
False
ghci> [CourseId 1, CourseId 2] == [CourseId 1, CourseId 2]
True
```

Earlier, parametrically polymorphic functions could only do one of two things with their arguments: return them or pass them to other polymorphic functions. To do anything else, additional information about the type is needed, because otherwise there is no guarantee that the required operations are defined for a value of that type. For example, the function `suc n = n + 1` will not work for strings, because addition is not defined for them. Therefore it would be incorrect to assign it the polymorphic type `suc :: a -> a`.

Type classes, in addition to overloading, are also a mechanism for restricting polymorphism. We can explicitly specify that a function requires not any type as input, but any type for which the operations we need are defined. Thus, for the type of `suc` it is enough to constrain it with the presence of addition (the operation denoted by + is declared in the `Num` type class):

```haskell
suc :: Num a => a -> a
```

Exercise 5 *Implement a function that checks whether all elements of a list are equal.*

Exercise 6 *Implement the semigroup instance for functions.*

Exercise 7 *Implement equality checking for functions.*

1.6 Monads in Haskell

The type class `Functor` is declared for type constructors and allows replacing every element of one type in a container with elements of another type while preserving the container structure.

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x:xs) = f x : fmap f xs
```

In Haskell, every function simply produces a result of some type. However, in programming we often need functions that not only produce a result but also do something else. For example, modify some state or write to the console. In other words, they perform side effects. In any case, in Haskell we can only return a result from a function, so such side effects are encoded as additional structure wrapping the pure result. Thus, where a pure function would return a value of type `a`, after introducing effects it instead returns a *computation* of type `f a`.

- If a function can fail, then `f = Maybe`.
- If a function reads a global environment of type `e`, then `f = e -> _`.
- If a function reads and updates state `s`, then `f = s -> (s, _)`.

The Haskell standard library provides several type classes for working with values of the form `f a`. They allow you to abstract away from the structure `f` and work with values of type `a` inside as if there were no additional structure.

The first such type class allows writing expressions over computations `f a`.

```haskell
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c

instance Applicative Maybe where
  pure :: a -> Maybe a
  pure = Just

  liftA2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
  liftA2 _ Nothing _ = Nothing
  liftA2 _ _ Nothing = Nothing
  liftA2 f (Just x) (Just y) = Just (f x y)
```

The second type class allows sequential composition of computations in an imperative style:

```haskell
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b

newtype State s a = State { runState :: s -> (s, a) }
```

```haskell
instance Monad (State s) where
  (>>=) :: State s a -> (a -> State s b) -> State s b
  m >>= k = State \s ->
    let (s', x) = runState m s in
    runState (k x) s'
```

Now, if we define the basic state operations, we can write code in an imperative style with effects.

```haskell
get :: State s s
get = State \s -> (s, s)

put :: s -> State s ()
put newS = State \oldS -> (newS, ())

example :: State Int Int
example =
  get >>= \x ->
  put 42 >>= \() ->
  get >>= \y ->
  pure (x + y)

ghci> runState example 1
43
```

For such monadic chains there is special syntactic sugar:

```haskell
example :: State Int Int
example = do
  x <- get
  put 42
  y <- get
  pure (x + y)
```

Exercise 8 *Implement `liftA3` using `liftA2`.*

Exercise 9 *Implement* `>>=` *using* `join` *and vice versa.*

Exercise 10 *Read two numbers from the console, divide one by the other with integer division, and print the result; if the remainder is non-zero, print it too.*

2 Parametric Polymorphism

No non-trivial property of programs can be checked algorithmically[5]. To remain decidable (in terms of type checking and/or inference), many type systems sacrifice completeness and, besides incorrect programs, reject many correct ones. At the same time, type systems also try to provide various capabilities that make it possible to type as many correct programs as possible. One of these is parametric polymorphism.

By **parametric polymorphism** we mean the ability of code to work uniformly with arbitrary data types Strachey [2000], Cardelli and Wegner [1985], which in many cases allows avoiding code duplication.

In this chapter we will look at how polymorphism is described in its simplest form — in typed λ-calculus. We will study different forms of parametric polymorphism and the associated safe programming techniques. We will analyze possible ways to implement parametric polymorphism efficiently. And finally, we will consider runtime representation polymorphism, "polymorphism over polymorphism."

2.1 Parametric polymorphism in the language

λ-abstraction allows generalizing expressions over values; each abstraction adds an arrow to the type of the expression, while application removes an arrow:

$$\frac{x : \tau, \Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau.\, M : \tau \to \sigma}\ Lam \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \sigma}\ App$$

At the same time, Λ-abstraction allows generalizing expressions over types by adding a quantifier to the type (a Π-abstraction), and term application to a type (**universal application**) allows choosing which concrete type this term is intended to use [Pierce, 2002, chapter 23]:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\, \tau}\ TLam \qquad \frac{\Gamma \vdash M : \forall\alpha.\, \tau}{\Gamma \vdash M\ \sigma : [\alpha \to \sigma]\ \tau}\ TApp$$

Thus, for example, the function id actually takes *two arguments: a type and a value*:

$$\begin{aligned} &id : \forall\alpha.\, \alpha \to \alpha \\ &id = \Lambda\alpha.\, \lambda x : \alpha.\, x \\ &id\ nat : nat \to nat \\ &id\ nat\ 42 : nat \end{aligned}$$

In Haskell, type abstractions and applications are attached implicitly by type inference. However, there are language extensions that allow writing them explicitly: TypeAbstractions, TypeApplications. This can help, for example, when the term does not provide enough information to infer the type. Thus, you can explicitly specialize `id` to the desired type:

[5] `https://en.wikipedia.org/wiki/Rice%27s_theorem`

```haskell
id :: forall a . a -> a
```
```
ghci> :t id @Int
id @Int :: Int -> Int
```

Quantifiers are also implicitly attached at the beginning of a type, following a naming convention: concrete types start with an uppercase letter, while polymorphic ones start with a lowercase letter. Similarly, the user may explicitly write `forall`s using the ExplicitForAll extension. This may be useful either to manually fix the order of type abstractions, or to refer to the abstracted type inside the body of a function (the ScopedTypeVariables extension).

Polymorphic data types are expressed by another construct. If previously we controlled the type at the term level by universal application, now we want to control it at the type level. For this we introduce λ abstraction in types, type application, and, accordingly, β-reduction. The kind system (for now) is just the simplest "type system for types"and ensures well-formedness of types and strong normalization[6]. For example, we can write the type of a pair, abstracted over the concrete types of its components, so that the user can pick the desired instantiation:

$$
\begin{aligned}
&Pair : * \to * \to * \\
&Pair = \lambda \tau^{*}\ \sigma^{*}.\, \forall \gamma.\, (\tau \to \sigma \to \gamma) \to \gamma \\
&pair : \forall \alpha\ \beta.\, \alpha \to \beta \to Pair\ \alpha\ \beta \\
&pair = \Lambda \alpha^{*}\ \beta^{*}.\, \lambda x^{\alpha}\ y^{\beta}.\, (\Lambda \gamma^{*}.\, \lambda f^{\alpha \to \beta \to \gamma}.\, f\ x\ y) \\
&fst : \forall \alpha\ \beta.\, Pair\ \alpha\ \beta \to \alpha \\
&fst = \Lambda \alpha^{*}\ \beta^{*}.\, \lambda p^{Pair\ \alpha\ \beta}.\, p\ \alpha\ (\mathbf{K}\ \alpha\ \beta)
\end{aligned}
$$

In Haskell, the computational semantics of polymorphic types can be traced in type synonyms:

```haskell
type Pair a b = forall c . (a -> b -> c) -> c
intPair :: Pair Int Int -- forall c . (Int -> Int -> c) -> c
```

Regular type constructors are nominal. For example, `(Int, Int)` or `Maybe Int` do not further compute.

Haskell does not allow writing functions at the type level in place using an explicit type lambda[7] due to the difficulty of this construct for type inference. However, named functions at the type level do exist, and we will examine them later 3.2. Scala has a non-trivial trick[8][9], which makes this possible. Scala 3, however, has added this capability directly to the language[10].

2.1.1 Emulating type abstractions and applications (`Proxy`)

In Haskell, extensions that allow manually writing type applications and abstractions appeared relatively recently[11]. Before that, the following technique was used.

[6] **Strong normalization** means any order of reductions leads to a normal form.
[7] `https://stackoverflow.com/questions/4069840/lambda-for-type-expressions-in-haskell`
[8] (stackoverflow) Scala type lambdas.
[9] `https://stackoverflow.com/questions/9443004/what-does-the-operator-mean-in-scala`
[10] `https://docs.scala-lang.org/scala3/reference/new-types/type-lambdas.html`
[11] TypeApplications, TypeAbstractions.

The standard library defines the type `Proxy` with one parameter. This is a **phantom type parameter** — values of the corresponding type are not stored in the data structure, it only allows placing additional information at the type level[12]. Consequently, the uninformative constant `Proxy` can be annotated with the desired type and passed to a function to specialize the type parameter to the required type. Or you can accept a `Proxy` and use ScopedTypeVariables for type signatures in patterns[13].

```
data Proxy a = Proxy

id :: Proxy a -> a -> a
ghci> :t id (Proxy :: Proxy Int)
id (Proxy :: Proxy Int) :: Int -> Int

id (Proxy :: Proxy a) x = (x :: a)
```

Sometimes the proxy type is left polymorphic so as not to depend on `Data.Proxy`. Instead of a concrete value, one may pass a specialized value `undefined`, and the receiver, not knowing the type, cannot force it (however, any occurrence of `undefined` in a term is too suspicious, so this is not a very good practice).

```
id :: proxy a -> a -> a
id (_ :: proxy a) x = (x :: a)

ghci> :t id (undefined :: Proxy Int)
id (undefined :: Proxy Int) :: Int -> Int
```

2.1.2 First-class polymorphism

There is the possibility to write functions that accept other polymorphic functions as arguments. The types of such functions are called **higher-rank types**, they can be used with the extension RankNTypes. Thus, the type parameter of function `g` is determined by the function `f`, not by the caller of `f`:

```
f :: (forall a . a -> a) -> (Int, Char)
f g = (g @Int 42, g @Char 'a') -- universal application for clarity
ghci> f (\x -> x)
```

The problem with higher-rank types is that their inference is undecidable, meaning Haskell's global type inference does not work in this case. But if higher-rank types are annotated manually, the rest of the inference works as before. For example, Church numerals have higher rank[14][15]:

[12] `https://wiki.haskell.org/Phantom_type`

[13] The type parameter actually has polymorphic kinds: `data Proxy (a :: k) = Proxy`, so that this technique works with types of arbitrary kinds (see also 2.1.5).

[14] `https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html`

[15] Anyone interested in programming languages is invited to spend more than a month of their life on Oleg Kiselyov's site: `https://okmij.org/ftp/README.html`.

```haskell
suc :: (forall a . (a -> a) -> a -> a) -> (a -> a) -> a -> a
suc n s z = s (n s z)
```

Exercise 11 *What rank does the type* `Int -> (forall a . a -> a)` *have?*

Many problems associated with higher-rank types can be avoided by wrapping them. For example, for Church numerals we can define a wrapper `newtype Church`. Now code that works with the wrapper can be typed with first-rank types; only the constructor has a higher-rank type:

```haskell
newtype Church = Church (forall a . (a -> a) -> a -> a)
(+) :: Church -> Church -> Church -- rank 1
```

Code like that can also be written in Java (Kotlin):

```kotlin
interface Church { fun <a> fold(s: (a) -> a, z: a): a }
fun plus(n: Church, m: Church): Church = object : Church {
    override fun <a> fold(s: (a) -> a, z: a): a = n.fold(s, m.fold(s, z))
}
```

By default, type parameters can only be instantiated to concrete types. The extension ImpredicativeTypes allows specializing type parameters to polymorphic types (including `forall`s inside them) — **impredicative application**.

```haskell
runST :: (forall s. ST s a) -> a
($) :: forall a b . (a -> b) -> a -> b
foo = runST $ ... -- only typechecks with ImpredicativeTypes
```

Higher-rank types can be used as type-based escape analysis, that is, to prevent the user from passing some value outside a certain scope. For example, Haskell provides the efficient `ST` monad, which within a restricted scope allows working with mutable memory cells Launchbury and Peyton Jones [1995][Maguire, a, 7.2, ST trick]:

```haskell
newtype ST s a = ST (IO a)
runST :: (forall s. ST s a) -> a

sumTo :: Int -> Int
sumTo n = runST do
  ref <- newSTRef 0
  forM [0..n] \i -> modifySTRef ref (+ i)
  readSTRef ref
```

Notice that if you try to return a reference to a mutable cell from `runST`, the resulting type will not pass the well-formedness check because it would contain the phantom parameter `s` that is not bound anywhere:

```haskell
newSTRef :: a -> ST s (Ref s a)
ghci> runST (newSTRef 0) :: Ref s Int -- error
```

In practice, to distinguish such locally bound type variables, one uses the concept of levels[16] Jones [2019].

Higher-rank types together with impredicative application form **first-class polymorphism**, when polymorphic types can be used almost as freely as any other. The classical Hindley-Milner global inference algorithm cannot handle this (and in general the problem is undecidable), so there is a variety of solutions that make different tradeoffs. You can make inference local, relying only on nearby AST nodes and auxiliary type annotations Pierce and Turner [2000], Christiansen [2013], Dunfield and Krishnaswami [2019]. Or you can try to help global inference with additional preprocessing (Quick Look[17] Serrano et al. [2020], recently implemented in Haskell) or with additional controlling constructs (FreezeML Emrich et al. [2020]).

2.1.3 Higher-order/kinded polymorphism

Haskell also allows abstraction over types of arbitrary kinds, not only `Type`, both in `data` declarations (**higher-order/kinded types (HKT)**[18]) and in polymorphic functions. We will encounter many examples later. For example, `Fix` has kind `(Type -> Type) -> Type`, and the catamorphism is abstracted over the arrow kind:

```haskell
newtype Fix f = Fix (f (Fix f))
cata :: forall (f :: Type -> Type) a . Functor f => (f a -> a) -> Fix f -> a
```

Later we will study a technique that allows encoding higher-order kinds in a language that does not support them (see 3.1.8).

2.1.4 Generalized algebraic data types (GADTs)

Generalized algebraic data types (GADTs) allow attaching more information to data at the type level. As a model example, consider the syntax of a tiny programming language. Our goal is to prevent the construction of ill-typed syntax trees in Haskell.

```haskell
data Expr = Const Int | IsZero Expr | If Expr Expr Expr
```

As we know, data constructors in Haskell are ordinary functions, except that their implementation is generated by the compiler (allocation, field layout, ...). Functions have types. For example, `IsZero :: Expr -> Expr`.

Haskell has syntax for defining `data` by explicitly listing constructor types[19]. It is completely analogous to the form we saw earlier, only much more convenient for complex data structures. The previously considered term type `Expr` looks like this:

[16] https://okmij.org/ftp/ML/generalization.html
[17] (youtube) A Quick Look at Impredicativity (Simon Peyton Jones)
[18] https://serokell.io/blog/kinds-and-hkts-in-haskell
[19] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/gadt_syntax.html#gadt-style

```haskell
data Expr where
  Const :: Int -> Expr
  IsZero :: Expr -> Expr
  If :: Expr -> Expr -> Expr -> Expr
```

For polymorphic data structures, the list example uses the following syntax. The name `elem` is needed only for documentation and cannot be used otherwise; it only marks the presence of a type parameter and allows giving it a kind manually[20].

```haskell
data List (elem :: Type) where
  Nil :: List a
  Cons :: a -> List a -> List a
```

Let us add a phantom type parameter `ty` to `Expr`, indicating the Haskell type into which the expression should be interpreted, and use a GADT to assign concrete `ty` values to constructor result types. Thus we state that a program built with `Const` evaluates to an integer, `IsZero` evaluates to a boolean, and a conditional evaluates to the type of its branches.

```haskell
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
```

Now we can write a safe typed interpreter. Note that when pattern matching on constructors, the information about the type parameter is refined:[21]

```haskell
eval :: Expr ty -> ty
eval = \case
  Const x  -> x           -- ty                    sim Int
  IsZero t -> eval t == 0 -- ty                   sim Bool
  If c t e -> if eval c then eval t else eval e
```

Next we will examine how GADTs in Haskell are expressed through more basic language mechanisms 3.4.4.

2.1.5 Type-level structures, data promotion

To gain greater control over program correctness, let us learn to encode arbitrary data structures at the type level. As a model task, we will define a vector with length information.

First, define Peano-style natural numbers at the type level:

[20] The kind may be omitted. Alternatively, you can omit parameter names and simply write `data List :: Type -> Type where ...`.

[21] This uses the convenient LambdaCase extension, which avoids introducing extra names.

```haskell
data Zero
data Suc n
```

Exercise 12 *How many inhabitants does the type* `Suc (Suc Zero)` *have?*

Now we can define a vector type that carries length information:

```haskell
data Vec (size :: Type) (elem :: Type) where
  VNil :: Vec Zero a
  VCons :: a -> Vec n a -> Vec (Suc n) a

example :: Vec (Suc (Suc Zero)) Int
example = VCons 1 (VCons 2 VNil)
```

For such a type, for example, one can write a safe `vzip` function that only works on vectors of the same length:

```haskell
vzip :: Vec n a -> Vec n b -> Vec n (a, b)
vzip VNil VNil = VNil                                        -- n   sim Zero
vzip (VCons x xs) (VCons y ys) = VCons (x, y) (vzip xs ys) -- n  sim Suc n'
```

Note that in the other branches of `vzip`, equalities must arise that begin with different constructors, for example `Zero sim Suc n`. Since it is impossible to construct such function arguments, Haskell allows the corresponding branches to be omitted.

Exercise 13 *Write a function that appends an element to the end of a vector. Proceed step by step, filling in type holes and tracking the equalities that arise.*

Currently the kind language that should control types is too poor. Indeed, the kind `Suc` is `Suc :: Type -> Type`, so nothing prevents writing, for example, `Suc (Maybe Int)`. At the same time it is too restrictive because it does not support polymorphism, which led to a lot of duplication like `Typeable (ty :: Type)`, `Typeable1 (ty :: Type -> Type)`, ...

Modern Haskell has the TypeData extension, which allows declaring new types and kinds similarly to how `data` allows declaring new types.

```haskell
type data Nat = Zero | Suc Nat
```

Now the vector can be given a more precise kind:

```haskell
data Vec (size :: Nat) (elem :: Type) where
  VNil :: Vec Zero a
  VCons :: a -> Vec n a -> Vec (Suc n) a
```

Exercise 14 *What does* `ghci> :k Vec` *print?*

Another way to achieve the same result is using DataKinds Yorgey et al. [2012]. This extension automatically promotes all `data` declarations one level higher. Specifically: any type constructor also becomes a kind, and any data constructor becomes a type constructor. Thus, in the example with natural numbers, we can declare them normally and use them at the type level:

```
data Nat = Zero | Suc Nat
ghci> :k Suc :: Nat -> Nat -- here it is clear that Suc is used as a type
```

Because types and terms live in different namespaces in Haskell, one can name type constructors and data constructors identically. However, if such a data type is promoted, ambiguity arises: do we mean the type or the promoted constructor? Haskell allows explicitly indicating that we mean the promoted constructor with a leading apostrophe.

```
data T = T Nat
ghci> :k T
T :: Type       -- about the type constructor
ghci> :k 'T
'T :: Nat -> T -- about the promoted data constructor
```

Not every `data` declaration is suitable for promotion, while `type data` declarations allow explicitly requesting a type-level structure and getting sensible errors if the declaration is written incorrectly.
In the case of promoting a polymorphic type, we obtain polymorphic kinds (PolyKinds):

```
data [a] = [] | (:) a [a]
ghci> :k '(:)
'(:) :: forall k . k -> [k] -> [k]
```

Term	Type	Kind
`Zero`	`Nat`	`Type`
`[Zero, Suc Zero]`	`[Nat]`	`Type`
`[]`	`forall a. [a]`	`Type`
`(:)`	`forall a. a -> [a] -> [a]`	`Type`
	`'Suc 'Zero`	`Nat`
	`'['Zero, 'Suc 'Zero]`	`[Nat]`
	`'[Int, Double]`	`[Type]`
	`'[]`	`forall k. [k]`
	`'(:)`	`forall k. k -> [k] -> [k]`

Figure 5: An example of promotions in Haskell.

Examples of promotions for various constructions can be seen in Table 5.
As an example, let us define a heterogeneous list indexed by the types of its elements:

```haskell
data HList (tys :: [Type]) where
  HNil :: HList '[]
  HCons :: ty -> HList tys -> HList (ty ': tys)

example :: HList '[Int, Bool, Double]
example = HCons 42 $ HCons True $ HCons 12.5 HNil
```

Data structures can also be kind-polymorphic. Consider the type `Tagged`, which allows attaching an additional type tag to the value type. The kind of the tag may be arbitrary, so for example we can use type-level constants from TypeLits (another example of polykind usage we saw earlier in 2.1.1):

```haskell
newtype Tagged (tag :: k) (a :: Type) = Tagged a
ghci> :t Tagged
Tagged :: forall k (tag :: k) a. a -> Tagged tag a

example :: Tagged ("dbId" :: Symbol) Int
example = Tagged 42
```

Modern Haskell has ultimately arrived at the point where the type system does not distinguish between types and kinds (Fig. 6). In particular, `Type :: Type`. This is required to extend Haskell in the direction of dependent-type programming by adding non-syntactic equivalences for kinds (TypeInType). *System FC* was presented in the work Weirich et al. [2013][22][23].

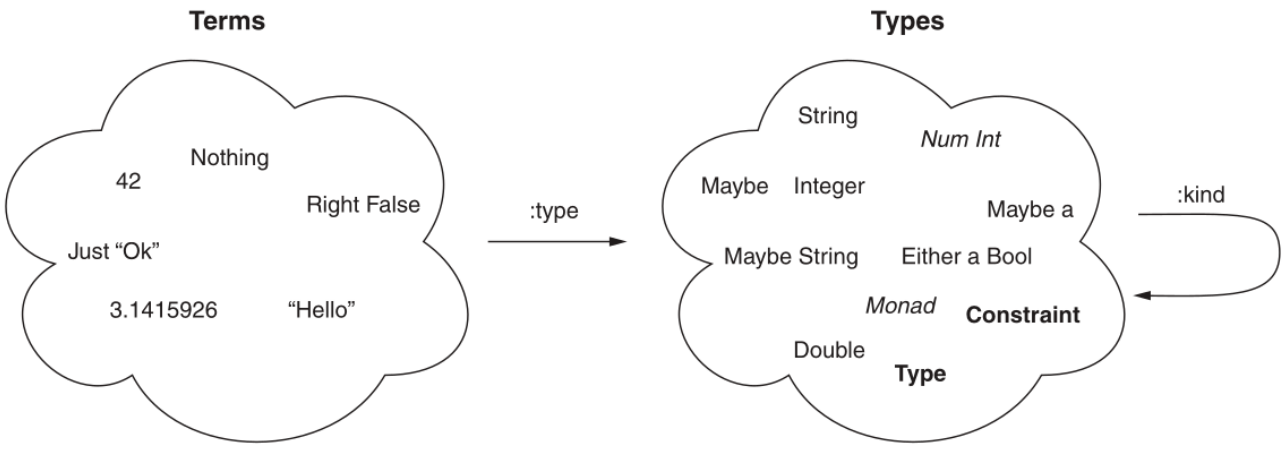


Figure 6: Types and kinds are the same thing (Bragilevsky).

[22](youtube) Mini-course in Russian on the development of Haskell toward dependent typing.
[23](youtube) Mini-course in Russian — Haskell type inference system.

2.2 Implementation of parametric polymorphism

A **calling convention**[24] is a set of agreements between how a function is compiled and how it must be called. For example, a function takes two word-sized arguments and returns one word-sized result. Then the generated low-level code may, for example, expect both arguments to be passed through a specific pair of registers and place the result in a third register. In that case, the caller must provide the arguments in the correct registers and expect the result in a predetermined register.

In general, the calling convention of a function depends on the types of its arguments and result. You need to know at least their size to decide whether to place them in registers or on the stack. You need to know whether it is a pointer (**reference type**) or a value itself (**value type**) to understand how to handle it. For data structures, you need to know field offsets.

Thus, implementing parametric polymorphism in a language is not a trivial task. Different languages use different approaches, each with its own strengths and weaknesses.

2.2.1 Monomorphization

Monomorphization is the most straightforward approach: compile polymorphic functions and structures for each set of type arguments. Thus, if there are 100 different sets of type arguments with which a function is called, then its code will be compiled 100 times and occupy 100 times more space in the binary. This is what C++ and Rust do, for example.

In fact it is even worse. If the project is multi-module and consists of many compilation units (pieces that are compiled separately), the same specialization of the function for type arguments will be compiled again in every compilation unit where it is needed. Then the linker will remove duplicates, which is also not the fastest or most efficient process.

+ Generated code is maximally efficient for each type;
+ `is`-style runtime type checks are easy to implement at compile time (in other approaches this is difficult);
− Compilation time becomes very large;
− The resulting binary size increases significantly, which may be critical for some applications;
− It may work inefficiently due to instruction cache pollution;
− Polymorphic methods cannot exist in interfaces, because at the call site we do not know which concrete implementation should be specialized (similarly, higher-rank polymorphism does not work);
− Polymorphic functions cannot be dynamically linked in general (they have no code before specialization);
− In general, variance cannot be supported because the code is compiled for a concrete type and cannot in general work for an arbitrary subtype or supertype (if reference and value types may live in the same subtyping hierarchy).

[24] `https://en.wikipedia.org/wiki/Calling_convention`

Some languages (for example, C++ and Zig) do not treat instantiation as a hidden implementation detail of the language, but provide it as a tool for users. This provides the following possibilities:

- If values are allowed in types, instantiation can be used as a compile-time computation mechanism.
- If error checking is deferred to instantiation time, we get a kind of static duck typing. This allows not describing complex signatures of polymorphic functions. However, then functions for testing must be manually instantiated against all possible types, otherwise it is impossible to determine statically whether they compile for those types.

2.2.2 Type erasure

One can do the opposite: unify the values that arrive at polymorphic functions and are stored in polymorphic data structures, instead of compiling code for each type.

Suppose each value is allocated on the heap and passed by pointer. Then we can reuse the same code for different type arguments — it will simply expect pointers.

+ Each function is compiled exactly once — fast;
+ It is possible to dynamically load new polymorphic functions and types and use them together;
+ Flexibility — variance, polymorphic methods in interfaces, higher-rank types, etc. just work;
− Heap allocation and pointer dereferencing can significantly slow down code;
− Since type information is erased, one cannot do anything with a type argument without having its inhabitants (for example, query reflection information or perform an `is` check).

This approach is followed by the JVM, Haskell, and generally other functional languages due to its flexibility and compile-time speed.

The special problem is working with primitives and other value types, because each value has to be boxed first and then used in a polymorphic context. Therefore languages fight this problem as best they can. Some languages reduce the range of primitive values to reserve a bit that indicates whether it is a pointer or a value. Code consults this bit to operate (similar to 2.2.4). This is done, for example, by OCaml and Koka. Aggressive inlining together with other optimizations also helps Blanchet [1999]. Java, for example, is also trying to move carefully toward the possibility of monomorphization[25][26].

2.2.3 Hybrid approach

C# implements a hybrid approach[27]. It distinguishes values stored on the heap — reference types — from values stored on the stack — value types. For the former it generates one specialization that works with pointers. For each set of value types it lazily generates specializations at runtime.

Thus, traces of generics exist both in the CIL intermediate representation and in the runtime.

[25] Type Specialization of Java Generics - What If Casts Have Teeth?

[26] `https://cr.openjdk.org/~jrose/values/parametric-vm.html`

[27] Generics in the runtime (C# programming guide).

```
struct value_witness_table {
  size_t size, align;
  void (*copy_init)(opaque *dst, const opaque *src, type *T);
  void (*copy_assign)(opaque *dst, const opaque *src, type *T);
  void (*move_init)(opaque *dst, opaque *src, type *T);
  void (*move_assign)(opaque *dst, opaque *src, type *T);
  void (*destroy)(opaque *val, type *T);
};
```

Figure 7: Swift value witness table.

Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Figure 8: Code generated by the Swift compiler for a polymorphic function.

+ Value types are stored and passed as-is without boxing;
+ Reflection over generics is available;
+ Compilation time is small;
− Runtime instantiation slows execution;
− Variance works only for reference types (which is odd — there is “correct” subtyping and “incorrect” subtyping).

2.2.4 Using a value witness table for types

Swift[28] passes with each type parameter a value witness table (Fig. 7). This is a table with all the necessary information about the type: size and alignment, what to do when copying and moving the object (for example, incrementing a reference count). Thus, the generated code constantly consults this table and makes virtual calls to functions from it (Fig. 8).

+ Compilation time is small;
+ The performance behavior is predictable (it does not lead to surprising runtime pauses);
+ Efficient work with value types;
+ High flexibility;
+ Type information is not erased;

[28](youtube) 2017 LLVM Developers’ Meeting: "Implementing Swift Generics"

– A significant constant overhead from virtual calls through the table, and performance strongly depends on compiler optimizations.

This is a kind of implementation of parametric polymorphism through a special ...

2.3 Representation polymorphism

As we already discussed above 2.2.2, parametric polymorphism in Haskell is implemented as follows: all values are stored on the heap and passed to polymorphic functions by pointer. However, if performance is important for computational code, that approach is not suitable because of the heavy load on the memory management subsystem and many indirections. Therefore Haskell also allows writing code using unboxed values. And if the calling convention is not essential, one can abstract over it and write one code path for both boxed and unboxed values Eisenberg and Peyton Jones [2017].

2.3.1 Varieties of runtime representations in Haskell

	Boxed	Unboxed
Lifted	*Int* *Bool*	
Unlifted	$ByteArray_{\#}$	$Int_{\#}$ $Char_{\#}$

Figure 9: Kinds of values in Haskell with examples Eisenberg and Peyton Jones [2017].

Figure 9 shows the classification of values in Haskell with examples of types. **Unboxed types** are held and passed by value. **Boxed types**, conversely, are passed by pointer and stored on the heap. The ordinary `Int` is just a declaration of the following form, where `I#` is an ordinary constructor with an unusual name that contains an unboxed value.

```
1 data Int = I# Int#
```

Lifted types may contain $\perp$ as a value. In other words, they may represent suspended computations (the compiler and runtime provide this specially). **Unlifted types**, in contrast, cannot be suspended. Operations producing values of unlifted types are always strict. The property lifted/unlifted is called **levity**. To extend the exposition to eager languages, one can replace levity with boxity and everything remains valid.

A hash sign in type and function names is a convention that indicates that unlifted values are involved nearby[29].

[29] You need to enable the MagicHash extension to use # in identifiers.

Haskell also has unboxed tuples, which do not exist as runtime objects. For example, the following function appears to return a pair of values, but in fact the compiler may place them, for example, in a pair of registers. Accordingly, pattern matching on such tuples simply allows referring to each of those values.

```
divMod# :: Int -> Int -> (# Int, Int #)
case divMod# n k of (# quot, rem #) -> ...
```

Consequently, there is no distinction between differently nested unboxed tuples:

```
(# A, (# B, C #) #) ≡ (# (# A, B #), C #) ≡ (# A, B, C #)
```

2.3.2 Classification of values by runtime representation

Values of different types may be arranged differently at runtime. That is, we need some system for classifying types. But such a system already exists in Haskell — kinds. We describe the domain as data structures and then promote it to the required level using DataKinds 2.1.5.

The standard Haskell library provides the following data types:

```
TYPE :: RuntimeRep -> Type

data Levity = Lifted | Unlifted

data RuntimeRep = BoxedRep Levity
                | IntRep | DoubleRep
                | TupleRep [RuntimeRep]
                | SumRep [RuntimeRep]
                | ...

type LiftedRep = BoxedRep Lifted

type Type = TYPE LiftedRep
```

`TYPE` is a magic type defined in the compiler. It is parameterized by the runtime representation of values. Now the familiar `Type` is simply a special case with boxed lifted values.

- `Int :: TYPE (BoxedRep Lifted)` or `:: Type`
- `IntRep` and `DoubleRep` correspond to the runtime representation of numeric constants (depending on the processor architecture, integers and floating-point numbers may need to be placed in different special registers)
  `Int# :: TYPE IntRep`
- `Maybe Int :: Type`
- `Maybe :: Type -> Type`

- `TupleRep` and `SumRep` are unboxed algebraic representations, parameterized by the representations of their stored values
  `(# Int, Bool #) :: TYPE (TupleRep '[LiftedRep, LiftedRep])`
- For simplicity, nested tuple types are not unified

```
(# Int#, (# Int, Double# #) #)
  :: TYPE (TupleRep '[IntRep, TupleRep '[LiftedRep, DoubleRep]])
```

2.3.3 Representation polymorphism

Having exposed the runtime representation in the kind structure, we can now parameterize over it. For example, the kind of the function arrow looks as follows[30]:

```
ghci> :k (->)
(->) :: forall {q :: RuntimeRep} {r :: RuntimeRep}. TYPE q -> TYPE r -> Type
```

Exercise 15 *Think about why the function has a boxed type. Can it be different? Could that be useful?*

Unfortunately, Haskell imposes a rather strict restriction: binders cannot have a type that is polymorphic over runtime representation. One can easily guess why — it is impossible to generate code for a function that works with a parameter of an arbitrary runtime representation. This can only be solved by monomorphization 2.2.1, but Haskell avoids that approach[31]. The community is also trying to find other solutions[32] (something like 2.2.4).

