Developing a Monadic Type Checker for an Object-Oriented Language: An Experience Report

Elias Castegren
Software and Computer Systems
KTH Royal Institute of Technology
Sweden
eliasca@kth.se

Kiko Fernandez-Reyes
Information Technology
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Abstract

Functional programming languages are well-suited for developing compilers, and compilers for functional languages are often themselves written in a functional language. Functional abstractions, such as monads, allow abstracting away some of the repetitive structure of a compiler, removing boilerplate code and making extensions simpler. Even so, functional languages are rarely used to implement compilers for languages of other paradigms.

This paper reports on the experience of a four-year long project where we developed a compiler for a concurrent, object-oriented language using the functional language Haskell. The focus of the paper is the implementation of the type checker, but the design works well in static analysis tools, such as tracking uniqueness of variables to ensure data-race freedom. The paper starts from a simple type checker to which we add more complex features, such as type state, with minimal changes to the overall initial design.

CCS Concepts • **Software and its engineering → Functional languages**; **Compilers**; *Object oriented languages*; • **Theory of computation →** *Type theory*.

Keywords functional programming, object-oriented languages, type systems, compilers

ACM Reference Format:
Elias Castegren and Kiko Fernandez-Reyes. 2019. Developing a Monadic Type Checker for an Object-Oriented Language: An Experience Report. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357766.3359545

SLE '19, October 20–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6981-7/19/10…$15.00
https://doi.org/10.1145/3357766.3359545

1 Introduction

Compilers for functional languages are often written in functional languages themselves. For example, the Haskell, OCaml, and F# compilers are written in their own respective language [1, 2, 5]. The recursive nature of functional programming, as well as constructs such as algebraic data types and pattern matching, lends itself well to traverse and manipulate abstract syntax trees, which is a large part of the task of a compiler. Functional programming languages provide abstractions, such as type classes or ML style modules, which allows writing concise and extensible code.

However, programmers are creatures of habit. Therefore, compilers for imperative or object-oriented languages tend to be written in languages of these paradigms, e.g., the Java compiler and the Scala compiler are written in their respective languages, and the `clang` compiler framework for C and C++ is written in C++ [3, 4, 6]. This habit prevents compiler writers from using attractive features from other languages. For example, a programmer developing a language tool in C will miss out on features like pattern matching and useful type system features that are not available in C.[1]

In this paper, we report on our experience using Haskell to develop the compiler for the object-oriented language Encore [10]. The full compiler consists of ≈15,000 lines of Haskell which compiles Encore to C. To make the presentation fit in a paper, we use a subset of the language and focus on the implementation of the type checker (≈7,000 lines in the full compiler). We start from a simple type checker and gradually refactor and extend it to make it more robust and feature rich. The monadic structure of the code allows these extensions with few or no changes to the original code.

The language started as a research project, but has since evolved to a full language with support for parametric polymorphism, subtyping via traits [31], concurrency, and a type system that prevents data-races between threads sharing state [12]. This paper reports on our experience creating a compiler for a research language, focusing on the features and robustness of the type checker. The techniques used are not novel on their own, but we have not seen them used in this combination for an object-oriented language.

[1] The same argument could be made in the other direction, since functional languages for example typically do not offer control over low-level details like memory layout.

Audience Our hope is that this paper can serve as a source of inspiration for language engineering researchers and compiler writers who want to use functional languages to develop compilers and tools for object-oriented languages. We expect the audience to be familiar with basic Haskell notation, the standard monads, and basic extensions.

Contributions

- An explanation of the initial design and implementation of a type checker for an object-oriented language, using common functional programming constructs (Sections 2–3).
- Descriptions of how to extend the type checker to handle backtraces, reporting warnings, and throwing multiple errors (Sections 4–6).
- A description of the use of phantom types to do type-level programming, which ensures that the type checker indeed type checks the entire program (Section 7).
- Explanation on how to add parametric polymorphism, subtyping via traits, and tracking of uniqueness of variables with no changes to the monadic design (Sections 8 – 10).

The main point of the work presented here is the ease of extension from the original type checker with new compiler functionality, and how easily one can add object-oriented features, such as subtyping via traits.

2 A Small Object-Oriented Language and its Typechecker

We define our object-oriented language below, and the encoding of the abstract syntax tree (*AST*) using algebraic data types in Figure 1.

$$
\begin{array}{rl}
\textit{Classes} & L ::= \textbf{class}\ C\{\overline{Q\ f : \tau};\ \overline{M}\} \\
\textit{Qualifiers} & Q ::= \textbf{var} \mid \textbf{val} \\
\textit{Methods} & M ::= \textbf{def}\ m(\overline{x : \tau}) : \tau\ \{\ \textbf{return}\ e\ \} \\
\textit{Binary operators} & B ::= + \mid - \mid * \mid / \\
\textit{Expressions} & e ::= x \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid e.m(\overline{e}) \mid e.f \\
& \quad \mid e\ B\ e \mid e(\overline{e}) \mid e = e \mid \textbf{new}\ C(\overline{e}) \\
& \quad \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid e : \tau \mid v \\
& v ::= \texttt{true} \mid \texttt{false} \mid n \mid \lambda(\overline{x : \tau}).e \mid \textbf{null} \\
\textit{Types} & \tau ::= C \mid \overline{\tau} \rightarrow \tau \mid \textbf{int} \mid \textbf{bool} \mid \textbf{unit}
\end{array}
$$

A program consists of a list of class definitions. A class definition contains a list of field and method definitions. A field definition has a name, a type, and a mutability modifier (**val** or **var**). A method definition has a name, a list of parameters, a return type, and a method body. Traits and subtyping are added in Section 9. Types are class names, function types or primitive types **int**, **bool** or **unit**.

Expressions are mostly standard for an object-oriented language. For simplicity we use a **let** expression for introducing names and handling scopes (in the Encore compiler, an earlier phase translates an imperative style of variable declarations into **let** form). Sequencing of statements can be emulated by binding the result of the statement to an unused variable: "`s1; s2`" $\Longrightarrow$ "`let _ = s1 in s2`"; or by using a "sequence expression" containing a list of expressions, only the last of which is used for its value (this is the approach taken in the Encore compiler). An anonymous function $\lambda(\overline{x : \tau}).e$ (`Lambda` in Figure 1) takes zero or more parameters. Note the difference between method calls, which call a method on a specified target, and function calls, which call a value of function type.

The task of the type checker is twofold: to ensure that the *AST* is well-formed, and to decorate each expression node with its type. Every expression in the *AST* data type has a field `etype`, which has value **`Nothing`** after parsing and which will be assigned a type during the type checking phase (this part of the design will be improved in Section 7). Additionally, the type checker uses an environment, storing a map of all the classes of a program and all variables currently in scope. Since we only allow assigning to immutable **val** fields if we are currently in a constructor method, we also store a boolean flag to track this.

```
data Env = Env {ctable :: Map Name ClassDef
               ,vartable :: Map Name Type
               ,constructor :: Bool}
```

We define a type class [37] `Typecheckable` with a single function `typecheck` which takes such an environment and an *AST* node, and returns either the decorated node, or an error value (`TCError`) representing one of the ways type checking can fail. To avoid having to check if each computation fails or not, we use the exception monad [35]:

```
data TCError = TypeMismatchError Type Type
  | UnknownClassError Name
  | NonArrowTypeError Type
  | NonClassTypeError Type
  | NonArrowTypeError Type
  | UninferrableError Expr
  | ..

class Typecheckable a where
  typecheck :: Env → a → Except TCError a
```

Figure 2 shows instances of `Typecheckable` for programs, classes, fields, types and expressions (for brevity, we show an excerpt). Type checking a program corresponds to checking its classes; type checking a class corresponds to extending the environment to include the variable **`this`** (Line 7), and checking its fields and methods (Lines 8–9); type checking a type corresponds to checking that named classes exist (Lines 17–27). Lines 30–54 type check a function call and an assignment.

```
type Name = String
newtype Program = Program [ClassDef]

data ClassDef = ClassDef {cname :: Name, fields :: [FieldDef]
                         ,methods :: [MethodDef]}

data FieldDef = FieldDef {fname :: Name, ftype :: Type
                         ,fmod :: Mod}

data MethodDef = MethodDef {mname :: Name, mparams :: [Param]
                           ,mtype :: Type, mbody :: Expr}

data Type = ClassType Name | IntType | BoolType
  | Arrow {tparams :: [Type], tresult :: Type}
  | UnitType deriving (Eq)
```

```
data Expr = BoolLit {etype :: Maybe Type, bval :: Bool}
  | IntLit {etype :: Maybe Type, ival :: Int}
  | Lambda {etype :: Maybe Type, params :: [Param], body :: Expr}
  | VarAccess {etype :: Maybe Type, name :: Name}
  | FieldAccess {etype :: Maybe Type,
                 target :: Expr, name :: Name}
  | Assignment {etype :: Maybe Type, lhs :: Expr, rhs :: Expr}
  | MethodCall {etype :: Maybe Type, target :: Expr,
                name :: Name, args :: [Expr]}
  | FunctionCall {etype :: Maybe Type,
                  target :: Expr, args :: [Expr]}
  | Let {etype :: Maybe Type, name :: Name,
         val :: Expr, body :: Expr}
  | ...
```

Figure 1. Encoding of *AST* nodes using algebraic data types

```
1  instance Typecheckable Program where
2    typecheck env (Program cls) =
3      Program <$> mapM (typecheck env) cls
4
5  instance Typecheckable ClassDef where
6    typecheck env cdef@ClassDef{cname, fields, methods} = do
7      let env' = addVariable env "this" (ClassType cname)
8      fields' ← mapM (typecheck env') fields
9      methods' ← mapM (typecheck env') methods
10     return cdef{fields = fields', methods = methods'}
11
12 instance Typecheckable FieldDef where
13   typecheck env fdef@FieldDef{ftype} = do
14     ftype' ← typecheck env ftype
15     return fdef{ftype = ftype'}
16
17 instance Typecheckable Type where
18   typecheck env (ClassType c) = do
19     _ ← lookupClass env c
20     return $ ClassType c
21   typecheck _ IntType = return IntType
22   typecheck _ UnitType = return UnitType
23   typecheck _ BoolType = return BoolType
24   typecheck env (Arrow ts t) = do
25     ts' ← mapM (typecheck env) ts
26     t' ← typecheck env t
27     return $ Arrow ts' t
```

```
28 instance Typecheckable Expr where
29   ...
30   typecheck env e@(FunctionCall {target, args}) = do
31     target' ← typecheck env target
32     let targetType = getType target'
33     unless (isArrowType targetType) $
34       throwError $ NonArrowTypeError targetType
35     let paramTypes = tparams targetType
36         resultType = tresult targetType
37     args' ← zipWithM hasType args paramTypes
38     return $ setType resultType e{target = target', args = args'}
39
40   typecheck env e@(Assignment {lhs, rhs}) = do
41     unless (isLVal lhs) $
42       throwError $ NonLValError lhs
43     lhs' ← typecheck env lhs
44     let lType = getType lhs'
45     rhs' ← hasType env rhs lType
46     checkMutability lhs'
47     return $ setType UnitType e{lhs = lhs', rhs = rhs'}
48     where
49       checkMutability e@FieldAccess{target, name} = do
50         field ← lookupField env (getType target) name
51         unless (isVarField field ||
52                 constructor env && isThisAccess target) $
53           throwError $ ImmutableFieldError e
54       checkMutability _ = return ()
```