For example, initially the application operator was generalized only over the result type. This does not cause problems, because the caller can infer the representation and generate the appropriate code:

```
($) :: forall r a (b :: TYPE r). (a -> b) -> a -> b
f $ x = f x
```

However, it was noticed that for the application operator one can obtain another implementation that does not use a levity-polymorphic binder[33]:

```
($) :: forall ra rb (a :: TYPE ra) (b :: TYPE rb). (a -> b) -> a -> b
($) f = f
```

Thus, in Haskell representation polymorphism is somewhat degenerate and helps only in a small number of cases, but not unimportant ones. If one allows monomorphization over `RuntimeRep` parameters, one gets a system similar to the hybrid implementation of parametric polymorphism 2.2.3, only with greater programmer control over monomorphization.

[30] Disable simplifications: `ghci> :set -fprint-explicit-foralls -fprint-explicit-runtime-reps`
[31] https://gitlab.haskell.org/ghc/ghc/-/issues/14917
[32] https://mail.haskell.org/pipermail/haskell-cafe/2023-January/135770.html
[33] https://gitlab.haskell.org/ghc/ghc/-/merge_requests/10131

3 Ad-hoc Polymorphism

Once, Joe Fasel in a conversation with Philip Wadler expressed the idea that function overloading should be reflected in types. Wadler misunderstood him Hudak et al. [2007]. But what he understood turned out to be type classes Wadler and Blott [1989].

Christopher Strachey introduced a classification of polymorphism into two categories Strachey [2000]. Parametric — the same code works with data of different types. **Ad-hoc polymorphism** — code is selected depending on the type. For example, the same multiplication symbol acts differently on integers and on floating-point numbers.

Overloading in languages refers to the ability to give the same name to several functions with different sets of input parameters. At the call site, the compiler statically determines by the argument types which one should actually be called.

```
string toString(x: int) { ... }
string toString(fmt: String, d: double) { ... }
```

Type classes require first declaring a named entity (the type class itself) that includes a set of function declarations that can be overloaded for different types.

```
class Show a where
  show :: a -> String

instance Show Int where
  show :: Int -> String
  show = ...
```

It is important to note that a type class declaration contains a formal type parameter, and by its occurrences in the function type, the overloading is selected. There can be many such parameters (MultiParamTypeClasses), and they can have arrow kinds. For example, in the case of the type class `Applicative`, the choice of implementation for the operation `pure` happens based on the type constructor of the result, i.e. not even on a full type.

```
class Functor f => Applicative (f :: Type -> Type) where
  pure :: a -> f a
  ...

instance Applicative Maybe where
  pure :: a -> Maybe a
  ...
```

Also, unlike overloading, type classes are compatible with parametric polymorphism. In the type of a polymorphic function, one cannot specify that a certain overloading must exist for a type. Type classes, however, allow constraining the set of possible type arguments to those for which an instance of the required type class is implemented:

```
showPrefixed :: Show a => a -> String -> String
```

Comparing type classes with overriding in OOP languages, the resolution of a virtual function call uses a table stored in the object of the first parameter (the call receiver). Type classes, on the other hand, rely exclusively on the type, so, for example, it is possible to define constants in type classes[34]:

```
class Enum a => Bounded a where
  minBound :: a
  maxBound :: a
```

At the same time, type classes are not types, but rather predicates on types. A type satisfies such a predicate, or property, if there is a corresponding instance for it. Therefore, in particular, the familiar OOP way of creating a heterogeneous collection of elements with a common interface does not work directly with type classes. For example, this type would not be valid: `[Show]`. We will return to this problem in 3.4.2.

3.1 Type classes in the language

Despite its remarkable power, the idea of implementing type classes is extremely simple. It was already fully presented in the original paper Wadler and Blott [1989]. Subsequent work refined the type inference mechanism by reducing it to the classical Hindley-Milner type system Hall et al. [1996]. Other work mostly proposes a huge variety of extensions and applications Jones et al. [1997].

3.1.1 Dictionaries

Let us consider the idea of implementing type classes using the example of polymorphic sorting. Writing a sort for a list of elements of a specific type is trivial:

```
sort :: [Int] -> [Int]
sort = \case [] -> []; x:xs -> insert x (sort xs)
  where
    insert x xs = let (l, r) = List.partition (<) x) xs in l ++ x : r
```

In the implementation, the only type-specific information we use is the ordering on its inhabitants. Thus, when moving to polymorphic sorting, we need to accept a dictionary with predicates defining the required ordering for the given type.

```
data OrdDict a = OrdDict { less :: a -> a -> Bool }

sort :: OrdDict a -> [a] -> [a]
```

[34] Modern OOP languages, however, strive to support static functions in interfaces, which makes them closer to type classes and allows doing similar things. For example, Swift.

```
sort d@OrdDict{ less } = \case [] -> []; x:xs -> insert x (sort d xs)
  where
    insert x xs = let (l, r) = List.partition (`less` x) xs in l ++ x : r
```

Now, to use sorting on a list of numbers, we need to construct the appropriate record and call the function with it on a list of specific types:

```
intOrd :: OrdDict Int
intOrd = OrdDict { less = (<) }
```

```
ghci> sort intOrd [3, 2, 1]
```

There can be a situation where an instance for one type depends on an instance for another. For example, ordering on lists can be derived automatically, knowing the ordering on elements. In the case of dictionaries, we model this as a function between dictionaries:

```
listDict :: OrdDict a -> OrdDict [a]
listDict d = OrdDict { less = ... less d ... }
```

Now we can sort a list of lists by constructing the appropriate dictionary:

```
ghci> sort (listDict intOrd) [[3, 2], [2, 1], [0]]
```

A comparison of explicit dictionary passing and type classes can be seen in the following table:

1. Definition of a dictionary of functions

```
data MyOrd a = MyOrd
  { less :: a -> a -> Bool }
```

2. Dictionary instance for a specific type
   - Named value

```
intMyOrd :: MyOrd Int
intMyOrd = MyOrd { less = (<) }
```

3. Explicit function parameter

```
sort :: MyOrd a -> [a] -> [a]
```

4. Passed by the user

```
test = sort intMyOrd [3, 2, 1]
```

1. Definition of a type class

```
class MyOrd a where
  less :: a -> a -> Bool
```

2. Declaring a type as a member of a type class
   - Has no name

```
instance MyOrd Int where
  less = (<)
```

3. Implicit function parameter

```
sort :: MyOrd a => [a] -> [a]
```

4. Passed by the compiler

```
test = sort [3, 2, 1]
```

Thus, a dictionary is a **witness** or proof that the type satisfies the constraint.

Exercise 16 *What dictionary would correspond to the higher-kinded type class* `Functor`*?*

3.1.2 Implicit arguments

One can think of it as follows: to the left of `=>`, implicit function arguments are passed, inferred by the compiler from the context. That is, for example, one should not be surprised by occurrences of `=>` in the type of an argument — it is simply a function with an implicit argument. Thus, the following code will not compile because at the use site of variable `y` there is no value of type `Show b`:

```
f :: (Show b => b) -> b
f x = x -- error
```

We can accept this value in function `f`, and then it will automatically propagate to `x`:

```
f :: Show b => (Show b => b) -> b
f x = x
```

The extension ImplicitParams provides the ability to make some function arguments implicit. In fact, this is an implementation of dynamic binding in a static language Lewis et al. [2000] (see 5.3.1). Implicit arguments are taken from the scope by name and substituted automatically:

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]

sort :: (?cmp :: a -> a -> Bool) => [a] -> [a]
sort = sortBy ?cmp
```

Haskell also provides the ability to store dictionaries in data structures:

```
data ShowDict a where
  ShowDict :: Show a => ShowDict a

f :: ShowDict b -> (Show b => b) -> b
f d x = case d of ShowDict -> x -- the Show b instance is available in scope
```

Exercise 17 *Is exactly this semantics possible in a strict language? Why?*

3.1.3 Instance inference

To call a constrained-polymorphic function, GHC performs instance inference, or in other words, automatically constructs witnesses. Instance inference is tightly integrated with Haskell's overall type inference system Peyton Jones [2019].

In reality, instance inference is nothing other than *the type inhabitation problem*. Indeed, after translation to Core (the intermediate representation in GHC), type classes are represented as dictionaries of functions. We have in context specific dictionaries and functions that allow deriving new dictionaries from existing ones. We need to find a term that constructs a dictionary of the required type.

Suppose, for example, that inside function `f :: Show a => ..` there is a call to a constrained-polymorphic function `g :: Show [a] -> ...` That is, we have a dictionary `d1 :: ShowDict a`,

as well as a function `d2 :: ShowDict a -> ShowDict [a]`, coming from imports[35]. We need to construct a term of type `ShowDict [a]`. Obviously, it is simply the application of one to the other: `d2 d1`.

Instance inference proceeds recursively. To infer `ShowDict [a]`, the premise `ShowDict a` is first inferred. That is, we get recursion on the structure of the type. In other words, instance inference can be exploited as a type-level computational primitive. For example, we can lower information from types into terms (analogous to GHC.TypeLits):

```
type data Nat = Zero | Suc Nat

class KnownNat (n :: Nat) where
  natVal :: Int

instance KnownNat Zero where
  natVal = 0

instance KnownNat n => KnownNat (Suc n) where
  natVal = 1 + natVal @n

ghci> natVal @(Suc (Suc Zero))
-- will be inferred as natVal {knownSuc (knownSuc knownZero)}
```

In the general case, the type inhabitation process, as one might expect by computational analogy, is undecidable. Therefore GHC imposes many restrictions on the form of instances that guarantee totality of inference. These restrictions are described in detail in Sulzmann et al. [2007a]. GHC also provides various extensions that relax these restrictions, shifting part of the responsibility to the programmer[36]. For example, with UndecidableInstances one can easily write type-level list reversal at compile time, as well as any other function:

```
class Reverse (acc :: [Type]) (tys :: [Type]) where
  showReverse :: String

instance ShowT acc => Reverse acc '[] where
  showReverse = showTypes @acc

instance Reverse (ty : acc) tys => Reverse acc (ty : tys) where
  showReverse = showReverse @(ty : acc) @tys

ghci> showReverse @'[] @'[Char, Int, Double]
```

One can notice that the process of type class inference is very similar to the evaluation of logic programs, for example in Prolog, only without backtracking (trying different solution variants in

[35] Instances can be imported with an empty import: `import Module ()`.

[36] `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/instances.html`

search of a suitable one). As, indeed, type inference in Haskell Peyton Jones [2019] in general: the equivalences collected across the program can be viewed as a logic program, and solving this system of type equations as executing that program.

There is an interesting synergy between type classes and type inference (Fig. 10)[37]. Based on terms, types are inferred. Then, based on types, type class instances are inferred. That is, we write some interesting intellectual code, and in parallel the compiler writes out the uninteresting code.

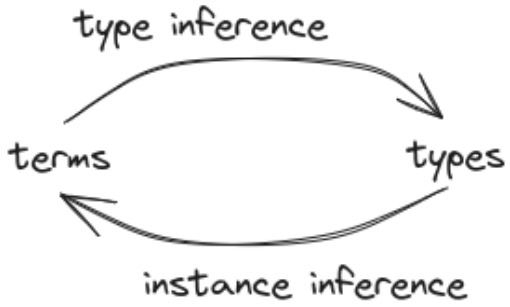


Figure 10: Type classes + instance inference = code generation.

Instance inference relies only on the form of the "head" of the declaration — to the right of `=>`, while constraints on the left are applied after the fact. This can be used to write more general instances. For example, this is how the **constraint trick**[38] works, allowing one to resolve ad-hoc polymorphic functions in a parametrically-polymorphic context.

3.1.4 Constructing a type from a value

After we have learned to lower values from types, it is natural to learn the reverse — lifting values into types. We use the technique described in Kiselyov and Shan [2004]. There is a corresponding library Data.Reflection (see 3.4.7).

In reality, we of course cannot honestly obtain a syntactically correct type of the needed size, simply because types exist strictly before the execution stage. However, as we know, type class dictionaries have a runtime representation (the case of polymorphic recursion is precisely an example where this cannot be entirely avoided). Therefore, we use continuation passing style, which will be examined in detail later in chapter 6: instead of returning a result, we accept a continuation that can work with any type with `KnownNat` (we use type abstractions and applications, see 2.1)[39]:

```
reify :: Int -> (forall n. KnownNat n => a) -> a
reify n k
  | n <= 0 = k @Zero
  | otherwise = reify (n - 1) \@n' -> k @(Suc n')
```

[37] (youtube) Hackett: a metaprogrammable Haskell.

[38] `https://chrisdone.com/posts/haskell-constraint-trick/`

[39] In not the most recent versions of GHC, you will need to use the `Proxy` technique from 2.1.1.

The continuation passed to the recursive call captures the dictionary for type `n` and constructs a dictionary for `Suc n`.

Finally, we can write the following surprising identity function that first lifts a value into a type and then lowers the type back into terms:

```
wonderId :: Int -> Int
wonderId n = reify n (\@t -> natVal @t)
```

3.1.5 Implicits and coherence

Type classes need not be a special language mechanism; instead, one can provide at the language level implicit parameters and inhabitation sufficient to implement type classes.

Thus, in Scala there is the mechanism of implicits Křikava et al. [2019][40]. Function parameters can be marked with the keyword `implicit`, and Scala will try to infer them automatically using the `implicit` declarations available in scope. Declarations of variables, functions, and object constructors can also be marked `implicit`, and they will be used during inhabitation. Now we can model a dictionary of functions, for example using interfaces (which in Scala are called `trait`) and OOP singletons, to obtain type classes Oliveira et al. [2010]:

```
// A bundle of functions.
trait Show[T] {
    def show(x: T): String
}

// Wrapper for convenient calling.
def show[T](x: T)(implicit ev: Show[T]): String = ev.show(x)

// Singleton object, a value for the bundle of functions.
implicit object intShow extends Show[Int] {
    def show(x: Int): String = x.toString
}

def showAll[T](xs: List[T])(implicit ev: Show[T]): String =
    xs.map(show(_)).join(", ")
```

As we discussed earlier (2.1), working with type parameters is similar to working with ordinary ones. Also, type inference can be viewed as the process of recovering omitted type applications. Note that the implicit inference mechanism does approximately the same thing. Therefore, when trying to make functions with implicits first-class functions, difficulties similar to those of first-class polymorphism arise (2.1.2).

In dependently typed languages, implicit parameters[41] are especially needed because, for example, types are exactly the same parameters of a function as all others. Therefore, type inference is

[40] The design of implicit parameters in Scala 3 has changed (youtube) Scala Implicits Revisited, Martin Odersky.

[41] `https://agda.readthedocs.io/en/v2.7.0.1/language/implicit-arguments.html`

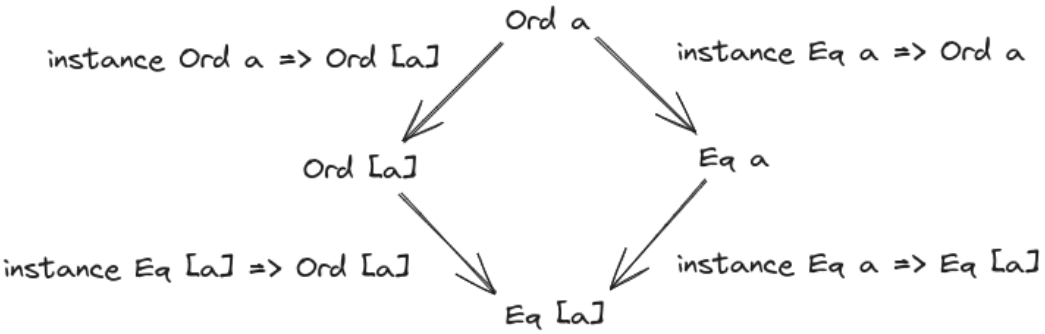


Figure 11: Instance coherence — the diagram commutes.

essentially the inference of implicit function arguments. Moreover, dependent functions, along with arguments, often accept proofs of some properties of those arguments, which one also wants to infer from context automatically where possible. Such an inference mechanism can be reused for emulating type classes[42] Devriese and Piessens [2011]. It also works in the reverse direction — one can emulate dependent typing mechanisms using type classes McBride [2002].

As we will see later, implicit parameters on their own are also needed as a static approximation of dynamic free variables for implementing effect systems (see 8). However, having type classes as a separate language feature is still useful, despite them seemingly being the same implicit parameters (albeit with a recursive inhabitation mechanism). One can support an important property when all restrictions are respected, i.e. in the absence of **orphan instances**[43]. **Instance coherence** — for a given type, all instances of a given type class obtained by different means are indistinguishable (Fig. 11). Accordingly, the origin of any particular instance does not matter. In other words, one need not think about this, which removes a significant amount of cognitive load and simplifies refactoring[44]. In contrast, other approaches require careful attention to the calling context, because an unexpected implementation may come from it.

3.1.6 Rules and specialization

GHC allows specifying optimizing rewrite rules for the compiler directly in code using a special pragma[45] Jones et al. [2001]. For example:

```
{-# RULES
  "map/map"    forall f g xs.  map f (map g xs) = map (f . g) xs
  "map/append" forall f xs ys. map f (xs ++ ys) = map f xs ++ map f ys
 #-}
```

[42] https://agda.readthedocs.io/en/v2.7.0.1/language/instance-arguments.html
[43] https://stackoverflow.com/questions/3079537/orphaned-instances-in-haskell
[44] Edward Kmett - Type Classes vs. the World.
[45] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/rewrite_rules.html

The first law is nothing other than the functor law. Ideally, we formulate laws at the design stage Maguire [b], verify their correctness using property-based testing[46], and then use them for optimizations.

One can rewrite a polymorphic version of a function to a specialized one when the types match. For this, one needs to implement the specialized version (semantic equivalence is entirely the programmer's responsibility) and set up the corresponding rewrite rule:

```
genericLookup :: Ord a => Table a b   -> a   -> b
intLookup     ::          Table Int b -> Int -> b

{-# RULES "genericLookup/Int" genericLookup = intLookup #-}
```

The main effect of this optimization is the guaranteed conversion of dynamic type class function calls into static ones (because the type is known, and therefore so is the corresponding dictionary).

3.1.7 Digression: defunctionalization

Defunctionalization — a technique for eliminating higher-order functions from a program[47] Xia. It was first proposed in Reynolds [1972, 1998].

The idea is to replace every creation of a lambda function with a call to a constructor of some algebraic data type. And every call site of a function is replaced with a call to a special first-order function `apply` that interprets this algebraic type.

Consider the example with the higher-order function `map` and two call sites that create lambda functions:

```
map :: (Int -> Int) -> [Int] -> [Int]
map f = \case [] -> []; x:xs -> f x : map f xs

example1 xs = map (\x -> x + 1) xs
example2 y xs = map (\x -> x * y) xs
```

For each lambda literal we create a constructor that stores the closure. Application is replaced with a call to `apply`. Thus we obtained a first-order version of `map`.

```
data Fun = F1 | F2 Int
apply :: Fun -> Int -> Int
apply df x = case df of F1 -> x + 1; F2 y -> x * y

map :: Fun -> [Int] -> [Int]
map df = \case [] -> []; x:xs -> apply df x : map df xs

example1 xs = map F1 xs
example2 y xs = map (F2 y) xs
```

[46] (youtube) John Hughes - Keynote: How to specify it!

[47] `https://en.wikipedia.org/wiki/Defunctionalization`

3.1.8 Emulating higher-kinded polymorphism

Not all languages have higher-rank polymorphism, but sometimes it is useful. Its most common application is emulating type classes of arrow kinds like `Monad`.

Note that a type constructor of kind `Type -> Type` is a function on types that takes one type and returns another. Let us apply defunctionalization to avoid the need to parameterize one type constructor by another Xia, Yallop and White [2014].

We associate to the type constructor `List` a type "symbol" `ListSym` (for illustration we use Kotlin):

```
class ListSym
```

We create a type corresponding to the application of a symbol to a type, storing the original value with erased type:

```
class Apply<Sym, T>(val value: Any)
```

We establish an isomorphism between the original type obtained by type application of the constructor and the new symbol application:[48]

```
fun <T> List<T>.to(): Apply<ListSym, T> = Apply(this)
fun <T> Apply<ListSym, T>.from(): List<T> = this.value as List<T>
```

Now we can declare a monad interface and provide an implementation for lists using a singleton object:

```
interface Monad<M> {
    fun <T> pure(x: T): Apply<M, T>
    infix fun <T, R> Apply<M, T>.bind(k: (T) -> Apply<M, R>): Apply<M, R>
}

object ListMonad : Monad<ListSym> {
    override fun <T> pure(x: T): Apply<ListSym, T> = listOf(x).to()
    override fun <T, R> Apply<ListSym, T>.
            bind(k: (T) -> Apply<ListSym, R>): Apply<ListSym, R> =
        this.from().flatMap { k(it).from() }.to()
}
```

And finally we can write functions over arbitrary monads:[49]

```
fun <M> Monad<M>.go(x: Apply<M, Int>): Apply<M, Int> =
    x bind { it -> pure(it + 1) } bind { it -> pure(it + 2) }

fun test(xs: List<Int>): List<Int> = ListMonad.go(xs.to()).from()
```

[48] To the left of the dot in the declaration, an additional function argument is specified with call syntax matching a method call on an object. From the body of the function, it can be referenced using `this`.

[49] Arguments to the left of the dot can automatically jump into subsequent calls. In fact, they are implicit parameters.

It is worth noting that the resulting code is rather unwieldy. Most likely, using this technique does not pay for itself and one should choose a different programming style.

3.2 Families

The idea of ad-hoc polymorphism is to obtain different code depending on the type. Families started as an extension of this idea into the domain of data and types. Thus, associated type synonyms provide different types for different indices (type parameters) Chakravarty et al. [2005a]. Associated `data` provide different representations for different indices Chakravarty et al. [2005b]. Eventually these ideas were generalized to open families Schrijvers et al. [2008], and then closed families were introduced Eisenberg et al. [2014].

One can think of families[50][51] as type constructors that define a set of types. A specific type from the set can be selected by passing a type parameter called an **index**. Compare this with ordinary polymorphic type constructors, which behave the same regardless of type parameters.

A large number of interesting usage examples can be found, for example, in Kiselyov et al. [2010].

3.2.1 Data families

Data families allow choosing an algebraic type declaration depending on the type index. For example, for a more efficient implementation of data structures. This resembles template specialization in C++.

```
data family XList elem
data instance XList () = IntList Int
data instance XList Bool = BoolList ByteArray
```

The only way to work with a data family is to place the implementation in a type class so that it accompanies the value from the point when the type index is concrete.

```
class XListOp elem where
  xelem :: elem -> XList elem -> Bool

instance XListOp () where
  xelem () (IntList size) = size > 0

instance XListOp Bool where
  xelem key (ByteArray bs) = ...

xelemAll :: XListOp elem => XList elem -> [elem] -> Bool
xelemAll xs = all (`xelem` xs)
```

[50] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_families.html
[51] https://serokell.io/blog/type-families-haskell

Unfortunately, unlike template specialization, one cannot provide a default definition like `data instance XL`

3.2.2 Synonym families

Synonym families (type families) are effectively functions on types. They can be open, closed, and associated. Open families are open in the sense that instances can be written separately from the declaration, similar to type classes. Closed families, on the contrary, are fully described in one place, cases are ordered top to bottom, which somewhat relaxes restrictions on patterns.

```
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus Zero m = m
  Plus (Suc n) m = Suc (Plus n m)
```

To see what something at the type level evaluates to, one can use the following command in ghci:

```
ghci> :k! Plus (Suc Zero) (Suc (Suc Zero))
Plus (Suc Zero) (Suc (Suc Zero)) :: Nat
= Suc (Suc (Suc Zero))
```

Associated families work similarly, only they are declared within a certain type class, serving as a functional alternative to FunctionalDependencies Jones [2000] (which look more relational). In other words, they allow associating another type with the type for which an instance is written. For example, collections and their element type:

```
class Container c where
  type Elem c
  elements :: c -> [Elem c]

instance Container [a] where
  type Elem [a] = a
  elements = id

instance Container ByteString where
  type Elem ByteString = Word8
  elements = ByteString.unpack
```

In modern languages, associated families often appear as associated types[52]. Swift relies heavily on associated types, not supporting generics in protocols (interfaces) at all[53]. Meanwhile, Scala attempts to track, in the presence of existential types (interfaces), which specific value a given associated type came from, using path-dependent types Amin et al. [2014].

[52] Associated types are essentially existential types; they are connected to associated families by the logical process of Skolemization.

[53] (youtube) 2017 LLVM Developers' Meeting: "Implementing Swift Generics"

3.2.3 Injective families

Type families differ from type constructors roughly in the same way that functions differ from data constructors. Constructors are passive and do not reduce (`Maybe Int`), while functions compute to some result (e.g. `F Int ∼ Bool`). In particular, like functions, families are not necessarily injective.

For type constructors, knowing that the types they construct are equivalent, one can infer that the type arguments are equivalent too. For example:

```
Maybe a ∼ Maybe b ⇒ a ∼ b
```

Obviously, for type families this property does not hold by default:

```
type family NonInjective a where
  NonInjective Int = Double
  NonInjective Char = Double
```

Haskell provides explicit syntax for declaring injective type families, resembling functional dependencies in type classes (TypeFamilyDependencies) Stolarek et al. [2015]. Of course, the compiler will verify that the implementation is injective. The syntax requires binding the result to a name via equality and specifying, analogous to FunctionalDependencies, that the result determines some of the type indices of the family:

```
type family InjectiveB a b = r | r -> b
  ...
```

3.2.4 First-class families

Besides injectivity, type families also do not necessarily possess the property of **generativity**, which is critically important for type inference — the same result is not necessarily obtained from the same family:

```
f a ∼ g a ⇒ f ∼ g
```

Together, injectivity and generativity constitute **matchability**. When Haskell works with a type of arrow kind, it assumes that the type is matchable. Accordingly, families cannot be passed as parameters, and all their occurrences must be fully applied to all arguments (fully saturated). Or one needs to explicitly indicate that the family returns a constructor (in this specific case the semantics depend on moving arguments to the right of `::`):

```
type family ToCtor (s :: Symbol) :: Type -> Type where
  ToCtor "maybe" = Maybe
  ToCtor "identity" = Identity
```

One way to work around this restriction is defunctionalization of families Xia, Eisenberg and Stolarek [2014]. As we discussed earlier 3.1.7, instead of a first-class function, a symbol designating it is introduced, along with an interpretation function that can perform the action corresponding to that symbol. In this case, the interpretation function will be an open family `Apply` [Maguire, a, chapter 10].

There is ongoing work in Haskell[54] to remove the saturation restriction Kiss et al. [2019]. For this, one needs to distinguish matchable and unmatchable type functions. This is proposed to be done by additional indexing of arrow kinds: $\rightarrow\equiv\rightarrow^{M}$ and $\twoheadrightarrow\equiv\rightarrow^{U}$. And, of course, these indices can be polymorphic.

```
data Matchability = Matchable | Unmatchable

hMap
  :: forall (m :: Matchability) (c :: Type -> Constraint)
   . forall (f :: Type ->^m Type) (as :: [Type])
   . All as c => (forall a. c a => a -> f a) -> HList as -> HList (Map f as)
```

3.3 The `Constraint` kind

There had long been proposals to add support for constraint synonyms, constraint families, etc. to GHC Orchard and Schrijvers [2010]. Eventually, a mechanism for unifying types and constraints was proposed[55] and implemented[56]. Thus, everything that worked for types now also works for constraints.

In GHC with ConstraintKinds, a special kind `Constraint` was added:

- A type class constructs a constraint: `Monad :: (Type -> Type) -> Constraint`;
- Equality is a constraint: `(a ~ b) :: Constraint`;
- An empty constraint tuple is a constraint: `() :: Constraint`;
- A constraint tuple is a constraint: `(Eq a, a ~ b) :: Constraint`.

Now, for example, we can reify a dictionary as a language-level object:[57]

```
data Dict (c :: Constraint) where
  Dict :: c => Dict c
```

Recall the heterogeneous list considered earlier 2.1.5:

```
data HList (tys :: [Type]) where
  HNil :: HList '[]
  HCons :: ty -> HList tys -> HList (ty : tys)
```

[54] GHC proposal: Unsaturated Type Families.
[55] https://gitlab.haskell.org/ghc/ghc/-/wikis/kind-fact
[56] http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/
[57] https://hackage.haskell.org/package/constraints-0.14.2/docs/src/Data.Constraint.html#Dict

This data structure is the first-class analogue of variadic generics in C++ or Swift[58] (the whole point of variadics is to avoid working with `HList`-like structures directly). For example, we can write `map` for such a structure if all types satisfy a certain constraint. For this, we first implement a family that generates a constraint tuple for each type in the list:

```
type family All (c :: k -> Constraint) (tys :: [k]) :: Constraint where
  All c '[] = ()
  All c (ty : tys) = (c ty, All c tys)

-- All Show [Int, Double] ~ (Show Int, (Show Double, ()))
```

Now we can implement `map`:

```
hmap :: forall c res tys . All c tys
     => (forall ty . c ty => ty -> res) -> HList tys -> [res]
hmap f = \case
  HNil -> []
  HCons x xs -> f x : hmap @c f xs

ghci> hmap @Show show (HCons (1 :: Int) $ HCons 'a' HNil)
```

More exercises of this kind with heterogeneous constructions can be found in de Vries and Löh [2014].

Constraints can also be parametrically polymorphic[59] Bottu et al. [2017]:

```
data Rose f x = Rose x (f (Rose f x))

instance (Eq a, forall b. Eq b => Eq (f b)) => Eq (Rose f a) where
  Rose x1 rs1 == Rose x2 rs2 = x1 == x2 && rs1 == rs2
```

3.4 Uses of ad-hoc polymorphism

Often languages that have something resembling type classes try to express as many other language features and useful techniques through them as possible. It turns out that this is a surprisingly powerful mechanism. In this section we will consider some selected examples.

3.4.1 Serialization

A classic example of using type classes is serialization. The problem is that deserialization is performed when the object does not yet exist (we do not consider the deficient Java approach of filling an object with default values followed by mutation). Therefore, it is not possible to write an OOP interface Serializable.

The standard serialization library in Kotlin[60] provides the entity `KSerializer`, which is an

[58] https://github.com/swiftlang/swift-evolution/blob/main/proposals/0398-variadic-types.md
[59] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/quantified_constraints.html
[60] https://github.com/Kotlin/kotlinx.serialization

interface for a separate serializer object for our type (for efficiency, CPS in the form of encoder and decoder event streams is used here; we will return to this approach later 4.4.1):

```
interface KSerializer<T> {
    fun serialize(encoder: Encoder, value: T)
    fun deserialize(decoder: Decoder): T
}
```

Obviously, to construct a serializer for a polymorphic type, serializers for the type parameters are needed. Which already clearly resembles type classes.

```
class PairSerializer(
    keySerializer: KSerializer<K>,
    valueSerializer: KSerializer<V>,
): KSerializer<Pair<K, V>> { ... }
```

However, Kotlin does not have type classes, so serializers must be created manually. A set of technologies helps with this: `inline fun`, `reified` generics, reflection, and a compiler plugin from the library, which however is not type-safe.

3.4.2 Existential types

The existential quantifier in types is an encapsulation mechanism [Pierce, 2002, chapter 24]. We can replace occurrences of a subtype σ in a type τ with a variable α and obtain $\exists\alpha.\,\tau'$, where $[\alpha \mapsto \sigma]\,\tau' \equiv \tau$. Thus, we have hidden part of the type from the user.

For example, one can use existential types to bring different types to a common form and put their values in a single list. In Haskell, an existential type is expressed through a `data` declaration in which the data constructor is polymorphic but the type is monomorphic[61]. Thus, the trivial existential type $\exists\alpha.\,\alpha$ can be obtained in Haskell as the type `Any`, defined as follows:

```
data Any where
  Any :: forall a . a -> Any  -- logically equivalent to (exists a . a) -> Any

list :: [Any]
list = [Any 42, Any "Hello", Any (Just Nothing)]
```

At the deconstruction site of `Any`, a value of some unknown type will be available. Obviously, nothing can be done with such a value. However, in addition to the value, one can place in the constructor a witness that this unknown type satisfies some type class[62].