Figure 2. Type checking functions for programs, classes, fields, types and expressions (excerpt)

Note the helper function `checkMutability` which ensures that immutable **`val`** fields are only assigned through **`this`** in a constructor. The function `hasType` (*e.g.*, Lines 37 and 45) is used whenever a certain type is expected (*cf.* bidirectional typechecking [27]). To throw an error, we use the function `throwError` from the exception monad library.

For the Haskell novice, it is worth pointing out some of the constructs and functions used here (and later in the paper). Almost all functions pattern matching on some algebraic data type uses the `NamedFieldPuns` extension, which binds record fields of a parameter to variables of the same name (*e.g.*, Lines 6 and 13). The `($)` operator simply applies the function on its left to the value on its right (*e.g.*, Lines 33, 34 and 38), which is mainly used for avoiding parentheses: `f (g x) == f $ g x`. Similarly, the `(<$>)` operator applies a function to the value inside a monad[2] (*e.g.*, Line 3), which avoids having to extract the value, apply the function, and put it back in the monad.

[2]Technically, the value only needs to be inside a functor, but this distinction is not important here.

The entry point to the type checker takes a program and runs the monadic type checking computation (with `runExcept`) under a built environment (`genEnv p`), and returns either an error or the decorated program:

```
tcProgram :: Program → Either TCError Program
tcProgram p = runExcept $ typecheck (genEnv p) p
```

3 Refactoring: Removing Boilerplate

The type checking function, `typecheck :: Env → a → Except TCError a`, expects an environment, passed to each type checked node. This environment can be updated in the current scope, but the changes are forgotten as we exit the current scope – the type checker does not return an updated environment. For example, when type checking a class declaration, the variable **`this`** (line 2) is added to the environment before type checking the fields and methods of the class declaration, as shown below (lines 3–4). When the fields and methods have been type checked, the variable **`this`** is no longer accessible in the environment:

```
1  typecheck env cdef@ClassDef{cname, fields, methods} = do
2    let env' = addVariable env thisName (ClassType cname)
3    fields' ← mapM (typecheck env') fields
4    methods' ← mapM (typecheck env') methods
5    return cdef{fields = fields', methods = methods'}
```

A better design considers the environment as a global entity, and allows running computations under a modified environment when required. This is the precise definition of the `Reader` monad [21]! We update the `typecheck` function to work with the `Reader` monad. In functional programming languages, monad transformers compose monadic behaviours [22], so that we can stack the `Reader` monad onto the `Except` monad. We use an alias to define a new type for the return type of the type checking function, using common type classes for monads:

```
type TypecheckM a =
    forall m. (MonadReader Env m, MonadError TCError m) ⇒ m a

class Typecheckable a where
  typecheck :: a → TypecheckM a
```

Now, instances of the `Typecheckable` class do not pass the environment around explicitly – instead it is available from the `Reader` monad. With the use of the standard function **`local`** `:: MonadReader r m ⇒ (r → r)→ m a → m a`, the compiler runs a computation in a modified environment. For example, type checking fields and methods of a class run under a modified environment – the helper function `addVariable` injects the variable **`this`** to the environment.

```
instance Typecheckable ClassDef where
  typecheck cdef@ClassDef{cname, fields, methods} = do
    let thisAdded = local $ addVariable thisName (ClassType cname)
    fields' ← thisAdded $ mapM typecheck fields
    methods' ← thisAdded $ mapM typecheck methods
    return cdef{fields = fields', methods = methods'}
```

To run the type checker, we update the function `tcProgram` to run the reader monad transformer, which returns the exception monad. As before, we run the exception monad with the function `runExcept`, returning either a type checking error or the program with typing information.

```
tcProgram :: Program → Either TCError Program
tcProgram p = runExcept (runReaderT (typecheck p) (genEnv p)))
```

4 Extension: Support for Backtraces

The current type checker returns the first error that occurs, but does not point out *where* this error occurred (*e.g.*, in which method, in which class). In this section, we add support for backtraces, *i.e.*, tracking the positional context of the type checker. The backtrace of a program is a list of backtrace nodes. Each backtrace node has specific information about the *AST* node that it represents.

```
newtype Backtrace = Backtrace [BacktraceNode]
data BacktraceNode = BTClass ClassDef
  | BTMethod MethodDef
  | BTField FieldDef
  | BTParam Param
  | BTExpr Expr
  | BTType Type
```

Type checking errors should have access to their backtrace, so that the compiler displays the backtrace as well as the error. We update the error data constructor, and create a separation of concerns between the error and its available backtrace:

```
data TCError = TCError Error Backtrace
data Error = UnknownClassError Name
  | TypeMismatchError Type Type
  | ImmutableFieldError Expr
  | ...
```