```
data Has (c :: Type -> Constraint) where
  Has :: c a => a -> Has c
```

[61] Similar to the type erasure idiom in C++.

[62] Similar to Rust fat pointers (a pair of a pointer to data and a pointer to a virtual function table) and OOP objects.

A value of type `Has` witnesses that there exists some inhabited type `a` that belongs to a certain type class. For example, consider `Show`:

```
showAll :: [Has Show] -> String
showAll = List.intercalate "," . map \(Has x) -> show x
```

In general, to eliminate such a data type, higher-rank types are needed [Pierce, 2002, chapter 24.3]:

```
foldHas :: Has c -> (forall a . c a => a -> b) -> b
foldHas (Has x) k = k x
```

More details can be found in [Maguire, a, chapter 7] and [Pierce, 2002, chapter 24].

3.4.3 Name resolution

The process of name resolution in programming languages determines which program entity a given use of a name is bound to. Name resolution considers imports, the hierarchy of namespaces and scopes, expression types. . . As a rule, this is a complex process, inseparable from type inference.

However, in GHC the name resolution stage is rather simple and runs before type inference. Its trivial nature is hinted at by its name — Renamer — it simply rewrites names in the program to fully-qualified names, relying on imports.

On one hand, simplicity is good. On the other, strict separation from type inference imposes an unpleasant restriction: types cannot participate in name resolution. This problem is most acute with record field labels. One has to name all fields in a module differently to avoid clashes.

To make name resolution depend on types, Haskell once again resorts to type classes. Namely, a type class `IsLabel` is defined that depends on a symbol and the expected type:

```
class IsLabel (s :: Symbol) a where
  fromLabel :: a
```

There is syntactic sugar for calling `fromLabel` (OverloadedLabels):

```
#name ≡ fromLabel @"name"
```

Now resolution of the name `name` will take the type into account[63]:

```
data Pet = Pet { name :: String }
instance IsLabel "name" (Pet -> String) where
  fromLabel Pet{ name } = name

data Person = Person { name :: String, pets :: [Pet] }
instance IsLabel "name" (Person -> String) where
  fromLabel Person{ name } = name

ghci> #name pet
```

[63] To get rid of the redefinition error, you need to enable NoFieldSelectors.

3.4.4 Non-syntactic type equalities, System FC

Modern Haskell is a syntactically rich language that, despite its diversity of constructs, is translated into a small typed internal language. This is the language *System* F_C Sulzmann et al. [2007b], which extends *System F* (2.1) with non-syntactic type equalities. It turns out that this is sufficient to support such Haskell features as generalized algebraic data types, associated type families, functional dependencies, and so on.

Specifically, a built-in constraint $\sim$ is introduced, witnessing the equivalence of two types[64]. For example, the type of function `id` can be written in this unusual way:

```
f :: forall a b . a ~ b => a -> b
f = id
```

In fact, this is a function of four parameters: two type parameters, a coercion, and an argument. A coercion is a zero-sized value, automatically inferred by the compiler, that serves as a witness that the two corresponding types are equivalent.

For example, the GADT from 2.1.4 is desugared as follows:

```
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty
-- translates to
data Expr ty where
  Const :: forall ty . ty ~ Int => Expr ty
  IsZero :: forall ty . ty ~ Bool => Expr Int -> Expr ty
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty
```

And after pattern matching on the constructors, the equality constraint enters the branch and allows the type inference system to perform the necessary rewrites.

More on type inference in the presence of local assumptions can be found in the classical paper *OutsideIn*(X) Vytiniotis et al. [2011].

Obviously, Haskell can inhabit an equality constraint following reflexivity, symmetry, and transitivity. Also, the compiler can generate new axioms (the user cannot directly write their own axioms). For example, for a type family the compiler generates equality axioms between the applied family constructor and the resulting types:

```
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus Zero m = m
  Plus (Suc n) m = Suc (Plus n m)
-- expands to
axiom Plus Zero m ~ m
axiom Plus (Suc n) m ~ Suc (Plus n m)
```

[64] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/equality_constraints.html

3.4.5 Coercions and roles

Haskell supports **generative type abstractions** in the form of `newtype` declarations. This technique allows defining domain-specific types that are indistinguishable from the wrapped types at runtime but allow distinguishing them at compile time. For instance, we can introduce wrappers for numbers that in the problem domain represent identifiers of different entities. Now the type system will prevent confusing them.

```
newtype ModuleId = ModuleId Int64
newtype CourseId = CourseId Int64
```

There is an extremely underappreciated programming practice[65] where our program has a clear boundary at which data from the external world is parsed. Beyond it, raw unstructured data is enriched with structure and takes on meaning within the problem domain. Or we reject this data as incorrect. In the remaining part of the program, we work with typed data whose properties have already been established and guaranteed. For example, we can be sure that the number `ModuleId` is strictly greater than zero.

However, if we have a collection of wrapped data and we want to work with it as a collection of raw data, we would have to transform the collection, even though this transformation does nothing. The Haskell optimizer cannot eliminate it because it works with a typed intermediate representation and cannot remove a conversion that changes the type.

```
newtype Csv = Csv { unCsv :: String }

concatC :: [Csv] -> Csv
concatC = Csv . concat . map unCsv
```

Therefore, Haskell has a mechanism of safe coercions between types with the same runtime representation. This is implemented using the magical type class `Coercible`. The compiler implements it automatically (see Fig. 12).

```
class Coercible from to where
  coerce :: from -> to
```

Now we can eliminate the unnecessary list transformation:

```
concatC :: [Csv] -> Csv
concatC = coerce concat
```

The safety of coercions is ensured by the **role system**. Each type parameter has a special property — a role.

The phantom role is assigned to phantom type parameters. They can be freely coerced (there are no prerequisites for `Coercible` instances):

[65] https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

The most important rules that GHC uses to solve Coercible constraints are as follows (the full rules are given in Figure 5):

(1) The *unwrapping rule*:

- For every **newtype** NT = MkNT t, we have Coercible t NT if and only if the constructor MkNT is in scope.

(2) The *lifting rule*:

- For every type constructor TC r p n, where
  - r stands for TC's parameters at a representational role,
  - p for those at a phantom role and
  - n for those at a nominal role,

  if Coercible r1 r2, then Coercible (TC r1 p1 n) (TC r2 p2 n).

(3) Coercible is an equivalence relation:

- The *reflexivity rule*: Coercible a a.
- The *symmetry rule*: If Coercible a b then Coercible b a.
- The *transitivity rule*: If Coercible a b and Coercible b c then Coercible a c.

Figure 12: Principles of constructing `Coercible` instances Breitner et al. [2014].

```haskell
data Phantom h = Phantom
data NestedPhantom b = MkNP [Phantom b] | SomethingElse

instance Coercible (Phantom a) (Phantom b)
instance Coercible (NestedPhantom a) (NestedPhantom b)
```

A type parameter has the representational role if the type constructor can be coerced only when the arguments can be coerced:

```haskell
data Maybe a = Nothing | Just a
instance Coercible a b => Coercible (Maybe a) (Maybe b)
```

A type parameter has the nominal role if the type constructor can be coerced only when the arguments are equal. This is required when the type argument indexes a family or a constraint.

```haskell
type family F a
data Applied a = Applied (F a)
instance (a ~ b) => Coercible (Applied a) (Applied b)

data ShowDict a where
  ShowDict :: Show a => a -> ShowDict a
instance (a ~ b) => Coercible (ShowDict a) (ShowDict b)
```

Sometimes the compiler infers the wrong role for a type parameter. For example, if the invariants of a structure depend on a specific implementation of some type class for the type argument, which is not visible in the type declaration itself. In such cases, roles can be specified explicitly:

```
type role Map nominal representational
data Map k v = ...
```

More details can be found in Breitner et al. [2014] and [Maguire, a, chapter 8].

3.4.6 Type reflection

Reflection is a language mechanism for obtaining information about types at runtime (at the term level). This sounds familiar, and indeed Haskell implements this mechanism through type classes Peyton Jones et al. [2016].

The library provides the magical type class `Typeable`, which is implemented by the compiler for each concrete type via `deriving`. To obtain information about a type, a `Typeable` instance for that type must be in scope. The type structure is represented by the sum type `TypeRep`, which provides the ability for additional type checking using generalized algebraic data types and type tags.

```
class Typeable a where
  typeRep# :: TypeRep a
```

For example, one can obtain the name of a type constructor as follows:

```
typeName :: forall a. Typeable a => String
typeName = tyConName $ typeRepTyCon $ typeRep $ Proxy @a

ghci> typeName @Int
```

Exercise 18 *Declare a type class that allows printing a list of types.*

Using the type representation structure and existential types, one can emulate dynamic typing in Haskell. Namely: any type can be converted to `Dynamic`, and then safely converted back.

```
data Dynamic where
  Dynamic :: Typeable a => a -> Dynamic

fromDynamic :: Typeable a => Dynamic -> Maybe a
```

This can be useful, for example, for defining a heterogeneous key-value store:

```
data Store = Map Key Dynamic
data Ref ty = Ref Key
get :: Typeable ty => Store -> Ref ty -> Maybe ty
```

3.4.7 Data reflection

As we discussed earlier, the coherence property guarantees that each type in Haskell corresponds to exactly one instance of a given type class. And the only way to declare an instance in Haskell is with a top-level declaration, meaning it cannot depend on any local data. However, Haskell has the library Data.Reflection[66], which allows creating local instances for fresh, magically generated[67] types.

It uses the idea of "lifting values into types" that we discussed earlier (see 3.1.4), but in a somewhat more general form. Instead of defining type classes of the form `Known_`, a single type class is introduced, indexed by the type of terms `terms` that are lowered from types:

```
class Reifies ty terms | ty -> terms where
  reflect :: Proxy ty -> terms
```

Also, using the following function, the library allows generating a fresh type and a `Reifies` instance that, given this fresh type, returns the given value of type `a` (passed as the first argument). Since it is passed into a higher-rank function, the fresh type cannot escape the scope 2.1.2:

```
reify :: a -> (forall fresh . Reifies fresh a => Proxy fresh -> res) -> res
```

To use a non-standard instance of a type class for some type `a`, one needs to declare a new type (for example, using `newtype`), containing the given one, and write the needed instance for it (see, for example, Down). We do not want to declare a new declaration for each case, so we create a wrapper similar to Data.Tagged, which allows adding a phantom type tag to the value type. By varying the tag, we can obtain as many types wrapping the given one as we need.

```
newtype Wrapped tag a = Wrapped { unwrap :: a }
```

We declare the wrapper type `Wrapped tag a` as a member of the needed type class. The implementation code is obtained via `reflect` by the tag type in the form of an actual dictionary.

```
data ReifiedOrd a = ReifiedOrd { compare :: a -> a -> Ordering }

instance Reifies tag (ReifiedOrd a) => Ord (Wrapped tag a) where
  compare = coerce $ compare $ reflect $ Proxy @tag
```

Finally, we can call the sort function, locally replacing the ordering with the reverse one:

```
sort :: Ord a => [a] -> [a]

sortReverse :: forall a . Ord a => [a] -> [a]
sortReverse xs =
  let dict = ReifiedOrd { compare = flip compare } in
  reify dict \(Proxy :: Proxy fresh) ->
    coerce $ sort @(Wrapped fresh a) $ coerce xs
```

[66] https://www.tweag.io/blog/2017-12-21-reflection-tutorial/
[67] https://www.schoolofhaskell.com/user/thoughtpolice/using-reflection

3.4.8 Open structures

In dynamic languages, one can create objects on the fly, sequentially adding content to them without introducing a declaration beforehand. In Haskell, this is also possible using pairs for products and `Either` for sums. For example, one can add a new field by creating a new pair: `(oldObj, newField)`.

However, this implementation is suboptimal both in terms of efficiency (more efficient implementations can be found in [Maguire, a, chapter 11]) and in terms of usability. Namely, the order of fields matters and there is no subtyping relation on types in Haskell (for example, one cannot pass a value with fewer fields or variants). But one can notice that constraints lack these drawbacks. Therefore, the type of a data structure can be organized, for example, as follows:

```
(Int, Double) replaced by (Member Int d, Member Double d) => Prod d
```

3.4.9 Exceptions and the open hierarchy

An important aspect of error handling is that many errors are handled uniformly. Thus, errors should form a hierarchy similar to the one obtained in OOP languages through inheritance, so that one can react to a group of errors with a single handler. This gives rise to the task of modeling such a hierarchy in Haskell.

Moreover, statically typed errors are an active area of research; we will discuss this in the context of effect systems (see 8). Classical exceptions, however, are dynamically typed. This variant is especially well-suited for programmer errors, which ideally should not be handled in the program except for resource cleanup.

Exception support is present in the Haskell runtime system as a simple and familiar way to handle exceptional situations: programmer errors, incomplete pattern match execution, asynchronous system signals Marlow et al. [2001]. . . Exceptions are dynamically typed and form a hierarchy Marlow [2006]. While pure code can throw an exception, since $\perp$ by Haskell's semantics inhabits every type, catching an exception is only possible[68] in `IO` Jones [2001], using special language primitives.

To make a type an exception, one needs to declare an `Exception` instance for it:

```
class (Typeable a, Show a) => Exception a where
  toException :: a -> SomeException
  toException = SomeException

  fromException :: SomeException -> Maybe a
  fromException (SomeException e) = cast e
```

Where `SomeException` is an existential wrapper similar to `Dynamic` (see 3.4.6), into which the concrete exception type is wrapped. By catching `SomeException`, one can catch any exception (`cast` will always succeed).

```
data SomeException where
```

[68] (stackoverflow) Why can Haskell exceptions only be caught inside the IO monad?

```
  SomeException :: Exception a => a -> SomeException

instance Exception SomeException
```

The Haskell runtime system provides intrinsics for throwing and catching exceptions; we wrap them to support any `Exception` type:

```
throw :: Exception e => e -> a
throw = primThrow (toException e)

catch :: Exception e => IO a -> (e -> IO a) -> IO a
catch io handler = io `primCatch` \e -> case fromException e of
  Nothing -> throw e
  Just e' -> handler e'

ghci> throw "error" `catch` \(e :: String) -> putStrLn e
```

In the simplest case, a custom exception type can be implemented in two lines. To catch it, one must either catch this type itself or `SomeException`, because for both of them `fromException` on an object of the form `SomeException MyError` will return `Just`[69].

```
data MyError = MyError deriving (Show, Typeable)
instance Exception MyError
```

Let us add the exception `ArithException` and one more, more general exception type between it and `SomeException` — `SomeArithException` (thus, there will be three ways to catch `ArithException`). For this, we make `SomeArithException` an existential wrapper, and automatically wrap each exception of type `ArithException` in it. In `fromException`, at each level of nesting, we try to obtain the wrapping constructor by a recursive call.

```
data SomeArithException where
  SomeArithException :: Exception a => a -> SomeArithException

-- SomeException - base (default implementation)
instance Exception ArithException

data DivisionByZero = DivisionByZero deriving Show
instance Exception DivisionByZero where
  toException = toException . SomeArithException
  fromException e = do
    SomeArithException e' <- fromException e
    cast e'
```

[69] `instance Exception SomeException where fromException = Just`

Thus, when throwing `DivisionByZero`, an object of the following form will be constructed:

```
SomeException (SomeArithException DivisionByZero)
```

The implementation of `fromException` for a concrete type can verify the presence of the corresponding constructor in the resulting exception object.

3.4.10 Lightweight partial stack traces

An amusing exploitation of type classes in Haskell is lightweight partial stack traces[70]. In general, collecting traces requires runtime support, which in Haskell's case is further complicated by the fact that the computation model, graph reduction, does not contain real traces and they have to be emulated. Here we obtain traces without runtime support.

The standard library defines the constraint `GHC.Stack.HasCallStack`, allowing one to obtain information about the call site of a function. This information is effectively placed by the compiler during instance inference. If information from the level above is available at the call site, the compiler propagates it further. Thus, information is available only to a certain depth of the call stack.

```
myHead :: HasCallStack => [a] -> a
myHead []     = error "empty"
myHead (x:xs) = x

bad :: Int
bad = myHead []

ghci> bad
*** Exception: empty
CallStack (from HasCallStack):
  error, called at Bad.hs:8:15 in main:Bad
  myHead, called at Bad.hs:12:7 in main:Bad
  -- no information about bad call site here
```

`HasCallStack` is simply an implicit (see 3.1.2) that the compiler knows about:

```
type HasCallStack = (?callStack :: CallStack)
```

3.4.11 Customizable type errors

When developing libraries that are complex from the type perspective, it is desirable to provide users with more informative type errors than the defaults. For this, GHC has a mechanism in GHC.TypeLits that allows constructing a special type whose information will appear in the error message. For example, this type can be returned from a synonym family with an incorrect set of arguments. Or one can use the constraint trick (see 3.1.3) and place such a type as a premise in an

[70] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/callstack.html

instance. If the instance matched and the compiler started processing the constraints on the left, something went wrong [Maguire, a, chapter 12].

```haskell
instance (TypeError
  ( Text "Attempting to show a function of type "
    :<>: Text "'" :<>: ShowType (a -> b) :<>: Text "'"
    :$$: Text "Did you forget to apply an argument?"
  )) => Show (a -> b) where
  show = undefined -- the implementation does not matter, execution will not reach her
```

4 Data Types

This chapter collects some general knowledge about types. We will also obtain various equivalent representations of recursive data types (in other words, collections). Many concepts are special cases of this variety.

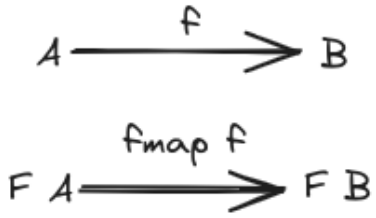
Sections 4.1, 4.2 mostly follow [Maguire, a, chapter 1].

4.1 Variance

In this section we will consider the topic from the programming perspective [Maguire, a, chapter 3], without doing justice to category theory. The gap can be filled with the help of a wonderful article written in the style of a play Hinze et al. [2012].

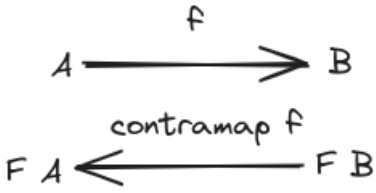
A **covariant functor** is a pair of some type constructor `F` and an operation on functions `fmap :: (a -> b) -> (F a -> F b)`. Plus laws stating that `fmap` respects `id` and composition.

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```



A **contravariant functor** is a pair of a type constructor and an operation on functions that reverses the arrow. Plus corresponding laws.

```
class Contravariant f where
  contramap :: (a -> b) -> (f b -> f a)
```



A type constructor can be declared a covariant or contravariant functor (or neither) with respect to a given type parameter depending on the form of the declarations of the corresponding data

constructors. Namely, on the sign of the positions in which that type parameter appears in the type.

Let us develop an intuitive understanding of position signs. Type `A` appears in a positive position in `B` if its value can be extracted from `B`. Conversely, type `A` appears in a negative position if its value must, on the contrary, be provided. Let us consider position signs of types in basic type constructors:

Type	sign of position `A`	sign of position `B`
`Either A B`	+	+
`(A, B)`	+	+
`A -> B`	−	+

Indeed, from a sum and a product one can extract components via pattern matching, and from an arrow one can obtain the right type by applying it to an argument. At the same time, a value of the type on the left of the arrow must be provided.

Plus and minus obey intuitive algebraic rules when considering more complex types. Let us consider the example `f :: ((A, B) -> C) -> (D, E)`.

- Plus times plus gives plus. Indeed, one simply needs to apply two eliminations instead of one to obtain the desired type. In our example, to obtain `D`, one first applies the function, then destructs the pair.
- Plus times minus (and vice versa) gives minus. Indeed, `C` must be provided by us: `f (\ab -> provideC)`.
- Minus times minus gives plus. The pair `(A, B)` is provided to us: `f (\ab -> ...)`.

Exercise 19 *Verify that plus times minus gives minus.*

Returning to functors, if a type parameter appears in the declaration only in positive positions, the type constructor can be declared a covariant functor with respect to that parameter. If only in negative positions — a contravariant functor. If in both, then it cannot be declared any functor. Accordingly, we call type parameters covariant, contravariant, and invariant.

Exercise 20 *Declare* `instance Contravariant F` *for* `data F a = L (a -> ()) | R Int`.

Thus, a covariant functor can be understood as a computation whose result can be post-processed, and a contravariant functor as a computation whose arguments can be pre-processed.

A type with two positive parameters can be declared a **bifunctor**:

```
class Bifunctor f where
  bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

A type with two parameters, one positive and one negative, is a **profunctor**:

```
class Profunctor p where
  dimap :: (c -> a) -> (b -> d) -> p a b -> p c d
```

Profunctors are certain generalizations of the function arrow. For example, if we have an SQL query that takes data and returns a result, it can be declared a profunctor with the semantics of adding pre-processing of input data and post-processing of output data:

```
dimap serialize deserialize (query :: Sql Text Text) :: Sql Age [User]
```

The concept of variance is also frequently encountered in object-oriented languages (and in subtyping theory in general) to denote the possibility of extending the subtyping relation to polymorphic types.

Indeed, the **subtyping relation** `B <: A` says that a value of type `B` can safely be used in a position where a value of type `A` is expected. In other words, there exists a function `upcast :: B -> A`. If the type constructor `F a` is covariant with respect to parameter `a`, then from `upcast` one can obtain `upcast' :: F B -> F A`. That is, the subtyping relation also automatically includes `F B <: F A`. The contravariant case is analogous.

Exercise 21 *Verify in your favorite language with subtyping and variance support that minus times minus gives plus.*

4.2 Isomorphism

Suppose we need to design a function or a data model. We start with a type declaration — how should we choose it and from which alternatives? First, let us understand when two types are interchangeable; for this, we consider the notion of isomorphism.

Two types `A` and `B` are called **isomorphic** (denoted `A` $\cong$ `B`) if and only if there exists a pair of functions `to :: A -> B` and `from :: B -> A` such that[71]

```
to . from = id
from . to = id
```

In other words, one can establish a bijection between the inhabitants of such types. It is easy to see that from a semantic point of view it does not matter which of the isomorphic types is used — they can be substituted for each other by adding calls to the conversion functions. Two such types contain the same "amount of information." For example, the types `Bool` and `Maybe ()` are in this sense completely interchangeable. Let us demonstrate this by exhibiting a pair of mutually inverse functions[72]:

```
to :: Bool -> Maybe ()
to b = if b then Just () else Nothing

from :: Maybe () -> Bool
from m = case m of Nothing -> False; Just () -> True
```

[71] Equality of terms can mean different things, for example $\alpha\beta\gamma$-equivalence. We will use **extensional equality** for functions — two functions are equal when their results are equal on all inputs. `https://ncatlab.org/nlab/show/function+extensionality`

[72] One must not forget to show that the functions are mutually inverse, but this is done trivially by enumeration of inputs (possibly with induction) and reduction.

Despite semantic interchangeability, to encode information about whether the user passed a particular flag to the program, we would most likely use the type `Bool` due to non-functional considerations of code readability. Analogously, one can consider efficiency considerations.

A categorical perspective on the matter can be found in Hinze and James [2010]. We, however, adhere to the set-theoretic interpretation of types.

4.2.1 Cardinality: sums, products, exponentials

Types can be thought of as syntax for writing sets, and the terms inhabiting them as syntactic representations of elements of those sets. Thus the term `(True, False)` is a representation of an element of the set of pairs, written in type syntax as `(Bool, Bool)` (instead of the mathematical $\mathbb{B} \times \mathbb{B}$). Or the term `\x -> x + 1` is a representation of the function adding one from the set of functions over integers, written as `Integer -> Integer` (instead of the mathematical $\mathbb{Z} \to \mathbb{Z}$).

Note that two types are isomorphic if the corresponding sets have the same number of elements. Moreover, there are $n!$ such isomorphisms in the case of finite sets. Let us learn to determine the number of such elements. Using $|\cdot|$ we will denote the **cardinality** of a type — the number of elements in the corresponding set.

Type and its declaration	cardinality
`data Void`	0
`data Unit = Unit`[73]	1
`data Bool = False | True`	2

The idea of algebraic data types is that complex types can be built from simple ones using the operation $+$ ("or") and the operation $\times$ ("and")[74]:

Type	cardinality
`data Either a b = Left a | Right b`	$\|a\| + \|b\|$
`data Pair a b = Pair a b`	$\|a\| \times \|b\|$

Let us count the number of inhabitants of various types (you can verify the conclusion by enumerating all terms manually):

- `|Either Unit (Either Bool Bool)| = |Unit| + (|Bool| + |Bool|) = 5`.
- `|Pair (Either Bool Unit) (Pair Unit Void)| = 0` — the type `Void` is uninhabited, as is a tuple containing it.
- If `data Example = FirstAlternative Bool | AnotherOne Unit Bool Bool`, then `|Example| = |Bool| + |Unit| * |Bool| * |Bool| = 2 + 1 * 2 * 2 = 6`.

The function arrow is called the exponential type. Indeed, combinatorially the number of inhabitants of `A -> B` is computed as

$$|A \to B| = |B|^{|A|}$$

So how should one design types? There are several considerations:

[73] `Unit` is written in Haskell using special syntax `()`, meaning something like an empty tuple.

[74] `https://stanford-cs242.github.io/f18/lectures/02-2-algebraic-data-types.html`

- The type should have no fewer elements than the domain, so that all necessary objects are representable.
- The type should have as few elements as possible that are not in the domain, so that the error space is minimal.
- Then among isomorphic types, the optimal one is chosen based on non-functional requirements.

Per the second rule, before manipulating a domain object one should first convert the data describing it into the most structured representation possible — the one that provides the strongest guarantees[75].

4.2.2 Algebraic representation of a type

As we saw above, to show the existence of an isomorphism between two types one can either exhibit a pair of mutually inverse functions, or show that the cardinalities of the two types coincide. In this section we will learn to associate an algebraic expression with a type, reflecting its structure and cardinality. Thus, we will be able to obtain equivalent expressions via syntactic transformations of formulas, from which we will recover types that are guaranteed to be isomorphic to the given one[76].

The algebraic representation is based on computing cardinalities of types. In effect, we forget information inessential for isomorphism about the names of data constructors and type constructors, i.e. we transition to structural typing.

Type	algebraic formula
`data Void`	0
`data Unit = Unit`	1
`data Bool = False | True`	$1+1$ (denote as 2)
`data Maybe a = Nothing | Just a`	$1+a$
`data Either a b = Left a | Right b`	$a+b$
`data Pair a b = Pair a b`	$a \times b$
`a -> b`	b^a

Exercise 22 *Write the following type in algebraic form:*

```
data T a b = Undefined | Defined a (a -> b)
```

As the equivalence relation, we use isomorphism of the corresponding types. Under this interpretation, the classical properties of algebraic operations are preserved (Fig. 13). Indeed, for example:

```
-- (c^b)^a ≅ c^(a×b)
to :: (a -> b -> c) -> (a, b) -> c
to = uncurry
from :: ((a, b) -> c) -> a -> b -> c
from = curry
```

sums

$$X + 0 \cong X \quad X + Y \cong Y + X$$
$$(X + Y) + Z \cong X + (Y + Z)$$

products

$$X \times 1 \cong X \quad 0 \times X \cong 0 \quad X \times Y \cong Y \times X$$
$$(X \times Y) \times Z \cong X \times (Y \times Z)$$

product over sum

$$(X + Y) \times Z \cong (X \times Z) + (Y \times Z)$$

exponentials

$$1^X \cong 1 \quad X^0 \cong 1 \quad X^1 \cong X$$
$$Z^{X+Y} \cong Z^X \times Z^Y \quad (Z^Y)^X \cong Z^{X \times Y}$$

exponential over product

$$(Y \times Z)^X \cong Y^X \times Z^X$$

Figure 13: Laws of school algebra for nostalgia's sake Hinze and James [2010].

Exercise 23 *Show that* $(a + b) + c \cong a + (b + c)$.

Exercise 24 *Show that* $c^{a+b} \cong c^a \times c^b$.

An interesting observation is that functions can be used as data structures, in accordance with the isomorphism $c^{a+b} \cong c^a \times c^b$. Indeed, in that case the function argument serves as an index (its cardinality must coincide with the size of the collection).

```
-- a × a ≅ a²
get :: (a, a) -> (Bool -> a)
get (x, y) idx = if idx then x else y
tabulate :: (Bool -> a) -> (a, a)
tabulate f = (f True, f False)
```

The **canonical representation of a type** is a sum of products of types:

$$\sum_i \prod_j t_{ij}$$

The canonical representation is a kind of normal form in which algebraic types can be written (any algebraic type can be reduced to it using the rules). It is easy to recognize in it the form of `data` declarations in Haskell.

Remarkably, even the derivative of an algebraic type has a perfectly understandable semantics. It is the context of a zipper, a data structure similar to an iterator that allows navigating through a data structure of arbitrary shape Huet [1997], McBride [2001], Abbott et al. [2003].

[75] https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

[76] https://codewords.recurse.com/issues/three/algebra-and-calculus-of-algebraic-data-types

4.3 Recursive types

Recursive types, similarly to recursive terms, can include themselves in their definitions. In other words, a recursive type is isomorphic to some other type in which it itself appears as a subtype.

```
fac n = if n <= 1 then 1 else n * fac (n - 1)
data Nat = Zero | Suc Nat
```

We will mostly look at recursive types from a practical perspective. However, their formal type-theoretic description, category-theoretic and set-theoretic interpretations are of independent interest [Pierce, 2002, part 4].

4.3.1 Just a list

Consider the classical functional list. A list is either a collection of zero elements, or one, or two... Algebraically this can be written as follows:

$$L = 1 + a + a^2 + a^3 + \dots$$

In effect we obtained a type with an infinite expression. Let us work with it as a formal series. Factor out a:

$$L = 1 + a \times (1 + a + a^2 + \dots)$$

Notice that the expression in parentheses represents a list, yielding the following recursive equation[77]:

$$L = 1 + a \times L$$

It is easy to see that this is in fact the familiar list definition from Haskell:

```
data List a = Nil | Cons a (List a)
```

Let us obtain a finite non-recursive representation of the type L. In effect, we need to obtain a type isomorphic to a type that includes the original:

$$L \cong 1 + a \times L$$

Let us extend the type language with abstraction (polymorphism) and solve the resulting recursive equation in the style of λ-calculus, using a fixpoint combinator:

$$L = FIX\ \lambda r.\, 1 + a \times r$$

Let us encode this in Haskell. As the recursion combinator we take `data FixList a`:

[77] Alternatively, one can obtain the same result by noticing that we are dealing with a Taylor series `https://codewords.recurse.com/issues/three/algebra-and-calculus-of-algebraic-data-types`.

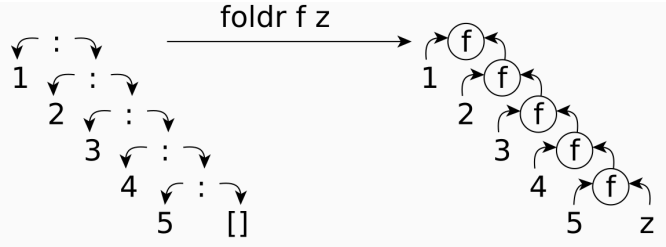
```haskell
type Shape a r = Either () (a, r) -- λar. 1 + a × r
data FixList a = In (Shape a (FixList a))
-- FixList a ≅ Shape a (FixList a) ≅ Shape a (Shape a (FixList a)) ≅ ...

example :: FixList Int -- [1, 2, 3]
example = In (Right (1, In (Right (2, In (Right (3, In (Left ()))))))))
```

Thus, we have separated the list definition into two parts: one is responsible for the shape of the type, and the other for recursion[78]. The shape can be reused in the definition of a fold:

```haskell
foldr :: (Either () (a, r) -> r) -> FixList a -> r
foldr phi (In shape) = case shape of
  Left () -> phi (Left ())
  Right (x, xs) -> phi (Right (x, foldr phi xs))
-- compare with the classical definition
foldr :: r -> (a -> r -> r) -> [a] -> r
foldr ini f list = case list of
  [] -> ini
  x:xs -> f x (foldr ini f xs)
```

Indeed, a fold in the general sense allows replacing each data constructor in the tree with some function. The result is a computation that has access to the entire contents of the data structure and returns an aggregated result:



That is, the number and types of arguments to the fold depend on the "shape" of the type being folded. In the case of a list, a nullary function is needed in place of `Nil` and a binary one in place of `Cons`. We represent these two functions as one using the shape type. Compare:

```haskell
foldr                   0 (      \e  r  -> e + r)
foldr (\case Left () -> 0; Right (e, r) -> e + r)
```

4.3.2 Fixpoint of a functor

Let us abstract `FixList` over the shape type:

[78] The technique of abstracting over the "recursive self-reference" is called **open recursion**. This gives the user more control. For example, at the term level, the recursion combinator can count the number of recursive calls and abort the computation when some limit is exceeded.

```haskell
newtype Fix :: (Type -> Type) -> Type
newtype Fix f = In { out :: f (Fix f) }

data ListF a r = Nil | Cons a r
type List a = Fix (ListF a)
```

Exercise 25 *What will the types of* `In` *and* `out` *be?*

One can show that `[a]` ≅ `List a`:

```haskell
to :: [a] -> List a
to = \case
  [] -> In Nil
  x:xs -> In $ Cons x (to xs)

from :: List a -> [a]
from (In shape) = case shape of
  Nil -> []
  Cons x xs -> x : from xs
```

The shape type can be made a functor in its last parameter. This will allow us to subsequently replace occurrences of subtrees with something useful.

```haskell
instance Functor (ListF a) where
  fmap :: (rec -> other) -> ListF a rec -> ListF a other
  fmap f = \case
    Nil -> Nil
    Cons x xs -> Cons x (f xs)
```

Exercise 26 *Express the following tree as a fixpoint of a functor. Declare a functor instance for the shape type.*

```haskell
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

4.3.3 Recursion schemes

Just as structured imperative programming, compared to undisciplined use of `goto`, helps in reasoning about programs, so **recursion schemes** allow one to algebraically describe properties of recursive functions, as opposed to "unstructured" recursion[79] Meijer et al. [1991], Meijer and Hutton [1995].

The overarching idea was to formulate required properties and compute the needed programs similarly to how mathematicians find solutions to differential equations. However, this idea did not gain the necessary development and adoption (though mastering the algebraic approach is generally

[79] `https://reasonablypolymorphic.com/blog/recursion-schemes/index.html`

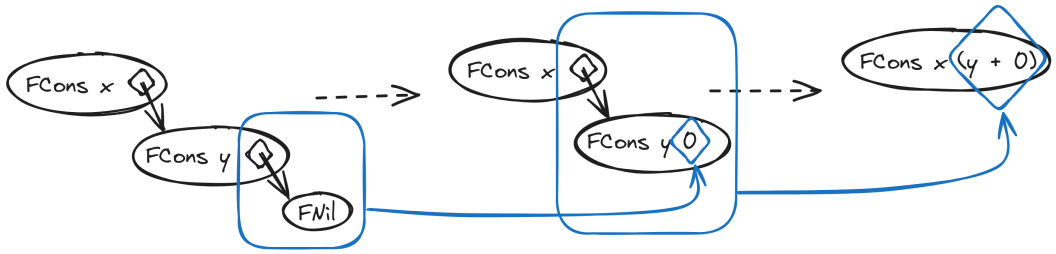


Figure 14: A catamorphism reuses occurrences of recursive subtrees to store the fold results of the corresponding subtrees.

useful Maguire [b]). Nevertheless, this knowledge, on the one hand, gives a deeper understanding of recursion, and on the other hand, will serve us when examining and solving the main problem of this course — the expression problem.

The universal fold is called a **catamorphism**. A catamorphism first recursively folds the subtrees, reaching them via `fmap`, and leaves fold results of type `a` in place of the former occurrences of subtrees. The result is a value of type `f a`, where `f` is some shape functor. Then a function of type `f a -> a` is applied, which determines how to fold one layer of the recursive structure when the subtrees have already been folded.

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata phi = phi . fmap (cata phi) . out
```

In other words, `cata` replaces all `In` constructors with the function `phi`, which contains information about how to react to different constructor variants in the tree:

```
example           = In  (Cons 1 (In  (Cons 2 (In  (Cons 3  (In  Nil))))))
cata phi example =β phi (Cons 1 (phi (Cons 2 (phi (Cons 3 (phi Nil))))))
```

For example, summing a list looks as follows (Fig. 14):

```
sum :: List Int -> Int
sum = cata \case
  Nil -> 0
  Cons x result -> x + result
```

The function `f a -> a` is called an **f-algebra**. Indeed, if we take the signature of an algebra as the functor `f` and the carrier as `a`, then the f-algebra defines some interpretation of the signature:

```
data MonoidSig carrier = Mempty | Mappend carrier carrier

interpretSig :: MonoidSig Int -> Int
interpretSig = \case Mempty -> 0; Mappend l r -> l + r
```

Exercise 27 *Which algebra can be used to copy a data structure?*

Exercise 28 *Which algebra can be used to print a list as a string?*

In contrast to the universal fold, one can construct an **anamorphism** — a universal unfold (analogous to `unfold` for a list). Here the f-coalgebra (the arrow in the opposite direction) shows how to obtain one layer of a data structure from some seed value, where in place of recursive references there will be seeds from which subtrees will subsequently grow. The anamorphism first unfolds one layer, and then recursively unfolds all the subtrees:

```
ana :: Functor f => (s -> f s) -> s -> Fix f
ana psi = In . fmap (ana psi) . psi
-- compare with the classical definition of list unfold
unfoldr :: (s -> Maybe (a, s)) -> s -> [a]
```

Exercise 29 *Implement an anamorphism that builds a list from* 0 *to a given n.*

One also introduces the **hylomorphism**, which allows describing an arbitrary recursive computation. A hylomorphism is defined as a composition of an anamorphism and a catamorphism. First, the anamorphism builds an explicit tree representing the call tree of some recursive procedure, then the catamorphism folds it into the result.

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo psi phi = cata phi . ana psi
```

For example, computing factorial can be implemented as follows:

```
fac n = hylo
  (\n -> if n > 0 then Cons n (n - 1) else Nil)
  (\case Nil -> 1; Cons n acc -> n * acc)
```

One can introduce many more different recursion schemes[80] and describe their properties. However, we will stop here for now.

An interesting fact: a catamorphism represents the set-theoretic principle of induction, and an anamorphism represents coinduction, if one takes the function as the set embedding [Pierce, 2002, chapter 21].

4.4 Everything through folds

It turns out that using a catamorphism one can obtain an isomorphism between data structures and their folds: `Fix f` $\cong$ `forall a . (f a -> a) -> a`.

```
to :: Functor f => Fix f -> (forall a . (f a -> a) -> a)
to = flip cata

from :: (forall a . (f a -> a) -> a) -> Fix f
from g = g In
```

[80] https://wiki.haskell.org/Zygohistomorphic_prepromorphisms

For example, the following two lists are equivalent (all `In` constructors are replaced with the given algebra):

```
data ListF elem rec = Nil | Cons elem rec

xs1 :: Fix (ListF Int)
xs1 = In (Cons 1 (In (Cons 2 (In (Cons 3 (In Nil))))))

xs2 :: (ListF Int a -> a) -> a
xs2 = \alg -> alg (Cons 1 (alg (Cons 2 (alg (Cons 3 (alg Nil))))))

ghci> xs2 @Int \case Nil -> 0; Cons x acc -> x + acc
6
```

Now let us get rid of the shape functor. It is a non-recursive type that can be presented in canonical form (see 4.2.2):

$$f\ a \cong \sum_i \prod_j (t_{ij}\ a)$$

Then the algebra can be written as follows:

$$f\ a \to a \cong a^{\sum_i \prod_j (t_{ij}\ a)} \cong \prod_i a^{\prod_j (t_{ij}\ a)}$$

Products remain, which can be eliminated using currying, yielding **Church encoded** data structures. For example, for a list we have:

$$\texttt{(ListF elem a -> a) -> a} \cong a^{a^{1 + elem \times a}} \cong a^a \times a^{elem \times a} \cong (a^a)^{\left((a^a)^{elem}\right)}$$
$$\cong \texttt{a -> (elem -> a -> a) -> a}$$

We have obtained nothing other than a Church list[81][82]. A data structure without a single constructor![83] Let us rewrite the list we already know once more:

```
xs3 :: a -> (Int -> a -> a) -> a
xs3 = \ini f -> f 1 (f 2 (f 3 ini))
```

Exercise 30 *Which familiar standard list function on a* `data` *list returns a Church list?*

Let us try to intuitively understand what all this means. Notice that a Church list accepts functions corresponding to pattern-matching branches or arguments of the folding function. Instead

[81] https://en.wikipedia.org/wiki/Church_encoding

[82] https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html

[83] In fact this is not at all surprising, if one recalls that first-class functions are represented as closures that contain data. We have obtained the same singly linked list, but using closures.

of constructors, the corresponding functions are called directly. That is, instead of creating a data structure and delivering it to the site of deconstruction (pattern matching), we *deliver the deconstruction site to the construction site*, and it turns out that there is nothing to construct after all.

We have already worked with the reverse process, defunctionalization, when first-class functions were turned into constructor calls, and their bodies into pattern-matching branches 3.1.7. One can notice that here we are dealing with the reverse process, **refunctionalization**, when instead of constructor calls the corresponding interpreting code is called directly Danvy and Millikin [2009].

From a technical perspective, we have understood that conditional branching (if, pattern matching, switch constructs) and virtual calls (closure calls, method calls on interfaces) are interchangeable.

It is also worth noting that the isomorphism `Fix f ≃ forall d . (f d -> d) -> d` is a generalization of the isomorphism `a ∼ forall r . (a -> r) -> r` (with `f = Const a`), which is the foundation of the CPS transformation that we consider later in the chapter 6.

4.4.1 Deforestation & list fusion

In functional programming we build new functions by composing existing, simpler functions. This approach allows reusing implemented functionality, reducing code complexity and the probability of errors. However, it can lead to unnecessary overhead from allocation and deconstruction of intermediate data structures.

For example, compare the following two implementations. The first is preferable from a code quality standpoint, but it creates an intermediate list during execution:

```
all p xs = and (map p xs)
-- or the fused version
all p [] = True
all p (x:xs) = p x && all p xs
```

Obviously, this is the job of an optimizing compiler to turn good code with abstractions into fast code. The optimization that eliminates (intermediate) data structures (trees) from programs is called **deforestation**. As a result, two functions are said to "fuse."

The term and the first deforestation algorithm were proposed by Philip Wadler Wadler [1988]; it is based on several simple rewriting rules aimed at obtaining the situation `case K args of ...`, and aggressive inlining (see examples of operation in Fig. 15). However, this algorithm can lead to exponential code blowup and may not terminate in the presence of recursive calls[84]. For the algorithm to terminate, programs must be written in a certain strict form called treeless.

Let us focus on lists and consider more practical solutions[85]. The first obvious solution would be to add a special rewrite rule for each pair of list functions (see 3.1.6), deforesting using algebraic properties of list transformations. However, there would be exponentially many such rules.

[84] Deforestation is a special case of supercompilation Романенко, which in turn is a generalization of a large number of compiler optimizations.

[85] `https://markkarpov.com/tutorial/ghc-optimization-and-fusion.html`

append (*append xs ys*) *zs*

transforms to

```
h0 xs ys zs
where
 h0 xs ys zs  =  case xs of
                   Nil        :  h1 ys zs
                   Cons x xs  :  Cons x (h0 xs ys zs)
 h1 ys zs     =  case ys of
                   Nil        :  zs
                   Cons y ys  :  Cons y (h1 ys zs)
```

---

flip (*flip zt*)

transforms to

```
h0 zt
where
 h0 zt  =  case zt of
             Leaf z         :  Leaf z
             Branch xt yt   :  Branch (h0 xt) (h0 yt)
```

Figure 15: Examples of the deforestation algorithm in action Wadler [1988].

```
std::variant<Msg1, Msg2>
    deserialize(bytes bs) {
    if (...) {
        return std::variant{Msg1(...)};
    else {
        return std::variant{Msg2(...)};
    }
}
```

```
template<class Impl>
auto deserialize(bytes bs) {
    if (...) {
        return Impl::processMsg1(...);
    else {
        return Impl::processMsg2(...);
    }
}
```

Figure 16: Manual deforestation in C++.

```
{-# RULES
"map/map" forall f g xs. map f (map g xs) = map (f . g) xs
  #-}
```

The modern deforestation technique in Haskell — fold/build list fusion Gill et al. [1993] — instead of using many algebraic rules, defines a universal way of constructing and deconstructing a list. We deconstruct a list using `foldr`. We construct using the function `build`:

```
build :: (forall b . (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

For example, the list `[1, 2, 3]` and the function `map` are now written as follows[86]:

```
list123 :: [Int]
list123 = build \s z -> s 1 (s 2 (s 3 z))

map :: (a -> b) -> [a] -> [b]
map f xs = build \s z -> foldr (\x acc -> s (f x) acc) z xs
```

One can notice that the input to `build` is a Church list, hence the unsurprising law:

```
foldr f ini (build g) ≡ g f ini
```

Thus, we eliminated list construction by replacing constructor calls with calls to folding functions. Deforestation can also be performed manually (see 16).

4.4.2 Visitor pattern

Consider some tree and its fold:

```
data Tree a = Leaf | Node a [Tree a]
foldTree :: Tree a -> r -> (a -> [r] -> r) -> r
```

[86] In the Haskell standard library, list functions are written normally, but alongside them are `RULES` rules (see 3.1.6) that substitute their implementation with fold/build versions.

Let us rewrite and rename:

```
data Visitor a r = Visitor { onLeaf :: r, onNode :: a -> [r] -> r }
visitTree :: Tree a -> Visitor a r -> r
```

To make this look even more like OOP style, the tree itself should be given by a fold (as an interface with a `visit` function), and different nodes by concrete implementations (descendant objects):

```
data Tree a = Tree { visit :: forall r . Visitor a r -> r }

leaf :: Tree a
leaf = Tree { visit = \Visitor{onLeaf} -> onLeaf }

node :: a -> [Tree a] -> Tree a
node x ts = Tree { visit = \v@Visitor{onNode} -> onNode x (map (`visit` v) ts) }
```

For clarity, let us show this code in Kotlin as well:

```
interface Visitor<a, r> {
    fun onLeaf(): r
    fun onNode(x: a, subtrees: List<r>): r
}

interface Tree<a> {
    fun <r> visit(visitor: Visitor<a, r>): r
}
class Leaf : Tree<Nothing> {
    override fun <r> visit(visitor: Visitor<Nothing, r>): r = visitor.onLeaf()
}
class Node<a>(val value: a, val subtrees: List<Tree<a>>) : Tree<a> {
    override fun <r> visit(visitor: Visitor<a, r>): r =
        visitor.onNode(value, subtrees.map { it -> it.visit(visitor) })
}
```

4.5 Everything through unfolds

Recall that there exists a universal unfold — the anamorphism, which from a generating procedure produces an entire data structure (see 4.3.3).

```
ana :: Functor f => forall s . (s -> f s) -> s -> Fix f
ana psi = In . fmap (ana psi) . psi
```

Let us bring the function `ana` to a type of the form `A -> B`, so that it is then easier to study the arrow `B -> A`. To do this, we uncurry and move the quantifier to the left of the arrow (it changes to the opposite):

```haskell
ana :: Functor f => (exists s . (s, s -> f s)) -> Fix f
```

Let us encode the existential quantifier using a new type (see 3.4.2) and rewrite the anamorphism to work with it:

```haskell
data Box f where
  --     exists s. (s,    s -> f s)
  Box :: forall s . s -> (s -> f s) -> Box f

ana' :: Functor f => Box f -> Fix f
ana' (Box currSeed psi) =
  In $ (\nextSeed -> ana' (Box nextSeed psi)) <$> psi currSeed
```

Now let us build an isomorphism between data structures and their trivial unfolds, which each time return the next layer of the given structure, `Fix f` $\simeq$ `Box f`:

```haskell
to :: Fix f -> Box f
to x = Box x out

from :: Functor f => Box f -> Fix f
from = ana'
```

Thus, we have obtained evidence that any recursive data structure can be, at least trivially, represented as `Box f`. Such a representation is sometimes called **co-Church encoding** Gibbons [2008].

For example, an infinite lazy list of natural numbers can be defined as follows. Note that here we *do not rely on Haskell's laziness*, which means we can use this technique in eager languages as well.

```haskell
nats :: Box (ListF Int)
nats = Box 0 \curr -> Cons curr (curr + 1)
```

Exercise 31 *Implement a lazy function*
`take :: Int -> Box (ListF Int) -> Box (ListF Int)`.

In other words, infinite data structures can be represented as procedures that lazily generate one layer of a recursive structure at a time on demand.

4.5.1 Abstract data types

Consider the case when the shape functor is a product:

$$\texttt{s -> f s} \cong \left(\prod_i (t_i\ s)\right)^S \cong \prod_i (s \to t_i\ s)$$

That is, the coalgebra is equivalent to a tuple of functions. Thus, `Box` is nothing other than an *abstract data type (ADT)*: it contains hidden state of unknown nature and a set of operations for working with it Gibbons [2008]. In fact, we have already encountered such a construction earlier, when we discussed existential types 3.4.2 (there we used type class instances as tuples of functions).

OOP objects can also be viewed through coalgebras. More about the relationship between objects and abstract data types can be found in Cook [2009].

4.5.2 Stream fusion

Earlier we considered the foldr/build list fusion optimization that eliminates intermediate lists (see 4.4.1). However, this technique does not work for many popular functions, such as `zip` (one needs to coroutine between two algebras) or `take` (one needs to abort the fold at a certain point).

Later, a technique using the co-structure of a list — an unfold or `Stream` — was proposed[87] Coutts et al. [2007]:

```haskell
data ListF a r = Nil | Cons a r
type MyStream a = Box (ListF a) -- ∃s . (s -> ListF a s, s)
-- ≅
data Step a s = Done | Yield a s
data Stream a where
  Stream :: forall s . (s -> Step a s) -> s -> Stream a

stream :: [a] -> Stream a
stream xs = Stream (\case [] -> Done; x:xs -> Yield x xs) xs

unstream :: Stream a -> [a]
unstream (Stream next s) = case next s of
  Done -> []
  Yield a s' -> a : unstream (Stream next s')
```

The idea is that now *stream processing functions are non-recursive* and easily undergo basic compiler deforesting transformations in the style of Wadler [1988]. Indeed, instead of a recursive call we remember the state for the next iteration.

```haskell
mapS :: (a -> b) -> Stream a -> Stream b
mapS f (Stream next s) = Stream next' s
  where
    next' s = case next s of
      Done -> Done
      Yield x s' -> Yield (f x) s'
```

The overall pipeline looks as follows:

[87] https://markkarpov.com/tutorial/ghc-optimization-and-fusion.html

1. We write all list functions through streams:

```haskell
map :: (a -> b) -> [a] -> [b]
map f = unstream . mapS f . stream
```

2. Using RULES (see 3.1.6) we define a rewrite rule for stream/unstream, dual to foldr/build: `stream . unstream = id`.
3. Standard compiler optimizations take care of the rest.[88]

To support a non-recursive filter function, a separate kind of step is usually added:

```haskell
data Step a s = Done | Skip s | Yield a s

filterS :: (a -> Bool) -> Stream a -> Stream a
filterS p (Stream next s) = Stream next' s
  where
    next' s = case next s of
      Done -> Done
      Skip s' -> Skip s'
      Yield a s' -> if p a then Yield a s' else Skip s'
```

The Haskell standard library still uses foldr/build fusion, but there exist many industrial stream libraries [Bragilevsky, chapter 14].

More about streams can be found at Oleg Kiselyov's page[89].

4.6 Ubiquitous duality

We have considered three forms of recursive data:

- `Fix f` — ordinary recursive data types, finite in eager languages;
- `∀a . (f a -> a) -> a` — Church encoding, finite structures defined by folds;
- `∃s . (s -> f s, s)` — co-Church encoding, potentially infinite data structures, data abstractions (working through an interface over a hidden representation).

In this section we consider some practical examples in which the solution space is defined by dual representations of collections.

4.6.1 Push vs pull streaming

Let us consider various representations of lists or, as they are sometimes called, event streams. A more detailed comparison can be found here [Kiselyov et al., 2017, section 3].

Church encoding for lists gives a **push stream**, or **internal iteration**, or **Observer pattern**[90] (in a sense a special case of the Visitor pattern 4.4.2), where we register an event handler and the

[88] Full deforestation can be guaranteed using staging (see later 9) Kiselyov et al. [2017].
[89] https://okmij.org/ftp/Streams.html
[90] https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1?view=net-9.0

collection calls (pushes) methods on it. Examples of push streams include the Java Stream API[91], Clojure Transducers[92].

```
events.observe(object : Observer<Event> {
    override fun onComplete() { .. }
    override fun onNext(elem: Event) { .. }
})
```

co-Church encoding for lists gives a **pull stream**, or **external iteration**, or **Iterator pattern**. To get the next element, the consuming code must explicitly request (pull) it. This representation is often much more convenient from a usage perspective (see 4.5.2)[93], but is more complex on the implementation side, because one needs to identify and maintain the existential state in the coalgebra. Generators are designed to automate the identification of this state, since one can notice that it is the continuation of the computation (see later 6.4.3).

Working with streams is often called **reactive programming**[94].

4.6.2 Data vs codata

Recall the shape functor for lists:

```
data ListF a r = Nil | Cons a r
```

Using the `data` keyword we define a new algebraic data type through the ways to construct it. Then, using pattern matching we can deconstruct values of an algebraic data type and use the information stored in it.

```
data List a where
  List :: ListF a (List a) -> List a

fold :: (ListF a r -> r) -> List a -> r
fold alg xs = case xs of
  List Nil -> alg Nil
  List (Cons y ys) -> alg (Cons y (fold alg ys))
```

However, one can also define a data type through the ways to deconstruct it using the `codata` keyword (in an imaginary language). Thus, from a co-list one can request the next layer of the structure. Such a data type can be constructed using co-pattern matching, describing how the deconstructing function should behave on a constructing term (`unfold coalg s`)[95].

[91] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
[92] https://clojure.org/reference/transducers
[93] https://github.com/pulldown-cmark/pulldown-cmark/?tab=readme-ov-file#why-a-pull-parser
[94] (youtube) React 2014 : Erik Meijer - What does it mean to be Reactive?
[95] (youtube) CS410 2017 Lecture 15 Coinduction and Coalgebras

```
codata CoList a where
  force :: CoList a -> ListF a (CoList a)

unfold :: (s -> ListF a s) -> s -> CoList a
force (unfold coalg s) = case coalg s of
  Nil -> Nil
  Cons x s' -> Cons x (unfold coalg s')
```

Essentially, `codata` represents a dictionary of functions that capture some state necessary for generating subsequent layers of the structure. Thus, `codata` can be compared to OOP interfaces, and co-pattern matching to an anonymous interface implementation. The two preceding examples can be written, for example, in Kotlin as follows (the list shape is encoded as `Pair<a, r>?`, verify for yourself that this type is isomorphic to `ListF`).

```
class List<a>(val layer: Pair<a, List<a>>?)

fun <a, r> fold(alg: (Pair<a, r>?) -> r, xs: List<a>): r =
    if (xs.layer == null) alg(null)
    else alg(Pair(xs.layer.first, fold(alg, xs.layer.second)))

interface CoList<a> { fun force(): Pair<a, CoList<a>>? }

fun <a, s> unfold(coalg: (s) -> Pair<a, s>?, ini: s): CoList<a> =
    object : CoList<a> {
        override fun force(): Pair<a, CoList<a>>? {
            val layer = coalg(ini)
            return if (layer == null) null else
                Pair(layer.first, unfold(coalg, layer.second))
        }
    }
```

The functional style of programming primarily operates on directly observable data `data`, while OOP hides data behind interfaces `codata`. However, both constructs have their clear areas of application and should not be neglected regardless of preferred style Downen et al. [2019][96].

In languages with totality checking, inductive and coinductive definitions are distinguished. For them, pattern matching and co-pattern matching, the induction principle (catamorphism) and the coinduction principle (anamorphism) are used. Inductive definitions specify finite data structures; totality checking verifies that recursive calls are made on structurally smaller subterms. Coinductive definitions specify potentially infinite data structures; for them, productivity is checked — the generating function always produces a new constructor after a recursive call[97].

[96] https://reasonablypolymorphic.com/blog/review-codata/

[97] https://rocq-prover.org/doc/V8.18.0/refman/language/core/coinductive.html

4.7 Appendix: category of algebras

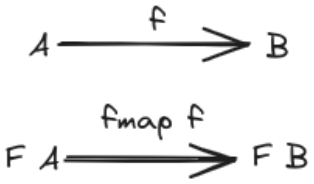
This is an optional section that is not necessary for understanding the rest of the course. However, it is useful for understanding frequently used terminology.

A **category** is a collection of objects and a collection of arrows. For each object X there exists an identity arrow, and for each pair of arrows there exists a way to obtain their composition: $f : Y \to Z, g : X \to Y \Rightarrow f \circ g : X \to Z$.

One defines a category corresponding to Haskell — *Hask*. In reality this is a poor category from the theoretical standpoint, but it will suffice for our informal reasoning[98]. Objects in Hask are types of the Haskell language, and morphisms are terms defining functions between the corresponding types. The identity morphism is `id`, and composition is defined as `f . g = \x -> f (g x)`.

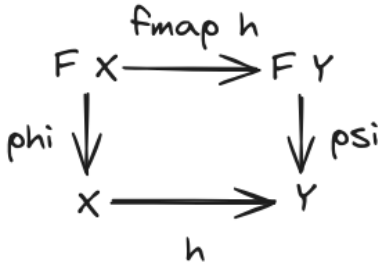
Exercise 32 *How does one represent constants in such a category?*

A **functor** is a mapping between categories that maps objects of one category to objects of another, and arrows of one to arrows of the other. In Haskell, type constructors define the mapping between objects, and `fmap` defines the mapping between arrows. A functor must preserve the identity morphism and composition.



An **algebra** in a category C is a pair of an object of the category $X \in Obj(C)$ and a morphism $\phi : F\ X \to X$, where F is a functor. The morphism $F\ X \to X$ itself is called an **f-algebra**. Algebras in the categorical sense can describe algebras. Thus, as the object X we take the carrier of the algebra. As the functor F — the signature of the algebra in the form of a shape type. Then the morphism is the interpretation of the signature.

An **algebra morphism** is a morphism between carriers $h : X \to Y$ such that the following diagram commutes. A **diagram commutes** if all possible paths along the arrows in it are equal.



[98] https://math.andrej.com/2016/08/06/hask-is-not-a-category/

In an algebra morphism one can recognize familiar features of homomorphisms, that is, operations between carriers that "respect" the operations of the algebraic theory's signature.

Algebras over a category C form a **category of algebras**, in which the objects are algebras and the morphisms are algebra morphisms.

An **initial object** of a category is an object from which there exists a unique arrow to every other object. A **terminal (final) object** of a category is an object into which there exists a unique arrow from every other object.

Initial and terminal objects of a category do not necessarily exist in a single instance. But all initial objects are isomorphic to each other, as are all terminal objects.

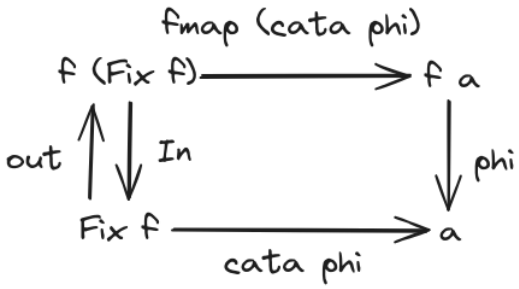
Exercise 33 *Give the initial and terminal objects of the category Hask.*

A recursive type is a type, which means it corresponds to an object in the category Hask. X is a recursive type with shape F if the following isomorphism holds:

$$X \simeq F\ X$$

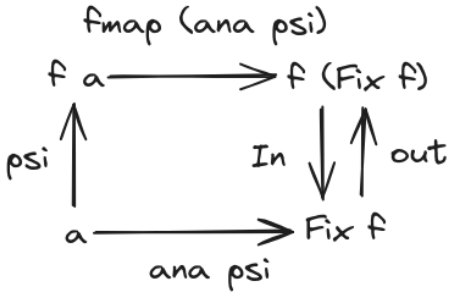
One can notice that the isomorphism witness from right to left resembles an f-algebra, and from left to right an f-coalgebra (everything is the same, but with all arrows reversed). And indeed, the appropriate object X must be either the initial object of the category of algebras, or the terminal object of the category of coalgebras (with the corresponding morphisms). The first variant corresponds to finite data structures, and the second to potentially infinite ones.

The initial object of the category of algebras over *Hask* for the functor f is the following algebra: `(Fix f, In)`[99][100]. Indeed, for every type `a` and for every f-algebra `phi` we can construct a morphism `cata phi :: Fix f -> a` such that the following diagram commutes:



Let us show that `(Fix f, out)` is the terminal object of the category of coalgebras (thanks to Haskell's laziness). Analogously, for every object `a` and f-coalgebra `psi` there exists a morphism `ana psi :: a -> Fix f`:

[99] https://bartoszmilewski.com/2017/02/28/f-algebras/
[100] https://ncatlab.org/nlab/show/catamorphism


fmap (ana psi)
f a
f (Fix f)
psi
In
out
a
Fix f
ana psi

5 Interpreters

The art of programming largely consists of the ability to manage complexity, and within this course we will consider building languages and interpreters as the primary tool for this.

5.1 Interpreters as the foundation

We begin with an overview of the role of interpreters in programming.

5.1.1 The tower of interpreters

The most basic interpreter is the processor, physically embodied in hardware. It receives a program in some language, such as x86, reads instructions, and turns them into actions on memory. However, it is difficult for a human to program in this language; a new language is needed that encapsulates part of the complexity and hides unnecessary details.

To obtain a new language, we build a software interpreter. A **software interpreter** U_M^N is a program in language M[101] that takes as input a program in language N and its input — data from D, and returns the result of executing that program on that data:

$$U_M^N : N \times D \to D$$

We will call the implementation language of the interpreter M the **meta-language**, and N the **object language**. One can intuitively think of an interpreter as follows: it is an explanation, understandable to the meta-language, of what the constructs of the object language mean. In other words, which meta-language instructions need to be executed to obtain the desired semantics of the object language instructions.

For example, given a program p_N and its input data d_{in}, the result of executing this program d_{out} can be obtained as

$$d_{out} = U_M^N \left(\underbrace{\langle p_N, d_{in} \rangle}_{\in N \times D} \right)$$

But an interpreter is also a program. How do we run it? Take our basic interpreter U^{x86}; it has no implementation language since it is implemented in hardware, not software. Take an assembly language interpreter implemented in x86 codes, U_{x86}^{Asm}, an assembly program p_{Asm}, and its input d_{in}. Recall that a program is also data, just in a special format. Then we obtain the result of applying p_{Asm} to the data as follows:

$$d_{out} = U^{x86} \left(\left\langle \underbrace{U_{x86}^{Asm}}_{\in x86}, \underbrace{\langle \overbrace{p_{Asm}}^{\in Asm}, \overbrace{d_{in}}^{\in D} \rangle}_{\in D} \right\rangle \right)$$

[101] By a language we mean the set of programs in that language, in other words, the set of trees of a certain form.

But assembly language is also not very pleasant for programming. However, in it one can already write an interpreter for a more complex language. And so on. We obtain a *tower of interpreters*, at the top of which is the language in which we want to solve our actual problem:

$$d_{out} = U^{x86}\left(\left\langle U^{Asm}_{x86}, \left\langle U^{C}_{Asm}, \left\langle U^{Has}_{C}, \langle p_{Has}, d_{in}\rangle\right\rangle\right\rangle\right\rangle\right)$$

In practice, a language is often defined via translation (compilation) into another. However, we will focus on embedded languages going forward, and the notion of an interpreter will be more useful to us. Moreover, there exist universal theoretical methods for deriving a compiler from an interpreter[102] (see 9), so for us they are similar.

5.1.2 Interpreters everywhere

Very well, we have arrived at the language of our heart (Haskell); why do we continue talking about interpreters? Because for solving concrete business problems, general-purpose languages are still too ceremonious — the programmer has to think about a large number of things irrelevant to their domain and the problem at hand. Complexity is the programmer's main enemy, because the human brain's resources are no match for the complexity of the reality our programs must describe. Thus, in practice one constantly has to describe new languages most suited for solving specific applied problems. And new languages are defined using interpreters.

What does a classical recursive interpreter look like? It receives a program in the form of some tree and recursively traverses it, computing the results of subtrees. When it visits a tree node, it determines its type and understands what actions need to be performed. That is, the node type dispatches, navigates, the interpreter's execution to the appropriate code. Thus, an interpreter for a simple expression language looks as follows:

```haskell
eval :: Expr -> Int
eval prog = case prog of
  Const x -> x
  Plus l r -> eval l + eval r
```

One can see that this resembles, for example, a command-line utility — we parse arguments, determine what needs to be done and how, and do it. Surprisingly, the Unix philosophy, in particular, consists of building small languages (utilities with a text API) that solve one problem well Bentley [1986]. It also resembles handling a request by a web server — determining the route the request arrived at and performing the corresponding action. That is, we write interpreters in real life more often than we think. We simply do not see that what we write is actually an interpreter for some language. In general, folding a data structure can already be considered as interpretation Gibbons and Wu [2014].

Moreover, as we will see later 5.4, writing any function is already defining a new language. There was a language in which one could not add a user to an application. We wrote a function `registerUser` — a new command appeared in the language — add a user. Below we will formally

[102] `https://en.wikipedia.org/wiki/Partial_evaluation`

show that this approach is equivalent to adding a new node to the language's syntax tree (5.4.1). Using functions is an example of language embedding, when instead of creating a new separate language, we implement it as a library for an already existing language Gibbons [2013].

As we will see throughout the course, many problems can be viewed as inventing a language and implementing an interpreter. This means that if we learn to write interpreters well, we will learn to do a lot of things at once! And our main efforts will be directed at studying the tools for building interpreters of embedded languages.

During the exposition we frequently use the Hutton's Razor technique, which involves considering a ridiculously small language for studying complex concepts. It is argued that for studying most questions one can construct such a language that makes everything important maximally clear.

5.1.3 Interpreters and programming language semantics

Programming language semantics[103] is a discipline studying the properties of languages and the meaning of programs, its properties, and methods of description. An excellent introduction is the Software Foundations book series[104] Pierce et al. [2010]. There are many different styles of describing program semantics; the most important for us will be denotational semantics.

Denotational semantics[105][106][107] describes the meaning of programs by associating them with objects of some set, the **semantic domain**. In other words, the denotational semantics of a language L is a total function from a program in that language to an element of the domain D:

$$[\![\bullet]\!] : L \mapsto D$$

The domain is chosen based on the language and the information one wants to extract from programs. For example, to find out the size of a program (here, expressions with addition), one can take natural numbers as the domain:

$$\begin{array}{lcl} [\![n]\!] & = & 1 \\ [\![l + r]\!] & = & 1 + \max([\![l]\!], [\![r]\!]) \end{array}$$

If we are interested in the final result, we can compute it:

$$\begin{array}{lcl} [\![n]\!] & = & n \\ [\![l + r]\!] & = & [\![l]\!] + [\![r]\!] \end{array}$$

If the program has input, the domain will be a function $\mathbb{N} \to \mathbb{N}$:

$$\begin{array}{lcl} [\![n]\!](m) & = & n \\ [\![l + r]\!](m) & = & [\![l]\!](m) + [\![r]\!](m) \\ [\![input]\!](m) & = & m \end{array}$$

[103] https://en.wikipedia.org/wiki/Semantics_(computer_science)
[104] https://coq.vercel.app/ext/sf/
[105] https://en.wikipedia.org/wiki/Denotational_semantics
[106] https://en.wikibooks.org/wiki/Haskell/Denotational_semantics
[107] (youtube) The Lost Art of Denotational Semantics — Eric Meyer.

Thus, *a program is merely a syntactic notation for some element of the semantic domain*. There are many possible domains; they can even be games[108].

Exercise 34 *Into which domain is it reasonable to interpret programs in a language with mutable integer variables? And in a nondeterministic language?*

It is easy to see that the denotational semantics of a language is simply an interpreter, only written in the language of mathematics. Such interpreters are placed at the base of the tower of interpreters when the goal is to study properties of languages and programs rather than to execute them.