The backtrace is updated whenever the type checker visits a new node, so that the position is tracked properly. The backtrace and type checker are similar to a stack machine: as soon as an *AST* node is type checked, the compiler creates a backtrace node, and uses this information upon throwing an exception. If there is no exception, the compiler pops the node and gets back to the previous backtrace state. We place the backtrace in the environment, so that it is available from the `Reader` monad:

```
data Env = Env {bt :: Backtrace
               ,ctable :: Map Name ClassDef
               ,vartable :: Map Name Type
               ,constructor :: Bool}
```

Even though here we are only using the backtrace to improve error messages, it could also be used to query the current context. For example, we could check whether we are currently in a constructor method without tracking it using the boolean flag `constructor`. This is how the Encore compiler does it, but for simplicity in this presentation we keep the boolean flag.

We can take advantage of the recursive nature of the type checker and update the backtrace before checking each node. To do this, we declare a type class `Backtraceable`, and create instances for each *AST* node. `Backtraceable` has functions `backtrace` and `push`: the former converts an *AST* node into a `BacktraceNode`; the latter prepends an *AST* node to a backtrace. We also create instances of the `backtrace` function for each *AST* node:

```
class Backtraceable a where
  backtrace :: a → BacktraceNode
  push :: a → Backtrace → Backtrace
  push x (Backtrace bt) = Backtrace (backtrace x : bt)

instance Backtraceable ClassDef where backtrace = BTClass

instance Backtraceable MethodDef where backtrace = BTMethod
...
```

To throw an error, we replace calls from `throwError` to `tcError`, which retrieves the backtrace from the environment, appends it to the type checking error, and throws the error.

```
tcError err = do
  bt ← asks bt
  throwError $ TCError err bt
```

We refactor the `Typecheckable` class with a new function, `doTypecheck`, performing the actual type checking, and define a default implementation of the `typecheck` function, which converts and pushes the current node to the backtrace, before type checking:

```
class Typecheckable a where
  doTypecheck :: a → TypecheckM a

  typecheck :: (Backtraceable a) ⇒ a → TypecheckM a
  typecheck x = local pushBT $ doTypecheck x
    where pushBT env@Env{bt} = env{bt = push x bt}
```

Finally, we change `typecheck` to `doTypecheck` in the *left-hand side* of all function definitions. The bodies of the type checking functions can remain as they are! The extension of adding backtracing can be done *without rewriting the original implementation of the type checker.*

5 Extension: Addition of Warnings

A compiler should inform the developer of patterns that may trigger a bug, *e.g.*, shadowing a variable. In this section, we add compiler support for warnings: the type checker will accumulate warnings and return them after type checking, if there are no errors. A warning contains the information needed to create an understandable message, as well as a backtrace that identifies the position of the expression that triggered it:

```
data TCWarning = TCWarning Warning Backtrace
data Warning = ShadowedVarWrn Name | UnusedVariableWrn Name | ...
```

At a first glance, one could think that a warning can be modelled inside the exception monad. However, the exception monad aborts type checking upon finding an error, whereas after finding a warning the type checker should continue, accumulating as many warnings as possible. Thus, warnings can be seen as a monotonically increasing list, for which we will use the `Writer` monad [21]. To integrate warnings into the type checking monad, we simply add a new constraint to the monad:

```
type TypecheckM a = forall m. (MonadReader Env m,
    MonadError TCError m, MonadWriter [TCWarning] m) ⇒ m a
```

This additional constraint requires us to update the entry point of the type checker, since we also need to run the writer monad.

```
tcProgram :: Program → Either TCError (Program, [TCWarning])
tcProgram p = runExcept $
                runReaderT (runWriterT (doTypecheck p)) (genEnv p)
```

To emit a warning, we create a function that accesses the environment, fetches the current backtrace, and appends it to the warning:

```
tcWarning wrn = asks bt >>= \x → tell [TCWarning wrn x]
```

We use the standard function `tell :: MonadWriter w m ⇒ w → m ()`, where `MonadWriter` requires the type variable `w` to be a monoid. This is trivially satisfied by the use of a list (of warnings). As an example, the following code raises a warning if introducing a variable shadows an old one:

```
checkShadowing name = do
  shadow ← isBound name
  when shadow $
    tcWarning $ ShadowedVarWarning name
```

Notice how *we did not change the implementation of the type checker*, except when the type checker raises warnings.

6 Extension: Support Multiple Errors

The type checker reports a single error at a time. However, finding an error in one class should not prevent type checking other classes.

To support multiple errors, we introduce a new combinator which "forks" two computations in the `Except` monad, and either returns both of their results, or aggregates the errors of one or both of the computations. We abstract over how the errors are aggregated by requiring that the error data type is a `Semigroup`:

```
(<&>) :: (Semigroup e, MonadError e m) ⇒ m a → m b → m (a, b)
tc1 <&> tc2 = do
  res1 ← (tc1 >>= return . Right) `catchError` (return . Left)
  res2 ← (tc2 >>= return . Right) `catchError` (return . Left)
  case (res1, res2) of
    (Right v1, Right v2) → return (v1, v2)
    (Left e1, Left e2) → throwError $ e1 <> e2
    (Left e1, _) → throwError e1
    (_, Left e2) → throwError e2
```

The forking combinator can also be used for mapping over lists:

```
forkM :: (Semigroup e, MonadError e m) ⇒ (a → m b) → [a] → m [b]
forkM _ [] = return []
forkM f (x:xs) = uncurry (:) <$> (f x <&> forkM f xs)
```

We generalise our exception type to contain a list of errors. A failing computation always contains at least one error, so we use the `NonEmpty` list type and update the error throwing function:

```
newtype TCErrors = TCErrors (NonEmpty TCError) deriving (Semigroup)
type TypecheckM a = forall m.
    (MonadReader Env m, MonadError TCErrors m) ⇒ m a

tcError :: Error → TypecheckM a
tcError err = asks bt >>=
   \x → throwError $ TCErrors (NE.fromList [TCError err x])
```

Now, we insert the forking combinators wherever we want to aggregate error messages. In general, calls to **`mapM`** can be replaced by `forkM`. We update the implementation, and show an example that aggregates errors from type checking fields and methods of a class:

```
instance Typecheckable ClassDef where
  doTypecheck cdef@ClassDef{cname, fields, methods} = do
    (fields', methods') ←
      local $ addVariable thisName (ClassType cname) $
        forkM typecheck fields <&> forkM typecheck methods
    return cdef{fields = fields', methods = methods'}
```

In contrast to previous extensions, here we actually need to change small parts of the implementation, to aggregate multiple errors. However, the changes are never more complicated than replacing **`mapM`** by `forkM` and changing

```
r1 ← e1
e2 ← e2
```

into `(r1, r2) ← e1 <&> e2`.

***Multiple Errors Using* `ApplicativeDo`** An alternative approach to aggregating multiple errors is based on using applicative functors (the `Applicative` type class). Since the composition of two computations in an applicative functor cannot depend on each other, it is straightforward to merge the errors if they both fail. In the spirit of the previous sections however, we would like to avoid having to rewrite the existing monadic compiler in an applicative style.

A way forward comes via the `ApplicativeDo` extension for GHC [24]. This extension allows desugaring **`do`** notation, which normally is only available for monads, to use the operations of applicative functors whenever there are no dependencies between computations. We start by redefining the `Except` data type used in our type checker so that it is an instance of `Applicative`, where composition of two error values merges the errors:

```
data Except err a = Result a | Error err deriving(Show, Functor)

instance Semigroup err ⇒ Applicative (Except err) where
  pure = Result
  Result f <*> Result a = Result $ f a
  Error e1 <*> Error e2 = Error $ e1 <> e2
  Error e1 <*> _ = Error e1
  _ <*> Error e2 = Error e2

instance Semigroup err ⇒ Monad (Except err) where
  return = pure
  Result a >>= f = f a
  Error e >>= _ = Error e
  (>>) = (*>)
```

By introducing the same definition of `TCErrors` as above and enabling `ApplicativeDo`, we can get multiple error messages in places where there are no dependencies between computations without changing the type checking code at all![3] This technique seems to be known in the Haskell community, but we do not know where it was first introduced.

7 Refactoring: Type State Phases

One unattractive part of the current design (as well as the current design of the Encore compiler) is the way each expression carries its type around as a value which might be **`Nothing`**. Forgetting to annotate some subexpression with its type in the type checking phase will cause later phases to abort when the necessary information is not present. We could get around this by using two different data types—one

[3] However, this breaks the expected invariant that `<*>` has the same behaviour as **`ap`** from the **`Monad`** type class, since the monad does *not* merge error values.

with type annotations and one without—but this would require us to duplicate the expression type and keep the two versions in sync.

Really, what we would like is for our implementation to be able to track which phase we are currently in, and only allow querying an expression for its type if it has passed the type checking phase. Similarly, an expression should only be allowed to pass the type checking phase if it has indeed been annotated with a type. Intuitively, we want the type checking function to have the type "unannotated expression → annotated expression", and the `getType` function to only accept annotated expressions.

We propose type state [7, 20] as a solution to this problem, based on phantom types [18]. Instead of storing the type of an expression as **`Maybe Type`**, we are going to parameterise the expression data type (and the other data types that pass through the type checker) with a type representing the current phase, which in turn carries the functor that the expression wraps its type in:

```
data Expr (p :: Phase f) =
    BoolLit {etype :: f (Type p), bval :: Bool}
  | IntLit {etype :: f (Type p), ival :: Int}
  | Lambda {etype :: f (Type p)
           ,params :: [Param p]
           ,body :: Expr p}
  | MethodCall {etype :: f (Type p)
               ,target :: Expr p
               ,name :: Name
               ,args :: [Expr p]}
  | ...
```

Note that `p` is a type parameter of *kind* `Phase f`. The type parameter `f` is going to be the `Proxy` functor (from `Data.Proxy`) for unannotated expressions, and the `Identity` functor (from `Data.Functor.Identity`) for annotated expressions. The former discards its wrapped value, while the latter stores it as is:

```
data Proxy a = Proxy
newtype Identity a = Identity {runIdentity :: a}
```

To define the `Phase` kind, we lift a regular data type to the kind level using the GHC extension `DataKinds`. Since different phases carry different functors, we define it as a GADT [25]:

```
data Phase (f :: * → *) where
  Parsed :: Phase Proxy
  Checked :: Phase Identity
```