Also, we can implement an interpreter in some real language, and it too will define the semantics of the object language. However, the formality of such a definition will depend on the formality of the meta-language's semantics description. Such interpreters are called **definitional**; they define the semantics of a language, typically sacrificing efficiency for clarity. The relationships between the object language and the meta-language are studied in the classical papers Reynolds [1972, 1998][109].

We will use definitional interpreters to define the semantics of new languages, using Haskell as the meta-language. And as domains we will take Haskell types. And we will interpret programs not into the set of functions between natural numbers, but, say, into the type `Nat -> Nat` in Haskell[110]. Thus, the denotational semantics of the language of sums with input will be written as follows:

```
1 eval :: Prog -> (Nat -> Nat)
2 eval = \case
3   Val n -> \_ -> n
4   Plus l r -> \m -> eval l m + eval r m
5   Input -> \m -> m
```

A semantics is called **compositional** if the meaning of constructs depends only on the meaning of subconstructs. In other words, if the denotational semantics is a fold over the program (Fig. 17) and can be written in terms of a catamorphism 4.3.3.

Another popular style of describing semantics is **operational semantics**, which represents the meaning of a program as a sequence of computation steps. This can be either sequential rewriting of the expression itself, or rewriting the state of some abstract machine. Operational semantics, in turn, is defined as an unfold (or anamorphism 4.3.3) of the sequence of computation steps from the program. Here one can clearly see a certain duality between denotational and operational semantics Hutton [1998].

5.1.4 Embedded domain-specific languages (eDSL)

By **domain-specific languages (DSL)**[111] one often means specialized languages for specific domains, such as database queries or document formatting. As a rule, such languages are not Turing-complete.

[108] https://en.wikipedia.org/wiki/Game_semantics

[109] The notion of continuation, actively used by the author, will be considered later in this course (section 6).

[110] https://okmij.org/ftp/Denotational.html

[111] https://en.wikipedia.org/wiki/Domain-specific_language

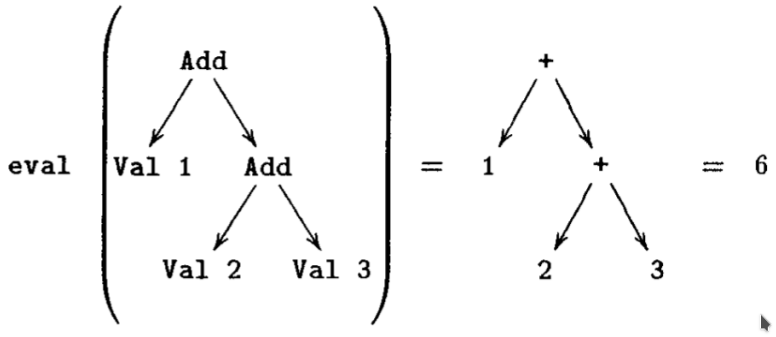


Figure 17: Denotational semantics defines the meaning of syntactic constructs through operations on the domain Hutton [1998].

In this course, however, we will consider any domain-specific specialization of a general-purpose language to be a domain-specific language[112]. This follows from the consideration that code should read like well-written prose with appropriate word usage, providing the reader with only the necessary amount of detail, hiding the inessential behind conventions and terminology.

Standalone domain-specific languages are languages that have their own concrete syntax as well as programming tools (IDE, runtime...). Examples: SQL, AWK, Antlr...[113]

Embedded domain-specific languages (eDSL) are languages that leverage the infrastructure of other languages. They are usually implemented as libraries for programs in an already existing general-purpose language. They do not have a fully independent concrete syntax. Examples: ORM, string processing functions, parser combinator libraries...

Deep eDSL — terms in such a language build an abstract syntax tree for further interpretation:

```
f :: Int -> Int
f = eval $ Const 1 `Plus` Input
```

Interpreters that take trees as input and interpret them into a semantic domain are called **initial**[114]; we will also call them "classical." We will start with them. Classical interpreters are useful, for example, for implementing the "last language" — the program's interface to the outside world, and as a foundation for our further constructions.

However, one can notice that the intermediate tree that is produced typically does not interest us. We only care about obtaining the domain element, which we already know how to use directly. **Shallow eDSL** skip the tree construction stage and directly build the value in the semantic domain. We will call such interpreters **final**.

```
cnst :: Int -> (Int -> Int)
cnst x _ = x
```

[112] https://en.wikipedia.org/wiki/Language-oriented_programming.

[113] There is an excellent practical book Nystrom that covers building interpreters and simple virtual machines. At the same time it covers the creation of a full-fledged language in all its aspects, from parsing to memory management.

[114] The syntax type defined using `data` is the initial object of the category of interpretations.

```haskell
input :: Int -> Int
input env = env

plus :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)
plus l r env = l env + r env

f :: Int -> Int
f = cnst 1 `plus` input
```

Interpreters are often called **observers**, which analyze terms and give them some meaning Gibbons [2013]. One can notice that for deep eDSL one can write arbitrarily many different observers. However, in the case of shallow embedding, observers are always `id`. We will discuss possible solutions to this problem later in section 5.4.1.

A discussion of terminology and comparison of approaches to DSL construction can be found in Gibbons [2013]. A brief description of terms is in the Language Engineering course notes Hutchinson.

Let us introduce one more important concept. A **meta-circular interpreter**[115] is an interpreter that defines constructs of the object language through constructs of the meta-language Reynolds [1972]. For example:

```haskell
interpret term = case term of
  App f t -> (interpret f) (interpret t)
  If c t e -> if interpret c then interpret t else interpret e
  ...
```

The properties of the meta-language in this case largely determine the properties of the object language Reynolds [1972, 1998]. In this course we will strive to reuse the capabilities of the meta-language as much as possible.

Exercise 35 *Guess which properties the object language from the example above inherits.*

5.1.5 Example: the Accelerate library

An interesting example of an embedded language that lies somewhere between deep and shallow is the Accelerate library[116] [Marlow, 2011, chapter 6]. It allows describing computations in Haskell that will be executed on a GPU[117].

To execute something on a GPU, one needs to generate and compile Cuda code. Thus, the Accelerate language must be a deep embedding in order to have the computation tree for translation into Cuda in the most efficient manner.

[115] `https://en.wikipedia.org/wiki/Meta-circular_evaluator`

[116] `https://hackage.haskell.org/package/accelerate`

[117] Another approach: Java code reflection, to extract a model of the code at runtime. However, such an approach does not provide static guarantees to the programmer and requires deep integration into the meta-language.

At the same time, describing numerical computations as a tree is extremely inconvenient. It would be nice to have familiar operators and higher-order functions for working with arrays on the GPU. Therefore, Accelerate actually provides a shallow interface for building trees. Thus, for expression trees, implementations of numeric type classes are defined, e.g. `Num`, where operations simply extend the tree:

```
example :: Acc (Vector Int) -> Acc (Vector Int) -> Acc (Vector Int)
example xs ys = A.zipWith (+) xs ys
```

5.2 Value types

Let us consider the possible ways to implement languages whose values can have different types.

5.2.1 Untyped tagless interpreters

First, let us consider a trivial untyped language. By untyped we mean the absence of type checking both before and during program execution. We define the abstract syntax of this language as follows:

```
data Expr = Const Int | IsZero Expr | If Expr Expr Expr
```

Values arising during the execution of programs in this language will be represented by values of types `Bool` and `Int` of the Haskell language. Accordingly, the semantic domain of a program in this language is either `Bool` or `Int`, depending on the program itself.

```
evalUnsafe :: Expr -> forall res . res
evalUnsafe = \case
  Const val -> unsafeCoerce val
  IsZero cond -> unsafeCoerce $ evalUnsafe @Int cond == 0
  If c t e -> if evalUnsafe c then evalUnsafe t else evalUnsafe e
```

Here `unsafeCoerce` is used to circumvent Haskell's static type system and simply execute programs in our untyped language. We are entitled to do this because `Int` and `Bool` in Haskell have the same size. Writing an incorrect program in this language or choosing the wrong interpretation domain leads to a crash.

5.2.2 Typed tagged interpreters

To achieve some safety during execution, we will attach tags to values that will be available at runtime. We introduce the following algebraic type:

```
data RtValue = RtBool Bool | RtInt Int
```

Now the semantic domain will be the type `RtValue`, and the interpreter can check types at runtime:

```haskell
evalRt :: Expr -> RtValue
evalRt (IsZero expr) = case evalRt expr of
  RtBool value -> error "Type error"
  RtInt value -> RtBool (value == 0)
-- ...
```

The safety situation has definitely improved, but type checking at runtime is already too late: it requires additional performance overhead and makes testing more expensive.

This approach is often called **dynamic typing**, when we attribute values with some type information for use at runtime (see also 3.4.6).

5.2.3 Typed tagless interpreters

Let us describe the type system of our small language.

$$\frac{n : Int}{Const\ n : int}\ Const \qquad \frac{n : int}{IsZero\ n : bool}\ IsZero \qquad \frac{c : bool \quad t : \tau \quad e : \tau}{If\ c\ t\ e : \tau}\ If$$

We can reuse Haskell types as type tags:

$$int \rightsquigarrow Int, bool \rightsquigarrow Bool$$

Notice that with the help of a generalized algebraic data type `Expr ty` (2.1.4), we have precisely encoded these typing rules. In other words, we have obtained a statically typed programming language by reusing Haskell's type system.

```haskell
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
eval = \case
  Const n -> n
  IsZero e -> eval e == 0
  If c t e -> if eval c then eval t else eval e
```

Thanks to static typing, we can dispense with tagging values during interpretation without losing safety.

5.3 Bindings and first-class functions

In this section we consider techniques and concepts related to implementing first-class functions and bindings in general. Since this functionality is typically already implemented in the meta-language, we will strive to reuse it as much as possible.

`let`-bindings can be represented through first-class functions as follows:

$$\textbf{let } x := N \textbf{ in } M \equiv (\lambda x.\, M)\ N$$

Recall that first-class functions can be eliminated using defunctionalization, considered earlier 3.1.7.

5.3.1 Name semantics

There are several ways to assign semantics to identifiers.

Dynamic scoping — the values of free variables of a function depend on the scope at the call site. That is, name resolution happens at the moment the variable is accessed. For example, the following code prints 42:

```
val x = 4
val f = () => x + 1
val x = 41
println(f())
```

This approach is simpler to implement and was used in early versions of Lisps, for example. However, in this case functions are not a reliable abstraction barrier; ideally all free variables should be part of the signature (we will return to this in the chapter on effect systems 8).

Lexical/static scoping — variables are bound to values at the point of function declaration; at the point of invocation the result depends only on the parameters (modulo mutable state[118]). The word "lexical" is often used in languages when we can determine something from the source code without running the program. Thus, the code from the example above would print 5.

In the rest of this section we will discuss various ways of implementing first-class functions with static variable scoping, of which there are, in fact, a great many[119].

5.3.2 Substitutions

As one can notice, in classical lambda calculus, substitutions from beta-reduction (recalled in section 1.1) provide static scoping. Indeed, the argument is immediately substituted into all occurrences of the variable, so it does not remain free but simply disappears.

$$(\lambda x.\, (\lambda x.\, \lambda y.\, x + y)\ 4)\ 41 \rightsquigarrow (\lambda x.\, (\lambda y.\, 4 + y))\ 41$$

This approach is not the most efficient, because each application requires rewriting the entire function body (!). At the same time, it is fairly simple to implement for certain representations of lambda terms. Let us consider an example of such a representation — **locally nameless** Charguéraud [2012].

[118] For example, in Kotlin, mutable variables can be captured in lambdas. Changes outside are observable inside the lambda, and vice versa. This can sometimes be very convenient, but often leads to very non-obvious behavior.

[119] `https://jesper.cx/posts/1001-syntax-representations.html`

```haskell
data Term var
  = Var var
  | App (Term var) (Term var)
  | Lam (Term (Maybe var))
```

In this representation one can choose any type for naming free variables:

```haskell
example :: Term String
example = Var "x" `App` Var "y" -- x y
```

Adding each bound variable adds a new inhabitant `Nothing` to the variable type for referring to the nearest bound variable:

```haskell
-- λx.x y
example1 = Lam $ Var Nothing `App` Var (Just "y")
-- λx y.x y z
example2 = Lam $ Lam $ Var (Just Nothing) `App` Var Nothing `App` Var (Just (Just "z"))
```

Monadic bind is the implementation of substitution for such terms:

```haskell
instance Monad Term where
  (>>=) :: Term var -> (var -> Term var') -> Term var'
  Var var >>= subst = subst var
  App l r >>= subst = App (l >>= subst) (r >>= subst)
  Lam t >>= subst = Lam $ t >>= \case
    Nothing  -> Var Nothing
    Just var -> Just <$> subst var
```

Exercise 36 *Think about why `fmap Just` is needed in the last line.*

Accordingly, a call-by-name interpreter for this lambda calculus will look as follows:[120]

```haskell
eval :: Term var -> Term var
eval = \case
  Var var -> Var var
  App f arg -> case eval f of
    Lam body -> eval $ body >>= maybe arg Var
    t -> App t (eval arg)
  Lam t -> Lam (eval t)
```

[120] Since we are considering classical λ-calculus, the resulting value is also a term, but in normal form.

5.3.3 Environment

One can perform substitution of variable values lazily, propagating an environment that maps free variables to terms. Things have not yet improved much overall, but we have obtained compositional semantics from non-compositional semantics by making context dependencies explicit (more on this later 5.4.3).

```
1  data Term1 = Var1 String | App1 Term1 Term1 | Lam1 String Term1
2  type Env = Map String Term1

3  eval1 :: Term1 -> Env -> Term1
4  eval1 term env = case term of
5    Var1 name -> Map.findWithDefault (Var1 name) name env
6    App1 f arg -> case eval1 f env of
7      Lam1 name body -> eval1 body (Map.insert name arg env)
8      t -> App1 t (eval1 arg env)
9    Lam1 name body ->
10     let env' = Map.delete name env in
11     Lam1 name (eval1 body env')
```

Exercise 37 *Explain why the environment is modified on line 10?*

If the `Lam1` branch does not recursively traverse the subterm and substitute variable values, the information about free variable values in it will be lost and we will get dynamic scoping instead of static scoping. To restore static scoping, the `Lam1` branch of the interpreter must construct a closure that includes the current environment (see 5.3.4).

5.3.4 Closures

To avoid eagerly performing substitution into function bodies while preserving static scoping, we add one more construct, a **closure**[121][122] [Nystrom, chapter 11]. It will store the context in which the corresponding term should be executed.

```
1  data Term1 = Var1 String | App1 Term1 Term1 | Lam1 String Term1
2             | Closure Env String Term1 -- only for computations
3  type Env = Map String Term1

4  eval1 :: Term1 -> Env -> Term1
5  eval1 term env = case term of
6    Var1 name -> Map.findWithDefault (Var1 name) name env
7    App1 f arg -> case eval1 f env of
```

[121] https://en.wikipedia.org/wiki/Closure_(computer_programming)
[122] The term closure was proposed by Peter Landin, along with a bunch of other things.

```
      Closure env' name body ->
          let arg' = eval1 arg env in
          eval1 body (Map.insert name arg' env')
      t -> App1 t (eval1 arg env)
    Lam1 name body -> Closure env name body
```

Closures are typically used in industrial languages as the runtime representation of higher-order functions. During compilation, **closure conversion** is performed first — higher-order functions are represented as a pair of an environment and a pointer to a function that takes the environment as an extra argument. Now that the function contains no free variables, **lambda lifting**[123] is done — it is lifted to the top level. Detailed examples can be found in the Harvard slides Chong.

5.3.5 Typed context

Let us consider the encoding described, for example, in Kiselyov [2012a].

First, let us learn to use Haskell's type system to check the validity of environment lookups. We represent the environment as a list of types encoded using nested pairs:

```
(4, (4.0, "hello")) :: (Int, (Double, String))
```

Environment lookups will be encoded as numbers in unary notation. The type of the number (a typed reference into the context) specifies the set of environments `env` from which a value of type `ty` can be extracted at the given position.

```
data Ref env ty where
  Here :: Ref (ty, env) ty
  There :: Ref env ty -> Ref (ty', env) ty
```

For example, the type of the number 1 asserts that it can be used to extract a value of type `ty` from a context where a value of the corresponding type is at position one (zero-indexed):

```
There Here :: Ref (ty', (ty, env)) ty
```

Now we can encode a typed safe lookup into the context:

```
envLookup :: env -> Ref env ty -> ty
envLookup env ref = case (ref, env) of
  (Here, (x, _)) -> x
  (There ref', (_, env')) -> envLookup env' ref'
```

Exercise 38 *Can the pair be deconstructed directly on line 2? Explain.*

[123] https://en.wikipedia.org/wiki/Lambda_lifting

5.3.6 Meta-circular interpretation

It would be extremely undesirable to implement bindings and first-class functions for an eDSL by ourselves. Let us build a meta-circular interpreter (see 5.1.4) that will reuse the meta-language's first-class functions to implement them in the object language.

Terms will now be annotated not only with resulting types but also with the types of environments needed for interpretation, considered earlier 5.3.5. An abstracted term has access to a larger environment.

```
data Term2 env ty where
  Var2 :: Ref env ty -> Term2 env ty
  App2 :: Term2 env (arg -> res) -> Term2 env arg -> Term2 env res
  Lam2 :: Term2 (arg, env) res -> Term2 env (arg -> res)
```

Now we can interpret abstraction as a Haskell function and application as application:

```
eval2 :: Term2 env ty -> env -> ty
eval2 term env = case term of
  Var2 ref -> env `envLookup` ref
  App2 f arg -> (eval2 f env) (eval2 arg env)
  Lam2 t -> \arg -> eval2 t (arg, env)
```

Exercise 39 *How did it happen that an extra argument needs to be accepted in the last line?*

Exercise 40 *Is this a call-by-value interpreter or call-by-name? What does this depend on?*

Exercise 41 *Think about which solution should be more performant, this one or the previous?*

Note that now the functions of the object language at runtime are simply functions of the meta-language. This means that in programs in the object language we *can fully reuse the meta-language*! Let us add a construct that allows storing an arbitrary meta-language value in the tree:

```
data Term2 env ty where
  Val2 :: ty -> Term2 env ty
  -- ...

eval2 :: Term2 env ty -> env -> ty
eval2 term env = case term of
  Val2 x -> x
  -- ...

example :: Term2 env (Int -> Int)
example = Lam (Val2 (+) `App2` Val2 1 `App2` Var2 Here)
```

5.3.7 Higher-order abstract syntax

Another thing we are still doing manually is defining binders (and in unary notation, no less). We want to reuse them from the meta-language. For this, we will store meta-language functions directly in the syntax tree — use **higher-order abstract syntax (HOAS)**[124][125] Pfenning and Elliott [1988]:

```
data Term3 ty where
  Val3 :: ty -> Term3 ty
  Plus :: Term3 Int -> Term3 Int -> Term3 Int
  App3 :: Term3 (arg -> res) -> Term3 arg -> Term3 res
  Lam3 :: (Term3 arg -> Term3 res) -> Term3 (arg -> res)

example3 :: Term3 Int
example3 = (Lam3 \x -> x `Plus` Val3 41) `App3` Val3 1
```

The interpretation is very simple and completely meta-circular:

```
eval3 :: Term3 ty -> ty
eval3 term = case term of
  Val3 x -> x
  Plus l r -> eval3 l + eval3 r
  App3 f arg -> (eval3 f) (eval3 arg)
  Lam3 f -> \arg -> eval3 (f (Val3 arg))
```

Exercise 42 *Could* `Lam3` *have been declared as follows?*

```
Lam3 :: (arg -> Term3 res) -> Term3 (arg -> res)
```

5.3.8 Serialization of functions

In this section we discussed possible implementations of first-class functions, i.e. functions that can be used as flexibly as data. A natural question arises: can we serialize a first-class function and send it for execution on another machine?

A function consists of code and captured free variables in the case of static scoping. Accordingly, if the code is in a serializable form (e.g. position-independent bytecode), it can in principle be sent over the network and executed on another virtual machine instance. This is what Erlang does, for example. However, this approach is inefficient, since the bytecode needs to be interpreted or pre-compiled. Thus, Erlang sacrifices execution speed for horizontal scalability.

If we guarantee that the same code runs on different cluster nodes, as is usually the case in practice, a more efficient implementation can be achieved. For example, using defunctionalization (see 3.1.7), we can serialize only the algebraic type objects encoding functions. Since the same code runs on the

[124] https://en.wikipedia.org/wiki/Higher-order_abstract_syntax

[125] What is higher-order in higher-order abstract syntax?

other cluster node, we can deserialize the object there and execute it using `apply`. However, this approach does not support modularity well (it is hard to split one algebraic type into many), and `apply` performs decoding before executing the code each time (which, admittedly, can be neglected given network overhead).

The approach implemented in Haskell[126][127] allows assigning each function without free variables a statically known address, the same for all application instances. Then one can construct a serializable closure through a sequence of partial applications:

```
data Closure a where
  StaticPtr :: StaticPtr b -> Closure b
  Encoded :: ByteString -> Closure ByteString
  Ap :: Closure (b -> c) -> Closure b -> Closure c

main = send "some-node" $
  closure (static factorial) `closureAp` closurePure 10
```

More details can be found in the foundational paper on Cloud Haskell Epstein et al. [2011]. From a practical perspective — in the book [Marlow, 2011, chapter 16].

Exercise 43 *Is it necessary to explicitly add the free variable (*) (the multiplication operator) to the closure in the factorial implementation?*

5.4 Tagless final interpreters

As we established earlier (5.1.2), programming consists of writing interpreters on top of each other. Interpreters define the semantics of new languages (5.1.3). In the classical form, a language is defined as a set of trees, and an interpreter maps trees to meta-language objects. If the language is embedded, this approach is called deep embedding (see 5.1.4), and the corresponding interpreter is initial.

$$[\![\bullet]\!] : L \to D$$

One can notice that ultimately we only use the domain element to which the interpreter maps the program. The program itself is merely a convenient syntactic notation for a domain element and is an intermediate step, not an end in itself. At the same time, the domain for embedded languages defined by interpreters consists of meta-language objects. Can we bypass the stage of interpreting our own syntax and directly build the domain object in the meta-language syntax? Yes, such an embedding is called shallow embedding (5.1.4); this chapter is about it.

[126] `https://blog.ocharles.org.uk/blog/guest-posts/2014-12-23-static-pointers.html`
[127] `https://hackage.haskell.org/package/distributed-closure`

5.4.1 Different interpretations for shallow embedding

As we learned earlier (4.4), any data structure can be represented as a fold. And, moreover, in the end one can do without a single data constructor (as in Church lists, for example): the algebra is represented by a set of functions, each responsible for folding a particular constructor.

```
Fix f ≅ forall a . (f a -> a) -> a
List e ≅ forall a . (e -> a -> a) -> a -> a
```

Thus, instead of constructing a language tree and then interpreting (folding) it, we can directly construct a term of type `∀a . (f a -> a) -> a`. By providing it with a domain type `a` and an algebra `f a -> a` (or as a bundle of functions), we immediately obtain an element of the desired domain. To specify a different interpretation, one needs to pass a different domain type and algebra:

```
example :: (Int -> a) -> (a -> a -> a) -> a
example cnst plus = cnst 1 `plus` cnst 41

ghci> example show (\l r -> l ++ " + " ++ r)
"1 + 41"
```

If we fix the interpretation, the argument functions can be implemented statically and simply referenced in the term. Thus, declaring functions can be thought of as extending some embedded domain-specific language. General discussions about shallow embeddings, folds, and libraries can be found in Gibbons [2013], Gibbons and Wu [2014]. Compare:

Deep	Shallow
The language syntax is defined by the set of valid tree nodes	A function declaration defines a new tree node: a call to that function
The interpreter, upon encountering each node, executes the corresponding meta-language code (a pattern-match branch) after computing subtrees	The interpreter, upon encountering a call, executes the function body code after computing arguments

5.4.2 Going all the way

Let us return to the uncurried version of folds: this is now a type accepting a tuple of functions. And a tuple of functions can be replaced by a type class. Then the class declaration will define the syntax of the embedded language, and instances for domains — the implementation. This approach is called **tagless final encoding**[128] and is effectively Church encoding of data with type classes and a set of tricks Carette et al. [2007], Kiselyov [2012a].

Let us again consider the language with addition:

[128] https://okmij.org/ftp/tagless-final/

```haskell
data Expr = Const Int | Plus Expr Expr

eval :: Expr -> Int
eval = \case Const x -> x; Plus l r -> eval l + eval r
```

Or via catamorphism:

```haskell
data ExprF rec = Const Int | Plus rec rec

eval :: Fix ExprF -> Int
eval = cata \case Const x -> x; Plus l r -> l + r
```

The corresponding tagless final encoding will look as follows:

```haskell
class Expr domain where
  cnst :: Int -> domain
  plus :: domain -> domain -> domain

instance Expr Int where
  cnst x = x
  plus l r = l + r
```

Now we can construct an ordinary Haskell term, specify the domain, and the type class machinery will supply the appropriate algebra automatically:

```haskell
example :: forall domain . Expr domain => domain
example = cnst 1 `plus` cnst 41

ghci> example :: Int
42
```

To add another interpretation, we implement an instance for a different domain:

```haskell
instance Expr String where
  cnst x = show x
  plus l r = l <> " + " <> r

ghci> example :: String
"1 + 41"
```

5.4.3 Recovering compositionality of semantics

A **semantics** is called **compositional** if the semantics of constructs depends only on the semantics of subconstructs (see 5.1.3). In other words, it can be defined by a catamorphism and, accordingly, by a type class instance in tagless final. That is, to be able to build a tagless

final implementation for any semantics, one needs to be able to universally turn non-compositional semantics into compositional ones.

Various code transformations are typically not compositional. As an example, consider pushing through unary negations:

```
data Expr1 = Lit Int | Add Expr1 Expr1 | Neg Expr1

transform1 :: Expr1 -> Expr1
transform1 = \case
  Lit x -> Lit x
  Add l r -> Add (transform1 l) (transform1 r)
  Neg (Lit x) -> Lit (-x)                                          -- problem
  Neg (Neg e) -> transform1 e                                      -- problem
  Neg (Add l r) -> Add (transform1 (Neg l)) (transform1 (Neg r)) -- problem
```

To recover compositional semantics, one needs to make context dependencies explicit using an arrow domain Kiselyov [2012a]:

```
data Ctx = CtxPos | CtxNeg
flipCtx = \case CtxPos -> CtxNeg; CtxNeg -> CtxPos

transform1' :: Expr1 -> (Ctx -> Expr1)
transform1' expr = case expr of
  Lit x -> \case CtxNeg -> Lit (-x); CtxPos -> Lit x
  Neg e -> \ctx -> transform1' e (flipCtx ctx)
  Add l r -> \ctx -> Add (transform1' l ctx) (transform1' r ctx)
```

From here one can obtain the tagless final version:

```
class Expr2 d where
  lit :: Int -> d
  add :: d -> d -> d
  neg :: d -> d

instance Expr2 d => Expr2 (Ctx -> d) where
  lit x = \case CtxNeg -> lit (-x); CtxPos -> lit x
  neg e = \ctx -> neg e (flipCtx ctx)
  add l r ctx = add (l ctx) (r ctx)
```

5.4.4 Typed tagless final interpreter

Let us consider our example of tagless initial encoding 5.2.3:

```
data Expr ty where
  Const :: Int -> Expr Int
```

```
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
eval = \case
  Const x  -> x
  IsZero t -> eval t == 0
  If c t e -> if eval c then eval t else eval e
```

To obtain a final encoding, we parameterize the domain by the resulting type of the expression:

```
class Expr (domain :: Type -> Type) where
  cnst :: Int -> domain Int
  isZero :: domain Int -> domain Bool
  if' :: forall ty . domain Bool -> domain ty -> domain ty -> domain ty

instance Expr Identity where
  cnst x = Identity x
  isZero (Identity x) = Identity (x == 0)
  if' (Identity c) t e = if c then t else e
```

Exercise 44 *Which domain would be suitable for printing expressions?*

5.4.5 Meeting old friends: `Applicative`, `Monad`

Consider the following language in initial encoding with higher-order abstract syntax (see 5.3.6, 5.3.7). On the right we rewrite it in tagless final, choosing appropriate names for pieces of syntax.

```
data Expr s ty where
  Val   :: ty -> Expr s ty
  App   :: Expr s (arg -> res)
        -> Expr s arg
        -> Expr s res


  LetIn :: Expr s ty
        -> (ty -> Expr s ty')
        -> Expr s ty'


  Get   :: Expr s s
  Put   :: s -> Expr s ()
```

```
class Applicative domain where
  pure  :: ty -> domain ty
  (<*>) :: domain (arg -> res)
        -> domain arg
        -> domain res

class Monad domain where
  (>>=) :: domain ty
        -> (ty -> domain ty')
        -> domain ty'

class MonadState s domain where
  get :: domain s
  put :: s -> domain ()
```

We implement a function that modifies a value in our new language:

```
--
modify :: (s -> s) -> Expr s s
modify f =
  Get `LetIn` \x ->
  Put (f x) `LetIn` \() ->
  Val x
```

```
--                        s -> (s, s)
modify :: (s -> s) -> State s s
modify f =
  get >>= \x ->
  put (f x) >>= \() ->
  pure x
```

And the painfully familiar interpretation:

```
eval :: Expr s ty -> s -> (s, ty)
eval = \case


  Val x -> \s -> (s, x)
  App fs xs -> \s1 ->
    let (s2, f) = eval fs s1 in
    let (s3, x) = eval xs s2 in
    (s3, f x)


  LetIn comp k -> \s ->
    let (s', x) = eval comp s in
    eval (k x) s'


  Get -> \s -> (s, s)
  Put s -> \_ -> (s, ())
```

```
newtype State s a = State
  { runState :: s -> (s, a) }

instance Applicative (State s) where
  pure x = State \s -> (s, x)
  fs <*> xs = State \s1 ->
    let (s2, f) = runState fs s1 in
    let (s3, x) = runState xs s2 in
    (s3, f x)

instance Monad (State s) where
  comp >>= k = State \s ->
    let (s', x) = runState comp s in
    runState (k x) s'

instance MonadState s (State s) where
  get = State \s -> (s, s)
  put s' = State \s -> (s', ())
```

Thus, *applicative functors are a meta-circular language with application*, and *monadic bind is essentially let-in in higher-order syntax*.

If we recall that let $x = M$ in $N \equiv (\lambda x.\, N)\ M \equiv [x \mapsto M]\, N$, then we understand that monadic bind is the substitution `a -> m b` of the computation `m b` in place of results of type `a` in the computation `m a`. Or, if we take an expression tree as `m`, then bind is the substitution of subtrees in place of variables:

```
data Expr var = Var var | Empty | Append (Expr var) (Expr var)
instance Monad Expr where
  (>>=) :: Expr var -> (var -> Expr var') -> Expr var'
  Var name >>= subst = subst name
  Empty >>= _ = Empty
```

```
6     Append l r >>= subst = Append (l >>= subst) (r >>= subst)
```

The idea to use the category-theoretic notion of a monad[129] in λ-calculus was originally proposed in Moggi [1988] for more conveniently writing denotational semantics[130]. This turned out to be so convenient at the time that monads began to be used ubiquitously in functional programming to extend simple functional languages with various powerful capabilities without changing the languages themselves Wadler [1990, 1992]. Moggi's principle was formulated:

"Computations of type α correspond to values of type $f\ \alpha$."

The use of applicative functors was proposed significantly later, to avoid naming intermediate computation steps when this is unnecessary McBride and Paterson [2008]. Thus, applicatives provide an embedded language of expressions, while monads provide a language of statements.

5.5 Expression problem

The **expression problem**[131] is a criterion for the expressiveness of a programming language, formulated by Wadler in 1998[132]. The question is: how easy is it to extend the syntax of an embedded language and add new interpretations? In other words, how easy is it to add new kinds of data and processing methods.

By "easy" we mean locality: does one need to modify various pieces of code for this? For example, if the language syntax is defined by an ordinary algebraic data type, then adding a new interpretation is "easy" — just add a new recursive function, but adding a new syntactic construct is "hard" — all interpreters must be changed:

```
1  data Expr
2    = Const Int
3    | Plus Expr Expr -- adding

4  eval :: Expr -> Int
5  eval = \case Const x -> x; Plus l r -> eval l + eval r

6  show :: Expr -> String
7  show = \case Const x -> show x; Plus l r -> show l ++ " + " ++ show r
```

If the language is defined, for example, using inheritance, then conversely, extending the syntax is easy — add a new class, but adding an interpretation is hard — add a method implementation in each class:

[129] https://ncatlab.org/nlab/show/monad+(in+computer+science)
[130] The notion of effect, often accompanying monads, will be considered later 7.
[131] https://en.wikipedia.org/wiki/Expression_problem
[132] https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

```kotlin
interface Lang {
    fun eval(): Int
    fun show(): String
}
class Const(val x: Int) : Lang {
    override fun eval() = x
    override fun show() = x.toString()
}
class Plus(val l: Lang, val r: Lang) : Lang {
    override fun eval() = l.eval() + r.eval()
    override fun show() = "$l + $r"
}
```

It turns out there exist approaches that achieve "ease" along both dimensions. We will devote much attention to them in this course.

Indeed, as we discussed earlier 5.1.1, programs are a series of interpreters. For the same fight against complexity, it is important to be able to describe these interpreters modularly — defining parts of languages independently of each other and assembling the needed languages on the spot from ready-made blocks. This helps compose programs from simple reusable components, each with a clear area of responsibility.