With this change, we define the `getType` function as

```
getType :: Expr 'Checked → Type 'Checked
getType e = runIdentity (etype e)
```

meaning it cannot be called unless the expression has been type checked. The type checking functions take an *AST* node from phase `Parsed` to phase `Checked`. Note that we cannot use the (more intuitive) type "`a 'Parsed → a 'Checked`", since the two occurrences of the type parameter `a` would have different kinds. Instead we use two different parameters, related by functional dependency:

```
class Typecheckable a b | a → b where
  doTypecheck :: a 'Parsed → TypecheckM (b 'Checked)

  typecheck :: (Backtraceable (a 'Parsed))
            ⇒ a 'Parsed → TypecheckM (b 'Checked)
  typecheck x = local (pushBT x) $ doTypecheck x
```

An important change is that the environment used for type checking must only contain well-formed types, which requires us to check the interfaces of every class before checking any expressions. Note that due to a chicken-and-egg problem, the environment can no longer contain full class definitions: in order to run the type checker we need a well-formed environment, but in order to get an environment containing well-formed classes we would need to run the type checker! Instead we change the environment to use special entries which only contain the (well-formed) types of classes, methods and fields (Figure 3). When building the environment, we use a simpler kind of environment which we dub a *pre-environment* which simply contains a list of all the valid class names, allowing us to check the well-formedness of types. We call the process of checking the types used by classes, fields and methods *pre-checking*, and use a type class scheme similar to the main type checker:

```
class Precheckable a b | a → b where
  doPrecheck :: a → TypecheckM b

  precheck :: (Backtraceable a) ⇒ a → TypecheckM b
  precheck x = local (pushBT x) $ doPrecheck x
```

Note that we reuse our type checking monad from before, including any of the previous extensions we might have added. For each kind of *AST* node `a`, we define an instance `Precheckable a b` which returns an entry of type `b` that can be used by the environment being generated. For example, pre-checking a class generates a `ClassEntry`, containing the (well-formed) types of all fields and methods (Figure 3).

After pre-checking, we have a well-formed environment (Line 5 below) that we can use to type check the program just as before. With these changes, the entry point to the type checker takes an undecorated program and returns either a list of errors or the decorated program and a list of warnings.

```
1  tcProgram :: Program 'Parsed
2            → Either TCErrors (Program 'Checked, [TCWarning])
3  tcProgram p = do
4    let preEnv = generatePreEnv p
5    (env, _) ← runExcept $ runReaderT (runWriterT (genEnv p)) preEnv
6    runExcept runReaderT (runWriterT (doTypecheck p)) env
```

```
data MethodEntry =
  MethodEntry {meparams :: [Param 'Checked]
              ,metype :: Type 'Checked}

data FieldEntry =
  FieldEntry {femod :: Mod
             ,fetype :: Type 'Checked}

data ClassEntry =
  ClassEntry {cefields :: Map Name FieldEntry
             ,cemethods :: Map Name MethodEntry}

data Env =
    PreEnv {classes :: [Name]}
  | Env {ctable :: Map Name ClassEntry
        ,vartable :: Map Name (Type 'Checked)
        ,constructor :: Bool}
```

```
genEnv :: Program 'Parsed → TypecheckM Env
genEnv (Program classes) = do
  clsEntries ← mapM precheck classes
  let cnames = map cname classes
      duplicates = cnames \\ nub cnames
  unless (null duplicates) $
    throwError $ DuplicateClassError (head duplicates)
  return Env {vartable = Map.empty
             ,ctable = Map.fromList $ zip cnames clsEntries}

instance Precheckable (ClassDef 'Parsed) ClassEntry where
  precheck ClassDef {fields, methods} = do
    fields' ← mapM precheck fields
    methods' ← mapM precheck methods
    let (fields'', methods'') = (map fname fields, map mname methods)
    return ClassEntry {cefields = Map.fromList $ zip fields'' fields'
                      ,cemethods = Map.fromList $ zip methods'' methods'}
```

Figure 3. Construction of an environment from the *parsed* phase

This design decouples the type checking of interfaces from type checking the rest of a program, *e.g.,* type errors in field or method definitions will not cause errors due to ill-formed types when type checking field accesses or method calls.

By tracking the current phase in the type of an *AST* node, we ensure that all *AST* nodes go through the `typecheck` function, and that it indeed returns checked *AST* nodes. Haskell statically prevents compiler writers from using undecorated *AST* nodes where one expects them to have typing information. With the exception of the environment generation and some type signatures, *the original implementation of the type checker did not change notably.*

8 Feature: Parametric Polymorphism

Parametric polymorphism, or in the Java jargon, generics [9, 11], allows code to be parameterised over one or more abstract type parameters, which can be instantiated with different concrete types at different use sites. The addition of generics to the type checking monad adds an auxiliary field to the environment, `typeParameters`, which keeps track of the available type parameters (which may be used as types).

```
data Env = Env {ctable :: Map Name ClassDef
               ,vartable :: Map Name Type
               ,typeParameters :: [Type]
               ,bt :: Backtrace
               ,constructor :: Bool}
```

The (initially empty) list of type parameters is built with the construction of the environment. Whenever polymorphic code introduces new type variables, one proceeds by running the type checker under a modified environment which contains these new type variables. For example, the code below shows type checking for a polymorphic method. The field `mtparams` contains a list of the formal type parameters of a method. These are added to the environment when checking the types of the method parameters[4] and the return type of the method, as well as when checking the body of the method:

```
doTypecheck m@(Method {mname, mtparams, mparams, mtype, mbody}) = do
  local (addTypeParameters mtparams) $ mapM_ typecheck mparams
  local (addTypeParameters mtparams) $ typecheck mtype
  eBody ← local (addTypeParameters mtparams .
                 setConstructor mname .
                 addParams mparams) $ hasType mbody mtype
```

In places where generic classes or methods are used, a map from formal type parameters to type arguments can be used to translate a polymorphic type to a concrete type. For example, if the following generic class

```
class C[a]
  val f : a
  def init(f: a): unit
    this.f = f
  end
  def get(): a
    this.f
  end
end
```

is instantiated as `C[int]`, the type checker applies the substitution $\{a \mapsto \texttt{int}\}$ to all looked up field and method types of the class Some care must be taken to make type variables unique, but this can be achieved with regular alpha conversion [26].

It is straightforward to infer type arguments for constructors of polymorphic classes, or polymorphic method and function calls, as long as all the type parameters are used in the arguments of the call. For example, given the class above, the type of the expression `new C(42)` is inferred as `C[int]`. If

[4] Notice the use of **`mapM_`** instead of **`mapM`**, which ignores the result of the operation.

a type variable is not used in a call (*cf.* phantom types [18]), explicit type arguments must be provided.

9 Feature: Subtyping

Subtyping is often considered one of the core features of object-oriented programming. Subtype polymorphism allows passing a value of type S to code that expects a value of type T, whenever S is a subtype of T. In Encore, we implement subtyping via *traits* [31]. This choice does not affect the overall design of the type checker much when compared to other subtyping mechanisms, such as class inheritance or interfaces.

Traits can be thought of as a special kind of Java style interfaces, which can provide default implementations for methods (like in Java 8), but which may also require the presence of other methods or fields in the class including it. Overlapping requirements of two included traits is allowed, whereas overlapping method implementations need to be overridden by the including class.

Figure 4 shows an example of traits in action. The trait `Showable` (Line 1) requires the presence of a method **show**, and is equivalent to a Java interface. The trait `Countable` (Line 5) requires the presence of an integer field `f`, and *provides* a method `bump`, which increments `f` by one. The two traits are included by the class `Cell` (Line 12), which must provide the required fields and methods, but which also gets any methods provided by the traits; in this case the `bump` method.

```
1  trait Showable
2    require def show() : String
3  end
4
5  trait Countable
6    require var f : int
7    def bump() : unit
8      this.f += 1
9    end
10 end
11
12 class Cell : Showable + Countable
13   var f : int
14   def show() : String
15     int_to_string(this.f)
16   end
17 end
```

Figure 4. Example of traits

We extend the type checker to cater for traits, adding a subtyping relation between classes and their included traits. To distinguish between class types and trait types, the type checker needs to keep track of the declared traits of a program. We extend the environment with a new field `traittable`.

```
data Env = Env {ctable :: Map Name ClassDef
                ,traittable :: Map Name TraitDecl,
                ,vartable :: Map Name Type
                ,typeParameters :: [Type]
                ,bt :: Backtrace
                ,constructor :: Bool}
```

When type checking a class that includes traits, the type checker checks well-formedness (lines 2–3) and trait requirements (lines 4–6):

```
1  doTypecheck c@(Class {cname, cfields, cmethods, ctraits}) = do
2    local addTypeVars $ mapM_ typecheck ctraits
3    mapM_ isTraitType ctraits
4    mapM_ (meetRequiredFields cfields) ctraits
5    meetRequiredMethods cmethods ctraits
6    ensureNoMethodConflict cmethods ctraits
7    ...
```

Checking subtyping between a class type C and a trait type T is as simple as looking up the declaration of C and seeing if it includes T. We capture this in a function `subtypeOf`, of type `Type → Type → TypecheckM Bool`, that we use whenever we need to check subtyping (for example in the helper function `hasType`).

10 Feature: Uniqueness Types

In addition to standard object-oriented features, like polymorphism and subtyping, Encore supports a capability-based type system to prevent data-races [12]. Part of this system involves reasoning about uniqueness (disallowing aliasing of an object), similar to languages like Rust [30] or Pony [13]. To maintain uniqueness, the compiler performs an additional pass over the program, called *capture checking*, to ensure that unique variables are not duplicated.

The capture checker traverses the *AST* and marks each node as either *free* or *captured*. A free node is a node whose value is not being stored anywhere, whereas a captured node is one whose value is bound to some reference. For example, the expression **new** `Foo()` is considered free (it has no existing references to it), whereas a variable x is considered captured (its value is reachable through x). Whenever a value of unique type is used in a way that would capture it (*e.g.*, when it is being bound to a variable), the capture checker ensures that the value is free, so that capturing it would not introduce overlapping references.

Just as before (but ignoring phantom typed phases for clarity), we introduce a type class for capture checking:

```
1  class Capturecheckable a where
2    capturecheck :: Pushable a ⇒ a → TypecheckM a
3    capturecheck x = local (pushBT x) $ doCapturecheck x
4
5    doCapturecheck :: a → TypecheckM a
```

The capture checking pass uses the same monad as the type checker, enabling reuse of the features we have discussed in this paper. We can get error reporting with backtraces, warnings, and so on, out of the box. To annotate the *AST*, we extend each expression with a field `captureStatus` of type **`Maybe`** `CaptureStatus`, where `CaptureStatus` is a data type simply defined as either `Free` or `Captured`.