The expression problem has arisen and been solved many times: expression problem, stable denotations, extensible (modular) interpreters. It took a long time before the understanding emerged that all of these are about the same thing[133].

5.5.1 Coproduct of functors

Let us use the representation of data as a fixed point of a functor (see 4.3.2). As a model language we take the expression language with addition:

```haskell
data Basic rec = Const Int | Plus rec rec

algBasic :: Basic Int -> Int
algBasic = \case Const x -> x; Plus l r -> l + r

evalBasic :: Fix Basic -> Int
evalBasic = cata algBasic
```

Note that the sum (coproduct) of shape functors gives a shape functor whose algebra is obtained from the algebras of the components[134]:

```haskell
data (l :+: r) rec = L (l rec) | R (r rec)
```

[133] https://okmij.org/ftp/Computation/having-effect.html

[134] We use the TypeOperators extension.

```
(\/) :: (l a -> a) -> (r a -> a) -> ((l :+: r) a -> a)
phi \/ psi = \case L l -> phi l; R r -> psi r
```

Let us extend our language with reading a number from the environment. Following the denotational semantics considered earlier 5.1.3, we choose the function `Int -> Int` as the domain:

```
data Basic rec = Const Int | Plus rec rec
data Input rec = Input

algBasic' :: Basic (Int -> Int) -> Int -> Int
algBasic' = \case Const x -> \_env -> x; Plus l r -> \env -> l env + r env

algInput :: Input (Int -> Int) -> Int -> Int
algInput = \case Input -> \env -> env
```

Thus, we have achieved the ability to separately define parts of the syntax and semantics of a language and assemble the needed language on the spot. Let us write a program in our language with implicit immutable state:

```
f :: Int -> Int
f = cata (algBasic' \/ algInput) $
  In (L (Plus (In (L (Const 1))) (In (R Input)))) -- 1 + input
```

However, note that we have not yet fully solved the problem, since interpreting the new construct `Input` required a more complex domain and we had to rewrite the interpretation of the old one: from `algBasic` to `algBasic'`. Now we understand why stable denotations is another name for the expression problem 5.5. We will complete this solution later 7.

5.5.2 Product of algebras

Let us build a tagless final representation for the interpreter from the previous section. Now the syntax is defined by type classes:

```
class Basic domain where
  cnst :: Int -> domain
  plus :: domain -> domain -> domain

class Input domain where
  input :: domain
```

As before, we take functions `Int -> Int` as the domain and implement the interpretation:

```
instance Basic (Int -> Int) where
  cnst value = \_ -> value
```

```
  plus l r = \env -> l env + r env

instance Input (Int -> Int) where
  input = \env -> env
```

Now, to assemble the language on the spot, one can simply take advantage of the composability of type class contexts; all the rest of the work:

```
example :: forall domain . (Basic domain, Input domain) => domain
example = cnst 1 `plus` input

ghci> (example :: Int -> Int) 41
42
```

6 Continuations

Since the early 1960s, continuations have appeared in the literature multiple times in various forms and diverse applications Reynolds [1993], Landin [1997], until in the 1970s Wadsworth coined a general term and a unified concept — **continuation**[135] — "the meaning of the rest of the program".

The initial impetus for these reflections was the Algol 60 language, which had a non-trivial mechanism of labels and jumps. The problem was both the implementation of the semantics and its denotational description together with translation into lambda calculus. Indeed, how does one mathematically describe `goto`? In which domain should one seek the semantics of such programs? How does one write a definitional interpreter mapping a program into this domain? The solution turned out to be the ability to refer to the semantics of the remainder of the program, the continuation, at a given point (for example, at a label).

The language designer must always think about continuations, since the evaluator at each point of a program needs to know what to execute next. And as we have already understood, every programmer is the author of many languages that bring them closer to solving their problem. Moreover, many everyday language constructs explicitly or implicitly directly manipulate continuations.

6.1 The concept of continuations

Let us consider the semantics of an expression from the operational point of view, as a sequence of rewriting steps (see 5.1.3). If we carefully examine each step, we discover that it consists of two stages: finding a subexpression (redex) in which an elementary computation step can be performed, executing that step, and so on[136]. During the search for a redex, the expression is split into two parts (see 18):

- The focus — the subexpression in which we search for a redex;
- The continuation — the remainder of the expression with a "hole," indicating the place where the result of the computation step should be plugged in.

As a rule, continuations exist outside user code as the state of the interpreter, which at each successive moment in time needs to remember what code to execute next and in what state. However, languages provide users with many constructs that allow controlling continuations (see 19):

- The function `exit` discards the entire program continuation;
- The construct `try-catch` allows discarding part of the continuation up to the exception handler and executing the remainder;
- The construct `return` allows resuming execution at the point where the function was called;
- The constructs `break` and `continue` restore the continuation after and before the loop, respectively...

[135] `https://en.wikipedia.org/wiki/Continuation`

[136] In fact, computing with continuations is taught in elementary school, when students learn to evaluate expressions "step by step."

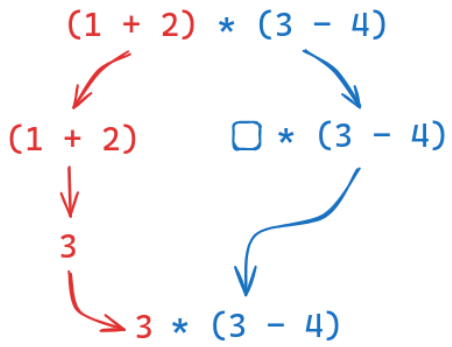


Figure 18: The expression is split into a focus (red) and a continuation (blue); when a step is performed in the focus, they are combined into a new expression, and the process repeats.

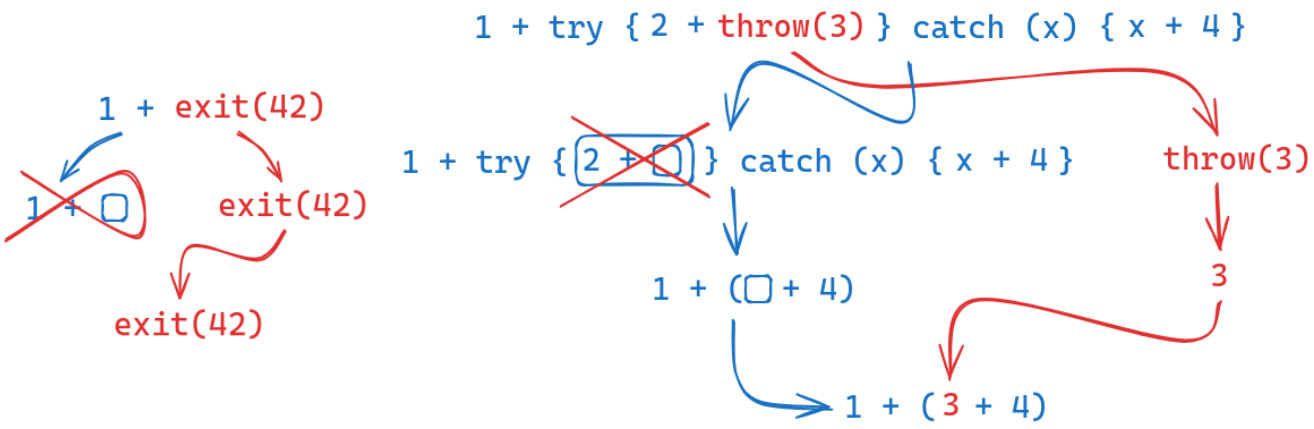


Figure 19: The construct `exit` discards the continuation and the program halts, while `throw` discards only part of the continuation up to the nearest `try-catch`.

6.1.1 Reduction semantics

The style of specifying operational semantics through the description of searching for primitive ("**head**") redexes and their reductions is called **reduction semantics (reduction context semantics)**[137]. As usual, following Hutton's Razor, let us consider the interpretation of a small simple language with subtraction (an asymmetric operation for testing the implementation):

```
data Expr = Const Int | Diff Expr Expr
```

First, we need to define the syntax of continuations, "expressions with a hole." That is, from a technical standpoint, a continuation is a data structure containing all the information necessary to continue execution (here — only the remainder of the expression):

```
data K
  = Hole          -- hole □
  | LDiff K Expr -- focus went into the left subexpression, remembered the right
  | RDiff Int K  -- computed the left, focus went right
```

Let us define the operation of splitting an expression into a focus and a continuation:

```
split :: Expr -> (Expr, K)
split e = case e of
  Const _ -> (e, Hole)
  Diff (Const _) (Const _) -> (e, Hole) -- primitive redex
  Diff (Const l) r ->                   -- left subexpression already computed
    let (focus, k) = split r in         -- search for redex in the right
    (focus, RDiff l k)
  Diff l r ->                           -- nothing computed yet
    let (focus, k) = split l in         -- start by searching for redex on the left
    (focus, LDiff k r)

ghci> split (Diff (Diff (Const 1) (Const 2)) (Const 3)) -- (1 - 2) - 3
(Diff (Const 1) (Const 2), LDiff Hole (Const 3)) -- (1 - 2, □ - 3)
```

The primitive reduction step only knows how to subtract numbers:

```
headReduction :: Expr -> Expr
headReduction = \case
  Diff (Const l) (Const r) -> Const (l - r)
  e -> e
```

After a primitive reduction step, we need to plug the result back into the continuation, so that from an "expression with a hole" we obtain a full expression that can be further reduced:

[137] https://en.wikipedia.org/wiki/Operational_semantics

```haskell
plugIn :: Expr -> K -> Expr
plugIn e k = case k of
  Hole -> e
  LDiff k' r -> Diff (e `plugIn` k') r
  RDiff l k' -> Diff (Const l) (e `plugIn` k')
```

As a rule, plugging a term t into a continuation E is denoted as $E[t]$.

Now we can define a full step:

```haskell
transition :: Expr -> Expr
transition e =
  let (focus, k) = split e in    -- split into focus and context
  headReduction focus `plugIn` k -- perform computation in the focus and plug in
```

Then the operational semantics is the unfolding of the list of intermediate expressions:

```haskell
eval :: Expr -> [Expr]
eval = List.unfoldr \prev ->
  let next = transition prev in
  if prev == next then Nothing else Just (next, next)
```

One can notice that `split` and `plugIn` are uniquely determined by the syntax of continuations and the form of head reductions, while `transition` and `eval` are the same for any language. Therefore, reduction semantics is usually specified as the syntax of continuations, a list of head reductions, and a single inference rule — a step in context:[138]

Values	v	$::= \mathbb{Z}$
Terms	t	$::= v \mid t \ominus t$
Evaluation context	E	$::= \square \mid E \ominus t \mid \mathbb{Z} \ominus E$
(diff)		$v_1 \ominus v_2 \longrightarrow v_1 - v_2$
step		$\dfrac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}$

There exists a standard tool PLT Redex[139] for describing and testing semantics in the reduction context style.

6.1.2 Continuation semantics

Let us write the denotational semantics of our language:

```haskell
evalDirect :: Expr -> Int
evalDirect = \case
  Const n -> n
  Diff l r -> evalDirect l - evalDirect r
```

[138] $\ominus$ here denotes syntactic minus (a node in the tree).

[139] `https://redex.racket-lang.org/`

Let us rewrite the denotational semantics in a style with explicit continuations. But first, let us work with the type `K` and simplify handling it:

```
data K = Hole | LDiff K Expr | RDiff Int K
-- K = 1 + (K × Expr) + (Int × K) = 1 + (Expr + Int) × K
data Frame = LDiff Expr | RDiff Int
type K = [Frame]
```

We have obtained a representation of the continuation as a stack of frames.

We will write the denotational semantics for the domain `K -> Int`. Such a variant of denotational semantics with an explicit representation of the meaning of the rest of the program (in the form of `K`) is sometimes called **continuation semantics**. And the corresponding programming style with passing continuations is called **continuation passing style (CPS)**.

In our case, the continuation semantics will look like a pair of mutually recursive functions:

```
evalK :: Expr -> K -> Int
evalK e k = case e of
  Const n -> k `appK` n              -- execute the rest of the program
  Diff l r -> evalK l (LDiff r : k) -- remember to compute the right next

appK :: K -> Int -> Int
appK k result = case k of
  [] -> result                                  -- nothing more to do
  LDiff r : k' -> evalK r (RDiff result : k') -- go compute the right
  RDiff l' : k' -> k' `appK` (l' - result)    -- continue on the result
```

Here we again descend through the expression searching for a primitive redex (here — a constant), while remembering what needs to be done afterwards.

The first recursive implementation `evalDirect` did not worry about continuations. However, continuations are an integral part of the computation process: the evaluator needs to know what to do next at every moment. In fact, `evalDirect` merely delegates the management of the object language's continuations to the meta-language, but how? Notice that the resulting implementation `evalK` is tail-recursive, meaning it can be compiled into a loop that does not consume the call stack, while `K` is the stack. Thus, in the first case we made recursive calls and the continuation was allocated on the (hardware) stack of the meta-language, while in the second case we manually allocate the stack on the heap.

In the case where the meta-language's stack is implemented on top of the hardware stack, there is a risk of a stack overflow error. To avoid this, the **trampolining** technique is used, which consists precisely in manually allocating the continuation on the heap Ganz et al. [1999], Bjarnarson [2012].

Just as the type of a zipper's context can be computed as the derivative of the algebraic representation of the corresponding type Huet [1997], McBride [2001], Abbott et al. [2003], one can compute the type of a fold's continuation McBride [2008].

6.1.3 First-class continuations

In the examples above, language constructs manage continuations implicitly (see Fig. 19). However, sometimes operators are introduced that allow explicitly manipulating continuations. With their help one can implement both control flow manipulation capabilities such as generators and coroutines[140][141], as well as all other effects such as state (see 7).

First-class continuations are continuations that can be represented as values in the program. Given that a continuation has a vacant slot for a not-yet-computed subexpression, first-class continuations are represented as first-class functions.

To obtain a first-class continuation in code, one needs to either write the code in CPS, or use a built-in language operator, of which a great many have been invented [Hillerström, 2022, appendix A]. For example, *J*, `escape` Reynolds [1972], `call/cc`...

As an example, let us implement in our language an operation `Cont` that allows capturing the current continuation. It will take a user function and pass to it the current continuation from itself to the end of the program:

$$E[cont\ f] \longrightarrow f\ (\lambda x.\, E[x])$$

First, let us extend the language with lambda calculus:

```
data Expr
  = Const Int | Diff Expr Expr
  | Var String | Lam String Expr | App Expr Expr

data Frame
  = LDiff Expr | RDiff Value
  | LApp Expr | RApp Value
  | SetEnv Env -- after executing a closure, restore the current environment
```

The interpreter implementation is fairly straightforward. The only thing is that the remainder of the program is now represented as a linear list that is effectively traversed sequentially in a loop. We must not forget to restore the original environment after executing a closure body for the continuation to proceed:[142]

```
evalK :: (?env :: Env) => Expr -> K -> Value
evalK e k = case e of
  Const n -> k `appK` Number n
  Diff l r -> evalK l (LDiff r : k)
  Var name -> k `appK` (?env ! name)
  Lam name body -> k `appK` Closure name ?env body
  App f arg -> evalK f (LApp arg : k)
```

[140] https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style#Example:_coroutines
[141] https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.coroutines/suspend-coroutine.html
[142] For convenience, we will pass the environment as an implicit parameter.

```haskell
appK :: (?env :: Env) => K -> Value -> Value
appK k result = case k of
  [] -> result
  LDiff r : k' -> evalK r (RDiff result : k')
  RDiff l' : k' -> k' `appK` Number (unwrapNumber l' - unwrapNumber result)
  LApp arg : k' -> evalK arg (RApp result : k')
  RApp f : k' -> case f of
    Closure name env body ->
      let currEnv = ?env in
      let ?env = Map.insert name result env in
      evalK body (SetEnv currEnv : k')
    K env k'' ->
      let currEnv = ?env in
      let ?env = env in
      (k'' ++ SetEnv currEnv : k') `appK` result
    other -> error $ "Expected callable, got " <> show other
  SetEnv env : k' -> let ?env = env in k' `appK` result
  where
    unwrapNumber = \case
      Number n -> n
      other -> error $ "Expected number, got " <> show other
```

Finally, let us implement the `cont` operator:

```haskell
data Expr = ... | Cont Expr
data Frame = ... | ContFrame

evalK e k = case e of
   ...
   Cont f -> evalK f (ContFrame : k)

appK k result = case k of
   ...
   ContFrame : k' -> [RApp result] `appK` K ?env k' -- reuse the RApp branch
```

Let us consider several examples (denoting `contLam name body = Cont (Lam name body)`, `(-.) = Diff`, `c = Const`, `v = Var` and `(@) = App`):

- `c 10 -. contLam "k" (c 1)` — discard the continuation and return 1 ($k = 10 - \square$);
- `c 10 -. contLam "k" (v "k" @ c 1)` — $\rightsquigarrow k\ 1 \rightsquigarrow 10 - 1 \rightsquigarrow 9$;
- `c 10 -. contLam "k" (v "k" @ c 1 -. c 2)` — $\rightsquigarrow k\ 1 - 2 \rightsquigarrow 9 - 2 \rightsquigarrow 7$;
- `c 10 -. contLam "k" (v "k" @ c 1 -. v "k" @ c 2)` — $\rightsquigarrow k\ 1 - k\ 2 \rightsquigarrow 9 - 8 \rightsquigarrow 1$.

Thus, we have obtained a language that allows the user to obtain the current continuation as a function at any point in the program.

The ability to obtain first-class continuations is almost never provided by industrial languages, as it is a rather dangerous tool. Indeed, if the user does not invoke the continuation, resources may not be released. If they invoke it multiple times, there may again be incorrect resource handling or corruption of mutable memory (continuations that can be safely invoked multiple times are called **multi-shot**). Such behaviors can be excluded by special handling of these situations Muhcu et al. [2025], compile-time checks (e.g., using linear types) or runtime checks. Currently, with active research on integrating effect handlers (see 7), first-class continuations may get a new chance.

6.2 Continuations by hand

Let us consider what CPS looks like in ordinary code, rather than in the context of deep embedding. For this, we will first switch to shallow embedding and directly construct elements of the target domain. Second, we will refunctionalize the continuations `K` and generalize: instead of `K -> Int` we will use `forall r . (a -> r) -> r`, where `a -> r` is the functional representation of the continuation.

Such CPS exploits the following isomorphism:

```
to :: a -> (forall r . (a -> r) -> r)
to x k = k x

from :: (forall r . (a -> r) -> r) -> a
from comp = comp id
```

In other words, instead of providing a value of type `a`, one can ask the caller how they intend to use this value `a -> r`, do it oneself, and return `r` to the caller.

Practitioners know this isomorphism through the technique of using callbacks. Theoreticians know that it is a special case of the Yoneda lemma Hinze and James [2010]. Compare CPS with the instance of this lemma in preorder theory (above we took `a = Unit; b = a`):

$$a \to b \cong \forall r.\, (b \to r) \to (a \to r)$$
$$a \le b \iff \forall r.\, (b \le r) \Rightarrow (a \le r)$$

For example, we can rewrite factorial in CPS. Note that the code has access to a first-class continuation (however, it does not yet use it in any non-trivial way), and is again tail-recursive.

```
facCps :: Int -> (forall r . (Int -> r) -> r)
facCps n k
  | n <= 1 = k 1
  | otherwise = facCps (n - 1) \res -> k (n * res)

facCps 3 id ⇝ facCps 2 \res -> id (3 * res)
⇝ facCps 1 \res -> id (3 * (2 * res)) ⇝ id (3 * (2 * 1)) ⇝ 6
```

Exercise 45 *How much stack memory does the function `facCps` consume?*

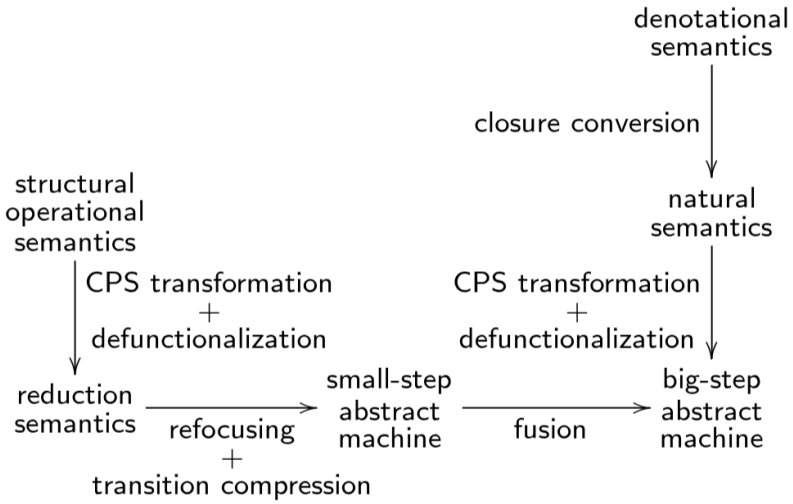


Figure 20: Relationship between semantics in various styles Danvy [2008].

6.2.1 Defunctionalization and accumulators

Let us defunctionalize the continuations in `facCps`. We use two higher-order functions, `id` and `\res -> k (n * res)`, which give us two constructors:

```
data K = Id | Times Int K
-- ≅
type K = [Int]

runK :: K -> Int
runK = product

facCps :: Int -> K -> Int
facCps n k
  | n <= 1 = runK k
  | otherwise = facCps (n - 1) (snoc k n)

facCps 3 [] ⇝ facCps 2 [3] ⇝ facCps 1 [3, 2] ⇝ runK [3, 2] ⇝ 3 * (2 * 1)
```

We have again obtained the representation of the continuation as a stack of frames. This means that the interpreter `evalK` we obtained earlier (6.1.2) is a CPS version of a regular interpreter, but with defunctionalized continuations (which are convenient for debugging — they can be printed, unlike Haskell functions). If we extend our earlier language and write factorial in it, then perform fusion or refunctionalization (4.4.1) of the program tree and the continuation tree, we obtain the current factorial implementation. And in general, correspondences can be established between various styles of semantics implementations (see Fig. 20).

This technique can be used in more complex cases as well, to easily obtain tail-recursive

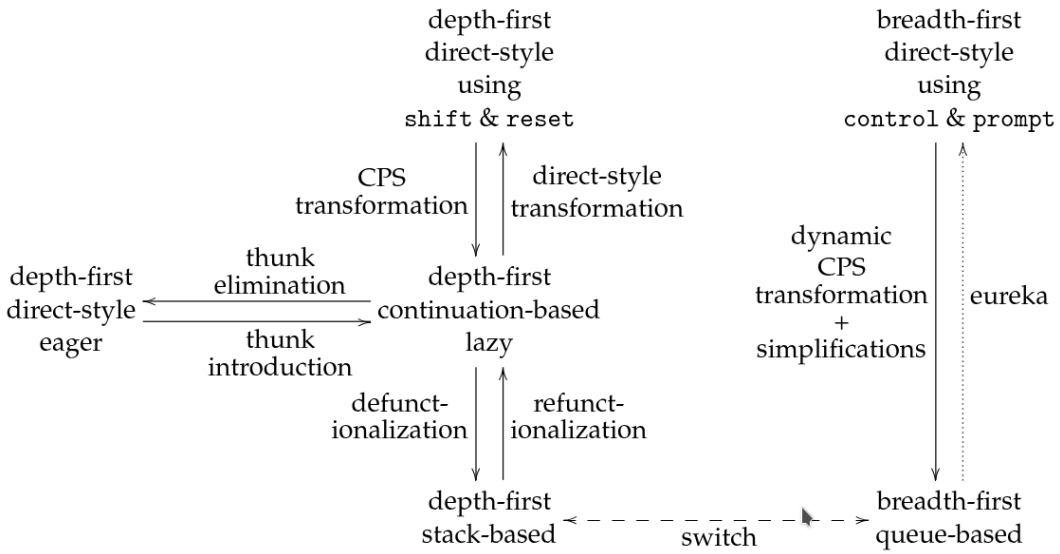


Figure 21: Code transformations based on CPS and de(re)functionalization Danvy [2006].

(iterative) implementations[143] Gibbons [2021][144]. In general, CPS and defunctionalization are a rich source of various refactorings (see Fig. 21).

Now let us notice that the multiplication operation is associative, so we can replace `snoc` with multiplication and represent the continuation as a single number. We obtain the familiar factorial implementation with an accumulator:

```
facAcc :: Int -> Int -> Int
facAcc n acc
  | n <= 1 = acc
  | otherwise = facAcc (n - 1) (acc * n)

facAcc 3 1 ⇝ facAcc 2 (1 * 3) ⇝ facAcc 1 ((1 * 3) * 2) ⇝ (1 * 3) * 2 ⇝ 6
```

Also, one can represent frames as endomorphisms and use function composition as the associative operation Ploeg and Kiselyov [2014][145].

6.2.2 Monad Cont

Due to CPS, the code has lost its familiar structure where functions directly return their results (i.e. **direct style**). With a large number of calls to transformed functions, the code becomes hard to read (a problem known as **callback hell**):

```
fibCps :: Int -> (forall r . (Int -> r) -> r)
```

[143] https://www.pathsensitive.com/2019/07/the-best-refactoring-youve-never-heard.html
[144] (youtube) Jeremy Gibbons - Continuation-passing style, defunctionalization, and associativity.
[145] https://wiki.haskell.org/Difference_list

```
fibCps n k = if n <= 2 then k 1 else
  fibCps (n - 1) \res1 ->
  fibCps (n - 2) \res2 ->
  k (res1 + res2)
```

The domain `(a -> r) -> r` can be made into a monad, restoring direct-style code inside `do`-notation. Let us introduce a `newtype` wrapper for declaring instances:

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

The functor adds post-processing to the result before passing it to the continuation:

```
instance Functor (Cont r) where
  -- fmap :: (a -> b) -> ((a -> r) -> r) -> ((b -> r) -> r)
  fmap f (Cont comp) = Cont \k -> comp (k . f)
```

The applicative simply passes the value to the continuation:

```
instance Applicative (Cont r) where
  pure x = Cont \k -> k x
  (<*>) = ap
```

One can notice that monadic bind also takes a continuation as its second argument, but a "small" one, up to the end of the `do`-block. Thus, the meaning of the monadic bind implementation for `Cont` is composing the "small" continuation with the "large" continuation passed from outside:

```
instance Monad (Cont r) where
  (>>=) :: Cont r a -> (a -> Cont r b) -> Cont r b
  Cont comp >>= k = Cont \k' -> comp \x -> runCont (k x) k'
```

Now we can write linear code, and the monadic machinery automatically constructs continuations and feeds them into previous computations:

```
fibCont :: Int -> Cont r Int
fibCont n = if n <= 2 then pure 1 else do
  res1 <- fibCont (n - 1)
  res2 <- fibCont (n - 2)
  pure (res1 + res2)
```

Exercise 46 *Abort the computation if `res1` is greater than 50.*

Exercise 47 *Abort the computation as soon as the total result exceeds 50.*

The monad `Cont` provides an implementation of an embedded language in which one can obtain the continuation of a computation:

```
Cont :: ((a -> r) -> r) -> Cont r a
```

Practitioners know this technique of writing CPS code via monads through the concepts of `Future/Promise`[146][147].

[146] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[147] https://github.com/promises-aplus/promises-spec/issues/94

6.2.3 `call/cc`

The most well-known classical operator used in Scheme for obtaining first-class continuations is `call/cc` **(call with current continuation)**[148][149]. The provided continuation is **undelimited (abortive)**, since it contains "the end of the program" — no code will execute after its invocation. Undelimited continuations are, de facto, not quite functions, since they do not return a result (it is already "beyond the grave"); consequently, they also do not compose (just as it would be strange to compose `abort` with `exit`)[150][151]. They are more like co-values: while a part of the program is executing, they await its result Curien and Herbelin [2000].

Let us emulate `call/cc` in the `Cont` monad. We cannot capture the entire program continuation, since it is only assembled within `Cont`, but we can ignore the continuation of the call to the captured continuation:

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = Cont \k -> runCont (f \x -> Cont \_ -> k x) k

foo :: Int -> Cont r String
foo x = callCC \k -> do
  let y = x ^ 2 + 3
  when (y > 20) $ k "over twenty" -- throws next line away
  pure (show $ y - 4)
```

6.3 Delimited continuations

In modern practice, continuations are typically used not up to the end of the program, but only up to a certain point. They are called **delimited** or **composable continuations**, **subcontinuations**[152][153].

Constructs for working with such continuations come in pairs: an operator is introduced that delimits the current continuation (possibly with a label); as well as an operator for capturing a fragment of the current continuation (possibly up to a specific delimiter with a given label). Many such operators have been invented [Hillerström, 2022, appendix A], but they are more or less all reducible to each other.

For example, working with exceptions involves two constructs: the continuation is delimited by `try-catch`, and `throw` discards the corresponding partial continuation without capturing it (see Fig. 19):

$$E_1[try\{E_2[throw(v)]\}catch(x)\{t\}] \to E_1[[x \mapsto v]\, t]$$

We will consider the universal operators from Dyvbig et al. [2007]. The paper introduces the

[148] https://en.wikipedia.org/wiki/Call-with-current-continuation
[149] https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style#callCC
[150] https://okmij.org/ftp/continuations/undelimited.html
[151] https://okmij.org/ftp/continuations/against-callcc.html
[152] https://www.cl.cam.ac.uk/teaching/2324/R277/handout-delimited-continuations.pdf
[153] (youtube) Keynote: Delimited Continuations, Demystified by Alexis King | Lambda Days 2023.

$$
\begin{array}{lrcl}
\text{(Variables)} & x, y, \ldots & & \\
\text{(Expressions)} & e & ::= & x \mid \lambda x.e \mid e\ e \\
& & \mid & \mathit{newPrompt} \mid \mathit{pushPrompt}\ e\ e \\
& & \mid & \mathit{withSubCont}\ e\ e \mid \mathit{pushSubCont}\ e\ e
\end{array}
$$

Figure 22: Syntax of λ-calculus with primitives for working with continuations.

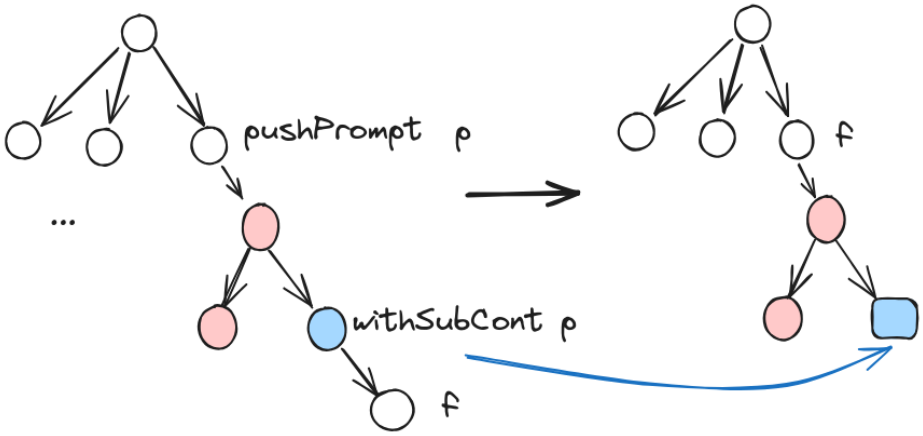


Figure 23: Example of `withSubCont` in action.

following set of syntactic constructs (Fig. 22) for working with delimited continuations, in addition to pure call-by-value lambda calculus:

- *newPrompt*[154] — creates a fresh identifier (label) for a delimiter;
- *pushPrompt p e* — installs a delimiter with label *p* and executes expression *e*;
- *withSubCont p f* — captures the partial continuation up to the delimiter with label *p* and passes it to function *f*, returns the result of *f* (Fig. 23);
- *pushSubCont k v* — executes the composition of the current continuation and *k* on value *v*.

Delimited continuation operators can be easily understood as **resumable exceptions**, i.e. exceptions that can be caught and the program resumed from the point where the exception was thrown (with some value). What is unusual is that the classical delimited continuation operators accept code that works with the partial continuation at the "throw site," rather than at the "catch site." That is, the handling block is written not with `catch`, but with `throw`:

$$
E_1[\mathit{pushPrompt}\ p\ \{E_2[\mathit{withSubCont}\ p\ f]\}] \to E_1[f\ E_2]
$$
$$
E_1[\mathit{try}\{E_2[\mathit{throw}\ v]\}\mathit{catch}(x,k)\{t\}] \to E_1[[x \mapsto v]\,[k \mapsto \lambda y.\, E_2[y]]\,t]
$$

[154] Historically in Lisps, undelimited continuations were delimited only by the REPL, hence the name for delimiters — "prompt."

$$
\begin{array}{l}
(\lambda p.\ 2 + \mathit{pushPrompt}\ p \\
\qquad \text{if } (\mathit{withSubCont}\ p \\
\qquad\quad (\lambda k.\ (\mathit{pushSubCont}\ k\ \text{False})\ +\ (\mathit{pushSubCont}\ k\ \text{True}))) \\
\qquad \text{then } 3 \\
\qquad \text{else } 4) \\
\ \mathit{newPrompt}
\end{array}
$$

Figure 24: Example expression (result is 9).

Exercise 48 *Reduce the example in Fig. 24.*

6.3.1 Implementing the operators

Let us extend the language from 6.1.3 with the operators for working with delimited continuations discussed above. For simplicity, instead of fresh prompt labels we will use names, and instead of `pushSubCont` ordinary application:

```
data Expr = ...| PushPrompt String Expr | WithSubCont String Expr
data Frame = ... | PushPromptFrame String | WithSubContFrame String
```

The redex search is extended in the obvious way:

```
evalK :: (?env :: Env) => Expr -> K -> Value
evalK e k = case e of
  -- ...
  PushPrompt promptName body -> evalK body (PushPromptFrame promptName : k)
  WithSubCont promptName f -> evalK f (WithSubContFrame promptName : k)
```

The entire job of `PushPrompt` is to get stuck in the continuation as `PushPromptFrame`, but when execution reaches it, it is simply ignored:

```
appK :: (?env :: Env) => K -> Value -> Value
appK k result = case k of
  -- ...
  PushPromptFrame _ : k' -> k' `appK` result
```

To capture a fragment of the continuation, we need an operation that finds the first prompt with the matching name in the continuation and returns the **subcontinuation** — up to that prompt, and the **metacontinuation** — after it:

```
splitByPrompt :: K -> String -> (K, K)
splitByPrompt k targetPromptName = go [] k
  where
    go _ [] =  error $ "Prompt " ++ targetPromptName ++ " not found"
```

```
    go subcont (PushPromptFrame promptName : metacont)
      | promptName == targetPromptName = (subcont, metacont)
    go subcont (frame : metacont) = go (subcont ++ [frame]) metacont
```

Now `WithSubCont` must simply apply the user function to the subcontinuation, while not forgetting to execute the metacontinuation afterwards:

```
appK :: (?env :: Env) => K -> Value -> Value
appK k result = case k of
  -- ...
  WithSubContFrame promptName : k' ->
    let (subcont, metacont) = k' `splitByPrompt` promptName in
    (RApp result : metacont) `appK` K ?env subcont
```

Exercise 49 *Reduce the following example:*

```
exampleDelimited = let ?env = Map.empty in flip evalK [] $
  c 10 -. PushPrompt "p"
    (c 5 -. WithSubCont "p"
      (lam "k" $ v "k" @ c 1 -. v "k" @ c 3))
```

Many classical operators, such as `shift/reset`, `prompt/control`, etc., can be obtained by leaving the prompt frame in the subcontinuation or in the metacontinuation in the implementation of `splitByPrompt` Dyvbig et al. [2007].

6.3.2 In `Monad Cont`

As an example, let us implement two classical operators for working with delimited continuations — **shift-reset**. We will use the continuations assembled by the `Cont` monad. `shift` simply captures the current continuation and passes it to the user computation similarly to `cont`:

```
shift :: ((a -> r) -> Cont r r) -> Cont r a
shift f = Cont \k -> runCont (f k) id
```

`reset` passes `id` as the continuation to the argument computation, so that computation does not have access to the continuation after `reset` (while the Haskell continuation remembers that after `comp` it needs to execute `k`):

```
reset :: Cont a a -> Cont r a
reset comp = Cont \k -> k (runCont comp id)
```

Exercise 50 *What will be the result of executing the following function:*

```
exampleShiftReset = flip runCont id $
  (1 +) <$> reset ((2 +) <$> shift \k -> pure (k 3 + k 5))
```

6.4 Applications of continuations

Continuations are useful for understanding the meaning of programs we write every day (see 107). Furthermore, as we will see in this chapter, first-class continuations can be used as a means of building powerful embedded languages.

6.4.1 Everything through continuations

Let us implement a number of useful embedded languages using first-class continuations. To gain access to continuations, we will use the shallow embedded language `Cont`[155]. Thus, we will work with the tower of languages Haskell, `Cont`, X, where X is the embedded language under consideration.

Let us start with a trivial embedded imperative language:

```
runIdentityC :: (forall r . Cont r a) -> a
runIdentityC comp = runCont comp id

exampleIdentity :: Int
exampleIdentity = runIdentityC do
  x <- pure 4
  y <- pure 5
  pure (x + y)
```

Let us implement a language with exceptions. For this we modify the first type parameter of `Cont`, the **answer (response) type**[156]. The function `abort` ignores the program continuation up to `runExn`, which concludes the continuation by wrapping the successful result in `Just`:

```
abort :: Cont (Maybe r) a
abort = Cont $ const Nothing

runExn :: (forall r . Cont (Maybe r) a) -> Maybe a
runExn comp = runCont comp Just

exampleExn :: Int -> Maybe Int
exampleExn n = runExn do
  when (n < 0) abort
  pure (n + 1)
```

Notice that the semantics of the abort operation is entirely concentrated in the function `abort`, whereas in `Monad Maybe` the corresponding function would only be responsible for constructing a domain element, while monadic bind would handle the continuations. Having access to first-class

[155] https://blog.poisson.chat/posts/2019-10-26-reasonable-continuations.html
[156] https://wiki.haskell.org/Cont_computations_as_question-answering_boxes

continuations, we no longer need to implement monadic bind manually. It is said that for arbitrary `m`, `Cont (m r) a` is a "free" implementation of a monad[157].

Error handling through returning a special result (`Either` in Haskell, `Result` in Rust, `nil` in Go...) has monadic semantics (conditional branches at each point discard only part of the continuation), while a full-fledged exception mechanism has access to the entire delimited continuation up to the corresponding `try-catch` block, which resembles our current implementation. From the perspective of syntactic noise at the source code level, the exception variant is clearly preferable, and we will consider the question of typing exceptions later (see 8). From a performance perspective, things are less obvious[158][159], though stack unwinding by the runtime seemingly has more room for optimization.

Now let us try invoking the continuation multiple times. We obtain a language with nondeterminism (or backtracking). Here, `choice` continues the rest of the program twice and accumulates all obtained results in a list:

```haskell
choice :: Cont [r] Bool
choice = Cont \k -> k True ++ k False

runNondet :: (forall r . Cont [r] a) -> [a]
runNondet comp = runCont comp (:[])

exampleNondet :: Int -> [Int]
exampleNondet n = runNondet do
  b <- choice
  if b then pure n else pure (n - 1)
```

Remarkably, having only first-class continuations one can even implement a language with a mutable memory cell:

```haskell
-- s -> (s, a) ~ ((s, a) -> r) -> (s -> r)
-- ~ (a -> s -> r) -> (s -> r) ~ Cont (s -> r) a

get :: Cont (s -> r) s -- (s -> s -> r) -> (s -> r)
get = Cont \k s -> k s s

put :: s -> Cont (s -> r) ()
put s' = Cont \k _s -> k () s'

runStateC :: (forall r . Cont (s -> r) a) -> s -> (a, s)
runStateC comp = runCont comp (,)
```

[157] https://hackage.haskell.org/package/kan-extensions-5.2/docs/Control-Monad-Codensity.html
[158] https://www.serpentine.com/2011/02/25/cps-is-great-cps-is-terrible/
[159] https://stackoverflow.com/questions/13835817/are-exceptions-in-c-really-slow

```haskell
exampleState :: Int -> ((), Int)
exampleState = runStateC do
  s <- get
  put (s + 1)
```

The idea is quite simple: capturing the continuation leaves behind a function into which the previous continuation call supplies the current state. And indeed, mutable state is simply a fragment of the continuation (e.g., the hardware stack) to which we have direct read and write access. Here we have placed an application at the beginning of the continuation, to which, by capturing the continuation, we jump to obtain the argument and update it to a new one.

Exercise 51 *Reduce the example above to understand how it works.*

Intuitively, the power of continuations can be explained as follows: maintaining the continuation is the primary responsibility of the language's runtime system; when continuations are handed to the user, their code becomes involved in the runtime's activity and effectively becomes part of it.

6.4.2 The mother of all monads

In the previous section (6.4.1) we implemented the capabilities of classical monads in a language with continuations. Does this work for an arbitrary monad? It turns out that yes[160][161] Filinski [1994].

Recall that a monad is a type constructor `m` with which we represent the denotation of a computation. The operations `pure` and `bind` allow creating and composing denotations. Thus, we have computations represented as first-class data.

If our language provides first-class continuations (let us take `Cont (m r)` as such a language for example), we can define two operations: `reflect` — execute a monadic computation in the language with continuations and obtain the result `a`; `reify` — obtain a denotation from a computation in the language with continuations[162]. In Haskell, this can be expressed as overloading this pair of functions for various monads:

```haskell
class MonadicReflection m where
  -- m a -> (a -> m r) -> m r
  reflect :: m a -> forall r . Cont (m r) a
  -- (forall r . (a -> m r) -> m r) -> m a
  reify :: (forall r . Cont (m r) a) -> m a
```

For example, for the `State` domain the implementation looks as follows:

```haskell
newtype State s a = State { runState :: s -> (a, s) }
```

[160] `http://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/the-mother-of-all-monads`

[161] `https://blog.poisson.chat/posts/2019-10-26-reasonable-continuations.html`

[162] For a strict language, `reify` should accept a thunk `(() -> a) -> m a`.

```haskell
instance MonadicReflection (State s) where
  reflect :: State s a -> (forall r . Cont (State s r) a)
  reflect comp = Cont \k -> State \s ->
    let (a, s') = runState comp s in
    runState (k a) s'

  reify :: (forall r . Cont (State s r) a) -> State s a
  reify comp = State $ runState (runCont comp (\x -> State (x,)))

exampleStateReflection :: Int -> (Int, Int)
exampleStateReflection = runState $ reify do
  x <- reflect get
  reflect $ put (x + 1)
  pure x
```

Intuitively, this can be understood as follows: the bind implementation does not actually care whether it is given a "small" continuation (assembled via desugaring of `do`-notation) or a "large" continuation assembled by the meta-language or the `Cont` monad. And in principle, monads can be defined through their embedding into the `Cont` monad[163].

Thus, if the language supports first-class continuations, one can implement the functionality of an arbitrary monad in direct style[164], without `do`-notation, applicative chains, and other syntactic noise[165][166]. Later we will see that this approach, unlike monads, natively supports composition (7), and also does not impose restrictions on typing (8). This raises the question: do we still need monads in programming?

6.4.3 Generators and coroutines

Generators and coroutines are usually defined as computations that can be suspended and resumed in the same state Moura and Ierusalimschy [2009]. Coroutines generalize generators and are typically used as a primitive for asynchronous programming in the form of `async/await` or other language constructs Elizarov et al. [2021]. The difference between synchronous and asynchronous programming can be understood as follows: in the former case, continuations are managed exclusively by the operating system, while in the latter — by the language facilities.

As an example, let us implement generators. As the result of a generator we will use a lazy list, an iterator encoded as an unfold (see 4.5):

```haskell
data Box f = forall s . Box s (s -> f s)
data ListF a rec = Nil | Cons a rec
type Iterator a = Box (ListF a)
```

[163] https://blog.poisson.chat/posts/2019-10-27-continuation-submonads.html
[164] https://www.unison-lang.org/docs/fundamentals/abilities/for-monadically-inclined/
[165] http://www.valuedlessons.com/2008/01/monads-in-python-with-nice-syntax.html
[166] https://github.com/lampepfl/monadic-reflection/blob/main/TUTORIAL.md

```haskell
box2list :: Iterator a -> [a]
box2list (Box s next) = case next s of
  Nil -> []
  Cons x s' -> x : box2list (Box s' next)
```

The hidden state of the iterator will be the not-yet-executed continuation. We will obtain access to the continuation via the `Cont` monad, where we take `GenState` as the response type. Thus, the `yield` operation suspends the computation and saves the continuation in the `Yield` constructor, while `makeGen` finishes the given generator by producing the `Stop` constructor:

```haskell
data GenState a = Stop | Yield a (() -> GenState a)

makeGen :: Cont (GenState a) () -> Iterator a
makeGen comp = Box
  (\() -> runCont comp (\() -> Stop))
  (\k -> case k () of
      Stop -> Nil
      Yield x k' -> Cons x k'
  )

yield :: a -> Cont (GenState a) ()
yield x = cont \k -> Yield x k
```

Now we can write code that sequentially produces multiple results:

```haskell
exampleGen :: [Int]
exampleGen = box2list $ makeGen do
  yield 1
  yield 2
  yield 3
```

Exercise 52 *Reduce `runCont exampleGen (\() -> Stop)` to normal form.*

6.5 Efficient work with continuations

In practice, implementing generators and coroutines requires suspending and resuming programs, in other words, capturing continuations. This needs to be done as efficiently as possible.

6.5.1 Contiguous stack

The continuation is represented as a hardware stack. As soon as a continuation needs to be captured, the stack is copied to the heap (possibly lazily)[167]. That is, this approach relies entirely on support from the language runtime.

[167] (youtube) Ivan Uglyansky - Java Project Loom.

6.5.2 Segmented stack

The call stack is represented as a linked list of heap-allocated segments, each corresponding to a delimiting operation. Thus, no copying is required; it suffices to swap pointers. However, in this case the stack is non-local, which is not great for cache performance.

6.5.3 Finite state machine (FSM)

This implementation involves automatic CPS transformation of user code by the compiler. To avoid allocating a large number of closures, continuations are defunctionalized and within each function are represented by a single mutable object. Thus, the function's entire state is allocated on the heap once, and resuming the function body at a particular point is implemented as a state machine — using labels and jumps[168].

This is how, for example, coroutines are implemented in Kotlin[168], generators in C#[169]...

Even in this form, CPS remains a heavyweight transformation that can slow down code execution by orders of magnitude. The issue is, in particular, that variables in this approach are difficult to place in registers (functions have many entry and exit points[168]), and they constantly have to be written to RAM — performing **spilling**[170].

[168] https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#state-machines
[169] https://csharpindepth.com/Articles/IteratorBlockImplementation
[170] https://en.wikipedia.org/wiki/Register_allocation

7 Effects and Modular Interpreters

Earlier we recognized managing complexity as the programmer's main task, and constructing embedded languages as the primary tool for solving it (5.1). In this chapter we consider the notion of an effect. It is closely related to embedded languages and will give us a better understanding of when to construct them, what this gives us, and what one needs to be careful about.

For implementing embedded languages we preferred shallow embedding in the form of tagless final (5.4), which maximally reuses the capabilities of the meta-language and allows giving different interpretations to the same program. Next we studied the computation process and extracted the notion of a continuation (6). It turned out that the tagless final languages we were building around monads can be expressed through continuations more conveniently and simply (6.4.1, 6.4.2). In this chapter we will understand how this will help us solve the expression problem (5.5) completely.

From a historical-philosophical perspective, the theory of effects is a solid bridge between the functional and imperative programming paradigms. The two originated separately — as the Turing machine and Church's λ-calculus — and remained largely isolated schools of thought for decades. Only in the last 30 years has this begun to change, and we have come to understand how the two worlds complement each other. The theory of effects remains an extremely hot topic to this day[171][172].

7.1 The notion of an effect

Let us begin the discussion from the opposite end, with the property of purity. A **pure function** possesses the following properties:

- Its result is always the same given the same set of arguments (it does not depend on anything else in any more non-trivial way);
- Its only observable result is its return value.

In general, the programming style using pure functions is encouraged, as it possesses many good properties. One can conveniently reason about them using equational reasoning; everything needed to understand the code is explicitly written in that code; classical type systems work well, providing abstraction completeness, quality documentation, and partial specification... It is also known that everything can be expressed using pure computations, even working with IO Jones [2001].

However, using only pure functions, everything must be done manually. In the case of IO (and similarly with state) — passing the resulting world as an argument again and again:

```
1 getList :: Int -> World -> (World, [Int])
2 getList n w | n == 0 = (w, [])
3             | otherwise =
4   let (w', x) = getInt w in
5   let (w'', xs) = getList (n - 1) w' in
```

[171] (youtube) The Evolution of Effects — Nickolas Wu.

[172] https://github.com/yallop/effects-bibliography

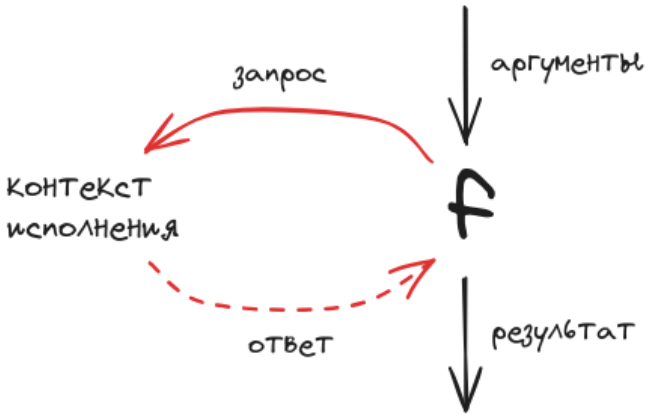


Figure 25: Effects as client-server interaction.

```
6     (w'', x : xs)
```

Thus, code made of pure functions is filled with inessential ceremony, behind which the business logic and essence are hidden. To focus on the important details, one needs to delegate all this bookkeeping to external code, sweep the uninteresting details under the rug. Then the code above can be rewritten, for example, as follows:

```
1  getList :: Int -> IO [Int]
2  getList n | n == 0 = pure []
3            | otherwise = do
4    x <- getInt
5    xs <- getList (n - 1)
6    return (x : xs)
```

The abstract entity to which we will delegate details inessential to the given fragment of business logic we will call the **execution context**[173]. And an **effect** is the interaction of a function with the execution context, which occurs through calls to **effect operations** (for example, `getInt` from the example above)[174]. Thus, one can view the execution context as a server to which the client function sends requests and receives responses (see Fig. 25). In this model, such a function can violate both properties of pure functions.

In practice, this context is the interpreter of an (embedded) language, and the effect operations are its constructs. If the language is embedded, one speaks of **user-defined effects**. Thus we again return to the task of building modular interpreters (5.5). Moreover, the implementation of a computational context can also delegate the implementation of certain functionality to another execution context, and so on. We obtain the already familiar tower of interpreters (5.1.1).

Let us consider some examples of computational contexts and operations:

[173] `https://okmij.org/ftp/Computation/having-effect.html`

[174] Sometimes the term "effect" also refers to the transfer function in dataflow analysis Møller and Schwartzbach [2012]. In essence, it describes the change of context.

- Context — memory management subsystem, `modify` — effect operation: the context maintains the state of memory cells for us;
- Context — exception handler, `throw MyException` — effect operation: the context determines how the error will be handled for us (note that here control does not return to the term);
- Context — dependency injection settings, requesting functionality — effect operation: the context determines the implementation of the functionality we use[175]. As we will see later, this and similar simple effects can be implemented simply and efficiently (see 7.2.2).

Exercise 53 *Give more examples of computational contexts and operations.*

When we talk about a standalone language in which we program (for example, Haskell), any action of ours in the program is executed by it. That is, for instance, is addition an effect operation? In that case it is reasonable to identify a subset of language constructs that are "interesting" enough to be considered as producing an effect.

Which language constructs should be considered "interesting"? Notice that on the one hand it is good that effects hide some complexity from us, allowing us to focus on other things. On the other hand, this is also bad, because we cease to observe this complexity, yet it permeates our code and maintains implicit dependencies between its parts. Thus, effects require additional care from the programmer[176]. Accordingly, it is precisely such non-trivial constructs that should be considered "interesting." At the very least, constructs whose use takes a function out of the pure category should definitely be considered "interesting." Also, these can be operations that make complex non-local modifications to the control flow (maintained by the interpreter in the form of a continuation).

Ultimately, the choice of "interesting" constructs depends on the task and the developer's perspective[173]. Thus, constructs affecting arbitrary observable properties of code, such as termination or computational complexity, can motivate considering recursive calls or long-running operations as effects.

Later we will learn to track and control the use of effects at the type level using effect systems (see later 8).

7.2 Effect handlers

Effect handlers are a modern universal method for building modular interpreters of embedded languages, directly implementing the client-server metaphor. As is often the case, handlers were invented multiple times. In this chapter we will look at the main implementations that will best help us understand the concept.

The main idea of effect handlers is fairly simple. A language construct `handle` is introduced that allows defining a computational context for a given scope, providing implementations of effect operations. A construct `perform` is also introduced that allows invoking an effect operation (sending a request to the context). Each operation has a set of parameters, as well as a "return address,"

[175] There exists the term **contextual polymorphism** — code in different contexts can have different behavior.

[176] (youtube) Kris Jenkins — Side-Effects Are The Complexity Iceberg.

the continuation of the call site, to which it will return the result. For example, in the experimental language Koka[177] a context providing some constant can be implemented as follows (`resume` is the name of the call-site continuation, `perform` is inserted implicitly):

```
with handler
  ctl ask() resume(21)
ask() + ask()
```

If the nearest context cannot handle the required request, the request is delegated to an outer context, and so on until a suitable context is found. The modularity of interpreters defined by handlers is based on this idea.

7.2.1 Handlers via delimited continuations

As we have already seen, various effects can be implemented using access to the current continuation (6.4.1, 6.4.2). Effect handlers extend this idea by using delimited continuations to transfer control to various interpreters (handlers).

It is known that the classical operators for manipulating delimited continuations, monadic reflection, and effect handlers are expressible in terms of each other Forster et al. [2017].

7.2.2 Efficient implementation of handlers

In general, the speed of `perform` is determined by the speed of capturing and restoring delimited continuations. However, there exists a class of operations that can be implemented much more efficiently.

If we look at the implementation of the `ask` operation, we see that its last action is calling the continuation, returning control to the calling code. Such operations are called **tail-resumptive**; they strongly resemble ordinary functions, except that their implementations are determined by the context (handler). Thus, tail-resumptive operations can be implemented as implicit passing of a dictionary of functions from the handler to `perform`, thereby avoiding expensive continuation manipulations Xie et al. [2020][178].

7.2.3 Embedded handlers as explicit client-server communication

To better understand the semantics of handlers, let us implement a language with handlers embedded in Haskell. We start with the variant proposed by Oleg Kiselyov, which most straightforwardly encodes the idea of client-server communication between the term and the context Kiselyov et al. [2013].

Let us begin with the `ask` effect, which requests numbers from the context. We define the data type of messages to the context: it is either the final result of the computation, or a request `Ask` containing a "return address" — the current continuation:

[177] `https://koka-lang.github.io/koka/doc/index.html`

[178] Handlers of tail-resumptive operations resemble co-pattern matching and, correspondingly, objects (see 4.6.2).

```haskell
data Message res = Val res | Ask (Int -> Message res)
```

We will assemble continuations into a specialized `Monad Cont` with a suitable response type (see 6.2.2, 6.4.1):

```haskell
newtype Eff res = Eff
  { runEff :: forall res' . (res -> Message res') -> Message res' }
```

Then the `ask` effect is implemented simply as "sending" an `Ask` request with the current continuation:[179]

```haskell
ask :: Eff Int
ask = Eff \k -> Ask k
```

We implement the handler as a "server" that processes requests in a loop until the computation sends the final result:

```haskell
run :: Eff res -> Message res
run comp = runEff comp Val

runReader :: Eff res -> Int -> res
runReader comp env = loop (run comp)
  where
    loop = \case
      Val res -> res
      Ask k -> loop (k env)
```

Finally, we can write effectful code:

```haskell
exampleReader :: Int -> Int
exampleReader = runReader do
  x <- ask
  y <- ask
  pure (x + y)
```

The `Eff` monad will turn it into a lazy list of messages:

```haskell
exampleReader :: Int -> Int
exampleReader = loop $
  Ask \x ->
  Ask \y ->
  Val (x + y)
```

[179] Earlier we similarly implemented generators, see 6.4.3.

7.2.4 Extensible messages and forwarding

Let us abstract the message type over the "shape" of requests that can participate in them (see 4.3.2):

```
data Message effs res = Val res | Request (effs (Message effs res))
```

The previous message type is obtained by passing the following shape functor:

```
newtype Reader env msg = Ask (env -> msg)
```

Exercise 54 *Verify that* `Message (Reader Int) res` *is equivalent to the previous message type.*

The coproduct of shape functors is a shape functor (see 5.5.1):

```
data (eff |> effs) a = L (eff a) | R (effs a)
```

Now the `ask` operation admits the existence of other request types `effs`:

```
ask :: Eff (Reader env |> effs) env
ask = Eff \k -> Request $ L $ Ask k
```

The new handler processes only a portion of the requests, forwarding the rest to the handler above (composing the continuations in the correct way):

```
runReader
  :: forall effs env res . Functor effs
  => Eff (Reader env |> effs) res
  -> env -> Eff effs res
runReader comp env = loop (run comp)
  where
    loop :: Message (Reader env |> effs) res -> Eff effs res
    loop = \case
      Val res -> pure res
      Request (L (Ask k)) -> loop (k env)
      Request (R unknownReq) -> do
        response <- Eff \k -> Request (fmap k unknownReq)
        loop response
```

Notice that uninterpreted effects remain in the resulting domain. The final domain is obtained by applying all the necessary interpreters. This is how the stable denotations problem (5.5) is solved.

7.2.5 Free monads

To obtain a full-fledged solution in a simpler and more minimalistic way, let us turn to the most classical implementation of handlers via free monads. First, let us discuss free monads themselves.

Consider some algebraic structure, for example, a monoid (a neutral element and an associative binary operation). From an arbitrary set X one can construct a certain monoid $M(X)$ in the most "economical" way — the **free monoid**. This is done as follows: to the set X one adds expression trees with the monoid operations:

```
data M x = Element x | Mempty | Mappend (M x) (M x)
instance Monoid (M x) where
  mempty = Mempty
  mappend l r = Mappend l r
```

One only needs to organize the new set so that there are no trees that are identical from the algebra's perspective (for example, `Mappend Mempty Mempty` and `Mempty`). In the case of a monoid, one can choose the list type:

```
type M x = [x]
instance Monoid (M x) where
  mempty = []
  mappend = (++)
```

Free monads are constructed analogously to free monoids. We use the definition of a monad as a functor with operations `pure` and `join`. Then from an arbitrary functor F one can obtain a monad $Free(F)$ (we show this using the equivalent monad definition from Haskell):

```
data Free f a = Pure a | Join (f (Free f a))
instance Functor f => Monad (Free f) where
  return = Pure
  Pure x >>= k = k x
  Join f >>= k = Join (fmap (>>= k) f)
```

Let us rename the constructors:

```
data Term sig var = Var var | Op (sig (Term sig var))
(>>=) :: Term sig var -> (var -> Term sig var') -> Term sig var'
data MonoidSig subtree = Mempty | MAppend subtree subtree -- (Bool -> subtree)
```

This is nothing other than an encoding of algebraic terms over the signature `sig` with variables from the set `var`. And monadic bind is substitution.

7.2.6 Handlers via free monads

Notice that the following types are isomorphic: `Message` ≅ `Free` ≅ `Term`. Handlers in the classical form arose precisely in the process of studying how to describe effects as algebraic structures Bauer [2018][180].

Instead of variables we store the pure result of the computation, and signatures are written in the form $P \times (A \to K)$, where P is the operation parameter and A is the operation result, by which the handler selects the needed continuation subtree for resuming the computation:

```
data Comp effs res = Res res | Op (effs (Comp effs res))
data Reader env comp = Ask () (env -> comp)
data State s comp = Get () (s -> comp) | Put s (() -> comp)
```

The `Monad Comp` implementation will compose continuations for us in `do`-notation:

```
ask = Op (Ask () (\e -> Res e))
example = do
  x <- ask
  y <- ask
  pure (x + y)
-- $\alpha\beta$-equivalent to
example =
  Op (Ask () (\x ->
  Op (Ask () (\y ->
  Res (x + y)))))
```

The handler also simply folds the list of operations. Operations it cannot handle are left in the tree. To skip an unknown operation and interpret the subtree (continuation), `fmap` is used:

```
runReader
  :: Functor effs => Comp (Reader env |> effs) res -> env -> Comp effs res
runReader comp env = case comp of
  Res res -> Res res
  Op (L (Ask () k)) -> runReader (k env) env
  Op (R other) -> Op (fmap (`runReader` env) other)
```

Exercise 55 *Build an example computation using* `Reader` *and* `State`*. What tree results after interpreting one of the effects?*

As usual, the fold can be generalized as a catamorphism:

```
handle
  :: Functor effs => (res -> d) -> (effs d -> d) -> Comp effs res -> d
handle val alg = \case
  Res res -> val res
  Op eff -> alg $ fmap (handle val alg) eff
```

[180] (youtube) What is algebraic about algebraic effects and handlers — Andrej Bauer.

Now in the implementation the continuation (subtree) is already interpreted into the desired domain:

```
runReader
  :: Functor effs => Comp (Reader env |> effs) res -> env -> Comp effs res
runReader = handle (\res _env -> pure res) \case
  L (Ask k) -> \env -> k env env
  R other -> \env -> Op (fmap ($ env) other)
```

When handlers are implemented as a built-in language feature, a design decision must be made: in the provided continuation, is the current effect already interpreted or not? The first variant is called **deep handlers**, the second — **shallow handlers**; they are expressible in terms of each other Hillerström and Lindley [2018].

Unfortunately, since the various effect variants are ordered both at the term level and at the type level, we need an operation that turns a computation with fewer effects into a computation with more:

```
liftF :: Functor effs => Comp effs res -> Comp (eff |> effs) res
liftF = \case
  Res x -> Res x
  Op effs -> Op $ R $ liftF <$> effs

example :: Comp (Reader Int |> State Int) ()
example = do
  env <- ask
  liftF (put env)
```

The ordering can be eliminated using type classes Swierstra [2008] (3.4.8).

One can notice that besides a large number of allocations, using `Monad Free` can lead to quadratic complexity of code due to linear passes in each bind. Various alternative encoding schemes exist Ploeg and Kiselyov [2014], Kiselyov and Ishii [2015].

Free monads find other, albeit analogous, applications: trampolining Bjarnarson [2012] and pipelines Kiselyov [2012b] [Bragilevsky, chapter 14].

7.2.7 Applications of handlers

Let us consider handlers of tail-resumptive operations, which treat continuations in a trivial way. They are needed for propagating values and functionality down the stack. As we discussed earlier 7.2.2, such handlers are analogous to having dynamic free variables or implicit function arguments 3.1.2, and, along with them, records or anonymous classes 4.6.2.

Among non-trivial use cases for continuations, we highlight the following:

- The continuation can be left uninvoked; thus, an exception mechanism can be implemented. However, in practice continuations may contain resource finalization logic. In that case, some special handling is still needed[181].

[181] https://koka-lang.github.io/koka/doc/book.html#sec-resource

- It can be invoked multiple times to emulate nondeterminism. This limits the use of efficient mutable continuations 6.5 Leijen [2018][182], and also requires non-trivial handling when working with resources.
- It can be invoked not immediately. This is needed for implementing mutable state, generators, and coroutines. For mutable state this is too expensive. Generators are a good application, but a built-in generator implementation can be more efficient by saving on continuation allocations (an `Iterator` is created directly)[183]. Implementing coroutines on top of handlers gives users the ability to write their own schedulers, which is the most valued application of handlers at the present time Sivaramakrishnan et al. [2021], Phipps-Costin et al. [2023].

7.2.8 Monad transformers

In the classical implementation of handlers via free monads 7.2.6, each handler produces an intermediate tree of uninterpreted operations. The desire to deforest 4.4.1 these intermediate trees arises naturally. One can go all the way and completely eliminate free monads and signature functors Wu and Schrijvers [2015]. In doing so, we effectively obtain the tagless final shallow embedding 5.4, another popular solution to the expression (stable denotations) problem — **monad transformers** Liang et al. [1995], Jones [1995][184]. In general, these approaches are equivalent in expressiveness, but may in practice have different characteristics when embedded in a language Schrijvers et al. [2019]. However, unlike transformers, handlers can be made a convenient built-in language feature.

7.2.9 Algebraicity and higher-order effects

In the early 2000s, the idea emerged of describing effect operations not with monads directly, but algebraically, using signatures and equations Plotkin and Power [2002], Bauer [2018]. Effects in this formalism are composable by construction, just as the signatures of algebraic theories are composable (concatenation of signatures is a signature 7.2.4); thus we have extensible syntax. However, a restriction is imposed on operations in the form of the **algebraicity property** — an operation commutes with the continuation and "bubbles up" to the top:

$$E[op(v, k)] \equiv op(v, \lambda x.\, E[k(x)])$$

The corresponding effects are called **algebraic**.

In our implementation (7.2.6), the algebraicity restriction can be seen as follows. Our operation signatures are functors in their continuations. This is used by both monadic bind for accumulating continuations and handlers for interpreting the continuations of unknown operations.

Many useful effects are algebraic, but not all. For example, catching exceptions `catch` is not an algebraic operation, since its signature would be written as follows:

[182] `https://koka-lang.github.io/koka/doc/book.html#sec-multi-resume`
[183] `https://csharpindepth.com/Articles/IteratorBlockImplementation`
[184] `https://hackage.haskell.org/package/mtl`

```haskell
data Catch e comp = Catch { try :: comp, onExn :: (e -> comp), next :: comp }
```

In this case the only reasonable functor implementation would also work with nested continuations, and the entire program, as a result of bind, would end up inside the `try` block.

In other words, *algebraic effects cannot accept other effectful computations as arguments*. It was precisely to model `catch` that handlers were originally proposed Plotkin and Pretnar [2013][185]. Thus, handlers have two tasks: delimiting the scope of certain functionality and interpreting effects. However, as we know, the order of handlers determines the resulting domain and, accordingly, the semantics. At the same time, the task of scope delimitation fixes the handler's position, which limits expressiveness and makes some domains inaccessible.

Operations that do not satisfy the algebraicity property correspond to **higher-order effects**, which can accept other effectful computations as arguments. Embeddings of handlers for such effects exist along with an extensive body of research on the topic Wu et al. [2014][186][187] Yang et al. [2022].

The key to higher-order effects lies in the ability to execute the effects of argument computations in the context of those handlers that are available at the call site of the higher-order effect van der Rest et al. [2022]. This is related to another idea, bidirectional effects, which allows the operation's implementation to produce effects at its call site (for example, throw exceptions there), an extremely necessary practical capability Zhang et al. [2020][188].

[185] Despite the fact that algebraic effects are simply a class of "well-behaved" operations, often when they are mentioned, effect handlers are implied.

[186] `https://github.com/fused-effects/fused-effects`

[187] (youtube) Building Haskell Programs with Fused Effects — Patrick Thomson

[188] `https://effekt-lang.org/docs/concepts/bidirectional`

8 Effect Systems TODO

TODO

9 Datatype-generic programming TODO

Earlier we said that everything programmers write is interpreters. It turns out that many of them can be obtained automatically by reaching a new level of generality. And code that does not have to be written — should not be written.

As a rule, such a situation arises when the interpretation is fully determined by the description (the shape) of the type. That is, by which data constructors are defined for it, with which fields, how those fields are named... Examples of such interpretations include serialisation into a particular format (e.g. json), structural equality, and so on.

The solution is for the language to support **structural polymorphism** [Maguire, a, chapter 13], or **datatype-generic programming**, as it is sometimes called.

One can imagine a language in which datatype-generic programming is just ordinary programming Chapman et al. [2010]. To achieve this, one takes a language with dependent types together with a distinguished universe of descriptions `Desc`. Concrete types are then obtained by applying an interpretation function `Desc -> Type`. Thus, parametric polymorphism is embodied as generalisation over a type, while structural polymorphism is generalisation over its description.

TODO

9.1 Specialisation

$$[\![[\![spec]\!](p, x)]\!](y) \equiv [\![p]\!](x, y)$$

TODOLiu [2023]

9.2 Ad-hoc solutions

TODO

9.3 Approaches to datatype-generic programming in Haskell

TODO

9.3.1 Template Haskell

TODO[189]

9.3.2 Deriving strategies

TODO[190]

[189] `https://markkarpov.com/tutorial/th.html`

[190] `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/deriving_strategies.html`

9.3.3 GHC.Generics

TODO[191]

9.3.4 Sum Of Products

TODO

9.3.5 Uniplate

TODO

[191] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/generics.html

10 Persistence and Optics TODO

In functional programming it is common to work with data persistently. In this chapter we consider when and why this is important. As well as functional optics — a set of abstractions for convenient work with data.

10.1 Persistent data structures

Data structures are divided into **mutable** and **immutable**. This is a classification solely by internal implementation — whether mutable memory cells are used or not.

By usage, one distinguishes **ephemeral** and **persistent** data structures. Changes to an ephemeral data structure can be observed through the same reference:

```
let xs = MutableList<Int>.empty()
xs.add(42)
print(xs)
```

Whereas modifying a persistent one produces new references each time, while the previous versions of the structure remain accessible through the old ones:

```
let xs = []
let xs' = 42 : xs
print xs
```

It might seem that ephemeral structures are mutable and persistent ones are immutable. In reality these are more or less orthogonal classifications. For instance, one can provide a persistent interface to a mutable data structure via copying, and an ephemeral interface to an immutable one (using the `State` monad).

In practice, when people talk about persistent data structures, they often mean data structures with a persistent interface that are specially optimized for it (not requiring a full copy on every operation). Such data structures can be surprisingly optimal and diverse Okasaki [1999].

For example, one can implement an efficient persistent array with logarithmic complexity for all operations. In Haskell, such a data structure is `Seq`[192][193]. If small arrays are stored in the nodes, which modern processor architectures can efficiently copy, one can substantially reduce the tree height and the algorithmic complexity of operations (e.g. `scala.immutable.Vector`).

Exercise 56 *Implement a persistent implicit-key treap in a non-Haskell language. Which features of Haskell complicate the use of this structure?*

When should one use ephemeral, and when persistent data structures? Comparing the two, one can find that ephemeral structures can be implemented more efficiently in many cases: processor

[192] `https://hackage.haskell.org/package/containers-0.7/docs/Data-Sequence.html`
[193] `http://www.staff.city.ac.uk/~ross/papers/FingerTree.html`

caches work better with local data, fewer allocations and indirections. At the same time, persistent structures are bound to be trees, in order to reallocate not the entire structure but only the path to the root.

Persistent structures, on the other hand, allow writing more modular and thread-safe code that need not account for the possibility of the structure being modified through a reference. Whereas working with an ephemeral structure is not pure code but involves producing observable side effects. Also, combining persistent results of different calls can be cheaper; for example, concatenation of persistent arrays is cheaper than concatenation of ephemeral ones (logarithmic complexity versus linear).

Thus, within a limited, easily surveyable scope it is better to use ephemeral structures due to their efficiency (for example, to initially populate a collection with elements). However, across abstraction boundaries it is better to pass only persistent structures (or rely on an effect system, see 8). That is, each data structure should support two phases of its life. For example, this is how it is done in Scala, where many persistent collections have a `Builder` version.

10.2 The idea of optics

Functional optics allows building functional references that focus on certain properties of data structures. Subsequently, using eliminating functions, one can read a property or persistently update it given a functional reference and an object. For example, we can focus on the first component of a pair using the reference `_1` and read the first component:

```
ghci> view _1 (42, 43)
42
```

Or we can focus on the property of membership of a certain element in a set `member`, set the property to `True`, and obtain a new set with the desired element:

```
ghci> set (member 42) (Set.fromList [1, 2]) True
Set.fromList [1, 2, 42]
```

Functional references compose (here, using %). So, if the set is in the first component of a pair, we can still add an element to it:

```
ghci> set (_1 % member 42) (Set.fromList [1, 2], 3) True
(Set.fromList [1, 2, 42], 3)
```

It is important to note that functional references, unlike ordinary pointers, are on the one hand more abstract and do not necessarily simply refer to a memory region, and on the other hand are detached from a particular object, thus more closely resembling pointers to methods in C++.

Thus, optics helps bring the convenience of programming with persistent data structures closer to that of programming with mutable ones. This is important because good programming practices should be at least as accessible as less optimal ones.

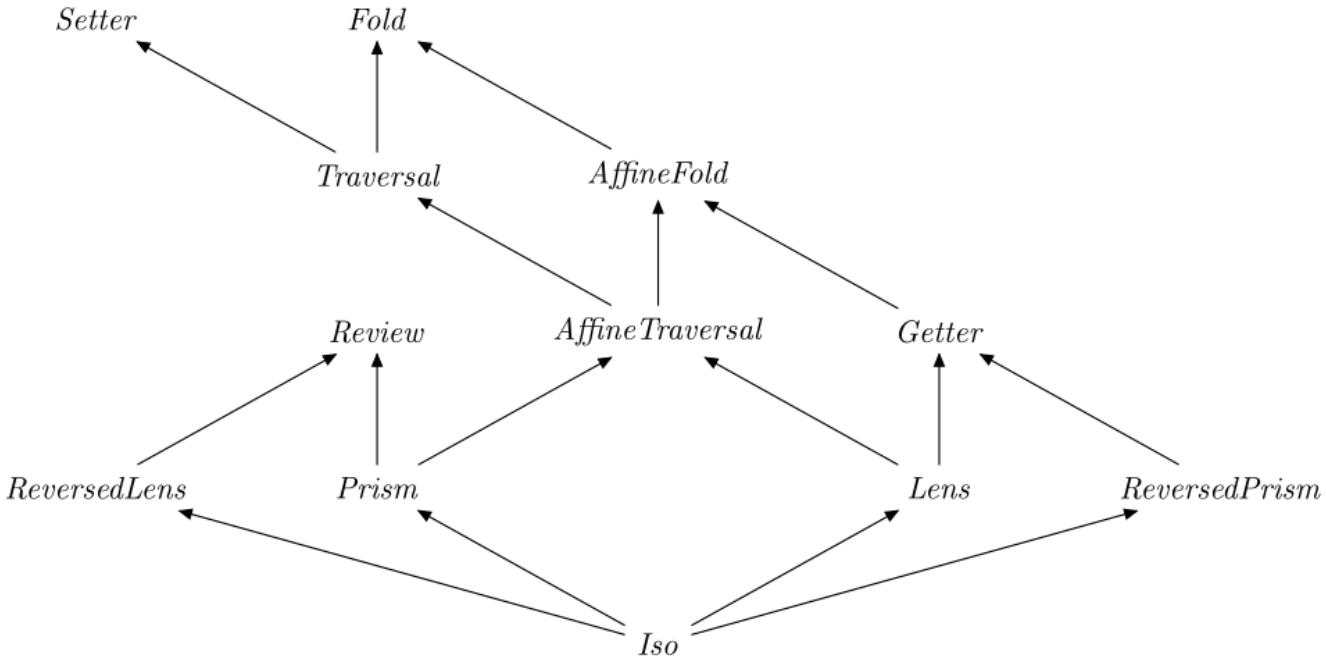


Figure 26: Hierarchy of optical devices of the `optics` library.

Functional references can provide various usage interfaces, which are called **optical devices**. Devices form a hierarchy (Fig. 26): when composing two devices, the result is the nearest common ancestor with functionality equal to the intersection of the functionalities of the originals.

Both references above are the optical device **lens**, which historically appeared first. Initially, lenses were proposed as a solution to the view-update problem in databases Bohannon et al. [2006], Foster et al. [2008]. Namely — how should the actual database be changed when the view is changed. Or how to reconstruct the whole after extracting and modifying a part. Lenses here serve as a means of bidirectional programming — they allow simultaneously describing both the view and the way to update. In parallel[194], lenses are mentioned in a series of blog posts describing an attempt to work more conveniently with mutable state when writing games[195][196] (lenses in games continue to delight[197]).

Numerous optics libraries exist for both Haskell[198][199][200] and other languages: Kotlin[201], Scala[202], Swift[203] (language support).

[194] `https://github.com/ekmett/lens/wiki/History-of-Lenses`
[195] (post) Making Haskell nicer for game programming
[196] (post) Haskell State Accessors (second attempt: Composability)
[197] `http://www.timphilipwilliams.com/posts/2019-07-25-minecraft.html`
[198] `https://hackage.haskell.org/package/optics-0.1/docs/Optics.html`
[199] `http://lens.github.io/`
[200] `https://github.com/marcosh/existential-optics/tree/main`
[201] `https://arrow-kt.io/`
[202] `https://github.com/optics-dev/Monocle`
[203] `https://github.com/swiftlang/swift-evolution/blob/main/proposals/0161-key-paths.md`

10.3 Using optics

As an example, let us consider the use of several optical devices from the `optics` library, which has excellent documentation[204].

Examples:

```
data Pet = Cat String | Dog Int deriving Show
newtype UserName = UserName { _getUserName :: String } deriving Show
data User = User { _userName :: UserName, _userCats :: [Pet] } deriving Show

exampleUser :: User
exampleUser = User (UserName "Bob") [Cat "Kitty", Dog 42, Dog 4]

exampleName :: User -> User
exampleName = over (userName % getUserName) (++ "!")

exampleIx :: User -> User
exampleIx = set (userName % getUserName % ix 2) 'k'

exampleFold :: User -> Int
exampleFold = getSum $ foldMapOf (userName % getUserName % folded) (const 1)

examplePrint :: User -> IO User
examplePrint = traverseOf (userCats % traversed % _Dog) (\x -> x <$ print x)
```

TODO

10.4 Data optics

10.4.1 Lenses

The simplest lens is formed by a pair of functions — viewing and setting a property:

```
data SimpleLens' s a = SimpleLens'
  { view' :: s -> a
  , set' :: s -> a -> s
  }
```

Natural laws are imposed on these functions:

```
view l (set l s x) ≡ x
set l s (view l s) ≡ s
set l (view l s x) y ≡ set l s y
```

[204] https://hackage.haskell.org/package/optics-0.4.2.1/docs/Optics.html

For example, one can easily modify a user's age:

```
newtype Age = Age { _getAge :: Int }
data User = User { _userName :: String, _userAge :: Age }

userAge :: SimpleLens' User Age
userAge = SimpleLens' { view' = _userAge, set' = \s x -> s { _userAge = x } }

getAge :: SimpleLens' Age Int
getAge = SimpleLens' { view' = _getAge, set' = \s x -> s { _getAge = x } }

ghci> set (userAge % getAge) user 1
```

One can generalize lenses to polymorphic lenses, which allow recreating the structure with new type parameters:

```
data SimpleLens s t a b = SimpleLens
  { view :: s -> a
  , set :: s -> b -> t
  }

_1 :: SimpleLens (a, c) (b, c) a b
_1 = SimpleLens { view = \(x, _) -> x, set = \(x, c) y -> (y, c) }
```

One can notice that a lens in this representation is a costate comonad coalgebra[205]. Indeed, the pair of functions can be replaced with a function returning a pair:

```
data Store a s = Store (a, a -> s)
data DataLens s a = DataLens (s -> Store a s)
```

If we define a comonad instance for `Store a s` and write out the laws of compatibility with the coalgebra, we obtain exactly the lens laws.

TODO

10.4.2 Prisms

Prisms allow obtaining a property that may be absent, and setting it when present. Thus, a specific field of a sum type may fail to be extracted if an unexpected constructor is passed.

Simple prisms can be defined analogously to lenses, accounting for possible absence of the property:

```
data SimplePrism' s a = SimplePrism'
  { preview' :: s -> Maybe a
  , review' :: a -> s
  }
```

[205] (youtube) Category Theory II 9.1: Lenses.

A polymorphic prism must provide the structure with an updated type in case the preview fails:

```
data SimplePrism s t a b = SimplePrism
  { preview :: s -> Either t a
  , review :: b -> t
  }
```

One can define a prism for working with the contents of the `Left` constructor:

```
_Left :: SimplePrism (Either a c) (Either b c) a b
_Left = SimplePrism
  { preview = \case Left a -> Right a; Right c -> Left (Right c)
  , review = Left
  }
```

Composition of prisms is defined in a natural way.

10.4.3 Composition of lenses and prisms

Let us introduce a type class with which we will learn to compose various combinations of lenses and prisms. The resulting device is uniquely determined by the operands of the composition.

```
class Composable o1 o2 o3 | o1 o2 -> o3 where
  compose :: o1 s t a b -> o2 a b c d -> o3 s t c d
```

Now we can naturally define composition for various devices. One can notice that this approach requires a quadratic number of instances in the number of optical devices.

```
instance Composable SimpleLens SimpleLens SimpleLens where
  compose :: SimpleLens s t a b -> SimpleLens a b c d -> SimpleLens s t c d
  compose l2 l1 = SimpleLens
    { view = view l1 . view l2
    , set = \s x -> set l2 s (set l1 (view l2 s) x)
    }
```

Composition of lenses and prisms forms a different device — affine traversals, which are something in between. The presence of a prism in the composition does not allow guaranteed viewing of the property, while a lens does not allow creating a new value without the original.

```
data AffineTraversal s t a b = AffineTraversal
  { apreview :: s -> Either t a
  , aset :: s -> b -> t
  }

instance Composable SimplePrism SimpleLens SimplePrism where
```

```haskell
compose :: SimplePrism s t a b -> SimpleLens a b c d -> SimplePrism s t c d
compose p l = SimplePrism
  { preview = fmap (view l) . preview p
  , pset = \s d -> case preview p s of
      Left t -> t
      Right a -> pset p s (set l a d)
  }
```

10.5 The path to profunctor optics

TODO

10.6 Generating optics

TODO

To be continued...

References

Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13:11–49, 2000. URL https://facweb.cdm.depaul.edu/smitsch/courses/csc447fa23/assets/articles/strachey-fundamental-concepts-in-programming-languages.pdf.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985. URL https://doi.org/10.1145/6041.6042.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

John Launchbury and Simon L Peyton Jones. State in haskell. *Lisp and symbolic computation*, 8(4): 293–341, 1995. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf.

Sandy Maguire. *Thinking with Types, Type-Level Programming in Haskell*. a. URL https://leanpub.com/thinking-with-types/?

Simon Peyton Jones. Type inference as constraint solving: how ghc's type inference engine actually works. Keynote talk at Zurihac 2019, 2019.

Benjamin C Pierce and David N Turner. Local type inference. *Acm transactions on programming languages and systems (toplas)*, 22(1):1–44, 2000. URL https://doi.org/10.1145/345099.345100.

David Raymond Christiansen. Bidirectional typing rules: A tutorial. 2013. URL https://davidchristiansen.dk/tutorials/bidirectional.pdf.

Jana Dunfield and Neelakantan R Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019. URL https://doi.org/10.1145/3290322.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. A quick look at impredicativity. *Proceedings of the ACM on Programming Languages*, 4 (ICFP):1–29, 2020. URL https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/.

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–437, 2020. URL https://link.springer.com/article/10.1208/s12249-010-9382-3.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, 2012. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/p53-yorgey.pdf.

Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System fc with explicit kind equality. *ACM SIGPLAN Notices*, 48(9):275–286, 2013. URL https://doi.org/10.1145/2544174.2500599.

Vitaly Bragilevsky. *Haskell in Depth*. Manning. URL https://www.manning.com/books/haskell-in-depth.

Bruno Blanchet. Escape analysis for object-oriented languages: application to java. *Acm Sigplan Notices*, 34(10):20–34, 1999. URL https://doi.org/10.1145/320385.320387.

Richard A Eisenberg and Simon Peyton Jones. Levity polymorphism. *ACM SIGPLAN Notices*, 52 (6):525–539, 2017. URL https://doi.org/10.1145/3140587.3062357.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007. URL https://www.researchgate.net/profile/Simon-Peyton-Jones/publication/221501761_A_history_of_Haskell_Being_lazy_with_class/links/0c960517e31f50f743000000/A-history-of-Haskell-Being-lazy-with-class.pdf.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989. URL https://doi.org/10.1145/75277.75283.

Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996. URL https://doi.org/10.1145/227699.227700.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *In Haskell Workshop*, 1997. URL https://courses.cs.washington.edu/courses/cse590p/06sp/multi.pdf.

Jeffrey R Lewis, John Launchbury, Erik Meijer, and Mark B Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, 2000. URL https://doi.org/10.1145/325694.325708.

Simon Peyton Jones. Type inference as constraint solving: how ghc's type inference engine actually works, 2019. URL https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

Martin Sulzmann, Gregory J Duck, Simon Peyton-Jones, and Peter J Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of functional programming*, 17(1): 83–129, 2007a. URL https://doi.org/10.1017/S0956796806006137.

Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, 2004. URL https://d1wqtxts1xzle7.cloudfront.net/43582096/Functional_pearl_implicit_configurations20160310-32037-1bu6179-libre.pdf.

Filip Křikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–28, 2019. URL https://doi.org/10.1145/3360589.

Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *ACM Sigplan Notices*, 45(10):341–360, 2010. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d30d65ca9ce7891352024a5c71ebe0ae8c41f7ac.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. *ACM SIGPLAN Notices*, 46(9):143–155, 2011. URL https://archive.alvb.in/msc/thesis/reading/typeclasses-agda_Devriese.pdf.

Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002. URL https://doi.org/10.1017/S0956796802004355.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, volume 1, pages 203–233, 2001. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2001/09/rules.pdf.

Sandy Maguire. *Algebra-Driven Design, Elegant Solutions from Simple Building Blocks*. b. URL https://leanpub.com/algebra-driven-design/.

Li-yao Xia. *Defunctionalization*. URL https://poisson.chat/aquarium/defunctionalization.pdf.

John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972. URL https://doi.org/10.1145/800194.805852.

John C Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11: 355–361, 1998. URL https://doi.org/10.1023/A:1010075320153.

Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *International Symposium on Functional and Logic Programming*, pages 119–135. Springer, 2014. URL https://www.cl.cam.ac.uk/~jdy22/papers/lightweight-higher-kinded-polymorphism.pdf.

Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *ACM SIGPLAN Notices*, 40(9):241–253, 2005a. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/at-syns.pdf.

Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, 2005b. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/assoc.pdf.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2008/01/icfp2008.pdf.

Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. *ACM SIGPLAN Notices*, 49(1):671–683, 2014. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/popl137-eisenberg.pdf.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. *Reflections on the Work of CAR Hoare*, pages 301–331, 2010. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/typefun.pdf?from=https://research.microsoft.com/~simonpj/papers/assoc-types/fun-with-type-funs/typefun.pdf&type=exact.

Mark P Jones. Type classes with functional dependencies. In *European Symposium on Programming*, pages 230–244. Springer, 2000. URL https://doi.org/10.1007/3-540-46425-5_15.

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. *Acm Sigplan Notices*, 49(10):233–249, 2014. URL http://lampwww.epfl.ch/~amin/dot/fpdt_post.pdf.

Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. Injective type families for haskell. *ACM SIGPLAN Notices*, 50(12):118–128, 2015. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1070&context=compsci_pubs.

Richard A Eisenberg and Jan Stolarek. Promoting functions to type families in haskell. *ACM SIGPLAN Notices*, 49(12):95–106, 2014. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1000&context=compsci_pubs.

Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in haskell. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–26, 2019. URL https://dl.acm.org/doi/pdf/10.1145/3341706.

Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In *International Symposium on Functional and Logic Programming*, pages 56–71. Springer, 2010. URL https://kar.kent.ac.uk/57498/1/constraint-families.pdf.

Edsko de Vries and Andres Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 83–94, 2014. URL https://doi.org/10.1145/2633628.2633634.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C d S Oliveira, and Philip Wadler. Quantified class constraints. *ACM SIGPLAN Notices*, 52(10):148–161, 2017. URL https://www.pure.ed.ac.uk/ws/portalfiles/portal/42495988/quantcc.pdf.

Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, 2007b. URL https://doi.org/10.1145/1190315.1190324.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of functional programming*, 21(4-5): 333–412, 2011. URL https://doi.org/10.1017/S0956796811000098.

Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 189–202, 2014. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1013&context=compsci_pubs.

Simon Peyton Jones, Stephanie Weirich, Richard A Eisenberg, and Dimitrios Vytiniotis. A reflection on types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 292–317. Springer, 2016. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1002&context=compsci_pubs.

Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 274–285, 2001. URL https://classes.cs.uchicago.edu/archive/2007/spring/32102-1/papers/p274-marlow.pdf.

Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 96–106, 2006. URL https://archive.alvb.in/msc/03_infoafp/papers/2012-12-18_HoorCollege_MarlowExtensibleExceptions_dk.pdf.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *NATO SCIENCE SERIES SUB SERIES III COMPUTER*

AND SYSTEMS SCIENCES, 180:47–96, 2001. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2e6c9d76f9cb690dc18019fc894ba9572a8c2812.

Ralf Hinze, Jennifer Hackett, and Daniel WH James. Functional pearl: F for functor. 2012. URL www.cs.ox.ac.uk/people/daniel.james/functor/functor.pdf.

Ralf Hinze and Daniel WH James. Reason isomorphically! In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 85–96, 2010. URL http://www.cs.ox.ac.uk/people/ralf.hinze/publications/WGP10.pdf.

Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997. URL https://doi.org/10.1017/S0956796897002864.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7de4f6fddb11254d1fd5f8adfd67b6e0c9439eaa.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *International Conference on Typed Lambda Calculi and Applications*, pages 16–30. Springer, 2003. URL http://www.strictlypositive.org/derivcont.pdf.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on functional programming languages and computer architecture*, pages 124–144. Springer, 1991. URL https://ris.utwente.nl/ws/portalfiles/portal/6142049/meijer91functional.pdf.

Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 324–333, 1995. URL https://dl.acm.org/doi/pdf/10.1145/224164.224225.

Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2007.10.007. URL https://www.sciencedirect.com/science/article/pii/S0167642309000227. Special Issue on Mathematics of Program Construction (MPC 2006).

Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358. Springer, 1988. URL https://link.springer.com/content/pdf/10.1007/3-540-19027-9_23.pdf.

СА Романенко. ВЫЯВЛЕНИЕ И ДОКАЗАТЕЛЬСТВО СВОЙСТВ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ МЕТОДАМИ СУПЕРКОМПИЛЯЦИИ. URL https://www.keldysh.ru/council/1/klyuchnikov-diss.pdf.

Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, 1993. URL https://dl.acm.org/doi/pdf/10.1145/165180.165214.

Jeremy Gibbons. Unfolding abstract datatypes. In *International Conference on Mathematics of Program Construction*, pages 110–133. Springer, 2008. URL https://ora.ox.ac.uk/objects/uuid:b51d2af6-25de-41ae-b3e7-9ac997de87b3/files/m7c19077c7bfba562a6cc9ceb3db62641.

William R Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 557–572, 2009. URL https://www.phaazon.net/media/uploads/on_understanding_data_abstraction.pdf.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. *ACM SIGPLAN Notices*, 42(9):315–326, 2007. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a4fad0d182605fcd155bebe3d620b7ffa0456968.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 285–299, 2017. URL https://arxiv.org/pdf/1612.06668.

Paul Downen, Zachary Sullivan, Zena M Ariola, and Simon Peyton Jones. Codata in action. In *European Symposium on Programming*, pages 119–146. Springer International Publishing Cham, 2019. URL https://library.oapen.org/bitstream/handle/20.500.12657/23330/1006825.pdf?sequence=1#page=132.

Jon Louis Bentley. Little languages. *Commun. ACM*, 29(8):711–721, 1986. URL https://doi.org/10.1145/6424.315691.

Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347, 2014. URL https://ora.ox.ac.uk/objects/uuid:0547e439-48d5-4749-a5eb-811e65cfaaee/files/mf31b32c6dcbd81dd6a8ef43166a8cc91.

Jeremy Gibbons. Functional programming for domain-specific languages. In *Central European Functional Programming School*, pages 1–28. Springer, 2013. URL https://ora.ox.ac.uk/objects/uuid:489909da-8818-4d6e-a4fd-0107b589ee76/files/m00be42b36cabbc6d11c8748fac8c7c71.

Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 16, 2010. URL https://idris-hackers.github.io/software-foundations/pdf/sf-idris-2018.pdf.

Graham Hutton. Fold and unfold for program semantics. *ACM SIGPLAN Notices*, 34(1):280–288, 1998. URL https://dl.acm.org/doi/abs/10.1145/291251.289457.

Robert Nystrom. *Crafting Interpreters*. URL https://craftinginterpreters.com/contents.html.

Dom Hutchinson. Language engineering - notes. Course COMS22201 notes. URL https://github.com/dajhutchinson/Language-Engineering.

Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School*, pages 339–401. Springer, 2011. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bd127ea1952996864c1542e1453973a78973ad5c.

Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49: 363–408, 2012. URL https://www.chargueraud.org/research/2009/ln/main.pdf.

Stephen Chong. *CS153: Compilers Lecture 12:Closures and Environments*. URL https://groups.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec12-Functions.pdf.

Oleg Kiselyov. Typed tagless final interpreters. In *Generic and indexed programming: International spring school, sSGIP 2010, oxford, uK, march 22-26, 2010, revised lectures*, pages 130–174. Springer, 2012a. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f0b6a04331821a294bad4d08975637b4f363ec5a#page=136.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM sigplan notices*, 23(7): 199–208, 1988. URL https://doi.org/10.1145/960116.54010.

Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129, 2011. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/remote.pdf?from=https://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/remote.pdf&type=exact.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. In *Programming Languages and Systems: 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007. Proceedings 5*, pages 222–238. Springer, 2007. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/finally-tagless-partially-evaluated-tagless-staged-interpreters-for-simpler-typed-lang7B2DC44A2127EBBA71ADE63809D9425F.

Eugenio Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for ..., 1988. URL http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-66/ECS-LFCS-88-66.pdf.

Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, 1990. URL https://dl.acm.org/doi/pdf/10.1145/91556.91592.

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992. URL https://dl.acm.org/doi/pdf/10.1145/143165.143169.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008. URL https://doi.org/10.1017%2FS0956796807006326.

John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6:233–247, 1993. URL https://ics.uci.edu/~jajones/INF102-S18/readings/08_histcont.pdf.

Peter J Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97), Technical report BRICS NS-96-13, University of Aarhus*, page 1. Citeseer, 1997. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=14c609276166517ed1afb56e6b84a41ba157030b.

Steven E Ganz, Daniel P Friedman, and Mitchell Wand. Trampolined style. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 18–27, 1999. URL https://doi.org/10.1145/317636.317779.

RO Bjarnarson. Stackless scala with free monads. *Scala Days*, 2012. URL https://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf.

Conor McBride. Clowns to the left of me, jokers to the right (pearl) dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, 2008. URL https://web.archive.org/web/20170705104305id_/http://www.cis.upenn.edu/~cis39903/static/clowns.pdf.

Daniel Hillerström. Foundations for programming and implementing effect handlers. 2022. URL http://dx.doi.org/10.7488/era/2122.

Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. Multiple resumptions and local mutable state, directly. *Proceedings of the ACM on Programming Languages*, 9(ICFP):704–733, 2025. URL https://doi.org/10.1145/3747529.

Olivier Danvy. Defunctionalized interpreters for programming languages. *ACM Sigplan Notices*, 43 (9):131–142, 2008. URL http://jfla.inria.fr/2014/danvy-ICFP08.pdf.

Jeremy Gibbons. Continuation-passing style, defunctionalization, accumulations, and associativity. *arXiv preprint arXiv:2111.10413*, 2021. URL https://doi.org/10.48550/arXiv.2111.10413.

Olivier Danvy. An analytical approach to program as data objects, 2006. URL https://www.cs.tufts.edu/comp/150FP/archive/olivier-danvy/d_scientia.pdf.

Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 133–144, 2014. URL https://doi.org/10.1145/2633357.2633360.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM sigplan notices*, 35(9): 233–243, 2000. URL https://doi.org/10.1145/357766.351262.

R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of functional programming*, 17(6):687–730, 2007. URL https://doi.org/10.1017/S0956796807006259.

Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457, 1994. URL https://doi.org/10.1145/174675.178047.

Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009. URL https://doi.org/10.1145/1462166.1462167.

Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 68–84, 2021. URL https://www.researchgate.net/profile/Mikhail-Belyaev-3/publication/355361443_Kotlin_coroutines_design_and_implementation/links/624c63c34f88c3119ce12a49/Kotlin-coroutines-design-and-implementation.pdf.

Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012. URL https://users-cs.au.dk/amoeller/spa/spa.pdf.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, 2017. URL https://doi.org/10.1145/3110257.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages*, 4 (ICFP):1–29, 2020. URL https://doi.org/10.1145/3408981.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices*, 48(12):59–70, 2013. URL https://doi.org/10.1145/2578854.2503791.

Andrej Bauer. What is algebraic about algebraic effects and handlers? *arXiv preprint arXiv:1807.05923*, 2018. URL https://doi.org/10.48550/arXiv.1807.05923.

Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems*, pages 415–435. Springer, 2018. URL https://www.research.ed.ac.uk/files/76099718/shallow_effect_handlers.pdf.

Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008. URL https://www.cs.tufts.edu/~nr/cs257/archive/wouter-swierstra/DataTypesALaCarte.pdf.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *ACM SIGPLAN Notices*, 50 (12):94–105, 2015. URL https://doi.org/10.1145/2887747.2804319.

Oleg Kiselyov. Iteratees. In *FLOPS'12: Proceedings of the 11th international conference on Functional and Logic Programming*, 2012b. URL https://okmij.org/ftp/Haskell/Iteratee/describe.pdf.

Daan Leijen. Algebraic effect handlers with resources and deep finalization. Technical report, Tech. Rep. MSR-TR-2018-10, Microsoft Research (April 2018), 2018. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2018/04/resource-v1.pdf.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 206–221, 2021. URL https://doi.org/10.1145/3453483.3454039.

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing webassembly with effect handlers. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):460–485, 2023. URL https://doi.org/10.1145/3622814.

Nicolas Wu and Tom Schrijvers. Fusion for free: Efficient algebraic effect handlers. In *International Conference on Mathematics of Program Construction*, pages 302–322. Springer, 2015. URL https://lirias.kuleuven.be/retrieve/322544.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995. URL https://dl.acm.org/doi/pdf/10.1145/199448.199528.

Mark P Jones. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*, pages 97–136. Springer, 1995. URL http://web.cecs.pdx.edu/mpj/pubs/springschool95.pdf.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pages 98–113, 2019. URL https://doi.org/10.1145/3331545.3342595.

Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002. URL https://era.ed.ac.uk/bitstream/handle/1842/196/Comp_Eff_Monads.pdf?sequence=1.

Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical methods in computer science*, 9, 2013. URL https://doi.org/10.2168/LMCS-9(4:23)2013.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 1–12, 2014. URL https://doi.org/10.1145/2633357.2633358.

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. Structured handling of scoped effects. In *European Symposium on Programming*, pages 462–491. Springer International Publishing Cham, 2022. URL https://library.oapen.org/bitstream/handle/20.500.12657/54028/1/978-3-030-99336-8.pdf#page=476.

Cas van der Rest, Jaro Reinders, and Casper Bach Poulsen. Handling higher-order effects. *arXiv preprint arXiv:2203.03288*, 2022. URL https://arxiv.org/pdf/2203.03288.

Yizhou Zhang, Guido Salvaneschi, and Andrew C Myers. Handling bidirectional control flow. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020. URL https://doi.org/10.1145/3428207.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. *ACM Sigplan Notices*, 45(9):3–14, 2010. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=55ab751e0af3808c741585df0210ca1785c235d6.

Yanhong A Liu. Incremental computation: What is the essence? *arXiv preprint arXiv:2312.07946*, 2023. URL https://arxiv.org/pdf/2312.07946.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. URL https://www.cs.tufts.edu/~nr/cs257/archive/chris-okasaki/dissertation.pdf.

Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, 2006. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8ee979ec0a1d8288a07a813c7bde91e4a2ae3187.

J Nathan Foster, Alexandre Pilkiewicz, and Benjamin C Pierce. Quotient lenses. *ACM Sigplan Notices*, 43(9):383–396, 2008. URL https://repository.upenn.edu/server/api/core/bitstreams/31dcfba8-8b4f-4143-bd2c-9443dc19c499/content.