In order to be able to pass unique values around in variables, Encore uses *destructive reads* [8] of the form **`consume`** `x`, which reads `x` and sets its value to **`null`**. While reading a variable gives a captured value, reading a variable destructively results in a free value. The following excerpt shows these rules, and the rule for **`let`** expressions, which capture their bound value, and is free if and only if the body of the **`let`** is free:

```
instance Capturecheckable Expr where
  doCapturecheck e@VarAccess{} =
    return $ makeCaptured e

  doCapturecheck e@Consume{} =
    return $ makeFree e

  doCapturecheck e@Let{val, body} =
    do val' ← capturecheck val
       let ty = getType val'
       when (isUniqueType ty) $
         unless (isFree val') $
           tcError $ UniqueCaptureError val' ty
       body' ← capturecheck body
       let e' = e{val = val', body = body'}
       if isFree body then
         return $ makeFree e'
       else
         return $ makeCaptured e'
```

As this example shows, the infrastructure presented in this paper can be reused for other kinds of static analysis of an *AST*. Any future extensions made to the typechecking monad would be useful for these analyses as well.

11 Related Work

In this section relate the Encore compiler to other compilers of functional and object-oriented languages.

11.1 Comparison with Functional Languages

In this section we compare our design with the compilers of three functional languages also written in Haskell: PureScript, Elm, and Haskell.

PureScript PureScript is a strongly typed, functional programming language that compiles to JavaScript [19]. Its syntax resembles Haskell, it is *strictly* evaluated, and has an advanced type system with support for type classes, type inference, and extensions similar to Haskell's, such as `DataKinds` (*cf.* Section 7).[5]

PureScript has a bidirectional type checker [28], encoded in a monadic design with the the `Except`, `State`, and `Writer` monads. Warnings are accumulated in the `Writer` monad, and are encoded using the error data type. Errors are accumulated in the `Except` monad. This design decision means that one cannot differentiate warnings from errors, unless there is a monadic context. At the same time, it also allows an interplay between the `Writer` and `Except` monad, where one can lift a warning into an exception easily, given that they are of the same data type.

PureScript's exception handling design is similar to ours, but differs in its implementation. To handle errors, PureScript first throws exceptions and then catches them at specific points to add more information, re-throwing the error to propagate it to higher levels. Our design relies on the interplay between the `Reader` monad and the `Except` monad. Upon throwing an exception, we first get the backtrace information from the environment, create the appropriate error, and throw the exception. There is nothing preventing our design to use the *throw-catch-rethrow* design used in PureScript. In terms of throwing multiple errors, PureScript follows the same design as ours (Section 6), except that it has been encoded differently.[6]

Overall, PureScript uses the techniques presented in this paper with some design variations. We see this as an indication that our design could be used for compilers of functional languages without having a negative impact on, *e.g.*, type inference.

Elm Elm is a strongly typed, functional, reactive programming language for creating responsive, web-browser based graphical interfaces, compiling to JavaScript [15].

Elm is known for its detailed compiler messages with understandable backtraces. The type checker does not use a monadic approach, nor even a common functional style – the type checker uses mutable references to register and keep track of variables. In many occasions, some of the boilerplate code is manually dealt with. For instance, errors could be easily accumulated using the `Except` monad, but Elm has a more imperative design approach:

```
addError :: State → Error.Error → State
addError (State env rank errors) err = State env rank (err:errors)
```

[5] More information https://github.com/purescript/documentation/blob/master/language/Differences-from-Haskell.md#extensions

[6] https://github.com/purescript/purescript/blob/b7b47b236e9892675c2e7854630f1ae5e219479c/src/Language/PureScript/Errors.hs#L1505

The `State` type contains an environment and a list of errors. The type checker does not throw monadic exceptions, but accumulates errors under this environment. Warnings are dealt with in a similar fashion to errors.

We believe that parts of the compiler could be improved by the advanced functional features explained in this paper, such as the `Except` monad. For example, the compilation function (code below) uses helper functions (*e.g.,* `typeCheck`) to pattern match on the result, do nothing if it is `Right`, or wrap the error into a new data type on the `Left` value. This is a clear case for throwing exceptions.

```
compile pkg ifaces modul =
  do canonical ← canonicalize pkg ifaces modul
     annotations ← typeCheck modul canonical
     () ← nitpick canonical
     objects ← optimize modul annotations canonical
     return (Artifacts canonical annotations objects)

typeCheck :: Module → Module → Either Error (Map Name Annotation)
typeCheck modul canonical =
  case unsafePerformIO (Type.run =<< Type.constrain canonical) of
    Right annotations → Right annotations
    Left errs → Left(E.BadTypes (Localizer.fromModule modul) errs)
```

Haskell Haskell is a lazy, pure, functional language with a state-of-the-art type system [23]. It has support for type classes, type inference [34], and advanced extensions. Haskell is the leading typed, functional programming language used in industrial settings.

The Glasgow Haskell Compiler [2], which is the de-facto standard Haskell compiler, uses a type checking monad `TcM` to do type checking and renaming. The type checking monad acts like the `Reader` monad with instances for the `Except`, `Fix` and `MonadPlus` monads. (The monadic structure of the type checking monad is also used in other phases). The `TcM` monad maintains global and local environments, tracking top-level information from modules and local information, respectively. `TcM` uses mutable references to keep track of changes that should be returned after type checking, for example, errors and warnings. Warnings are encoded in terms of errors, but use a type alias to differentiate warnings from errors when required.

Type checking of expressions takes an *AST* node, and an expected type, the type checker returns the new *AST* node wrapped in the type checker monad, as per `tcExpr :: HsExpr GhcRn → ExpRhoType → TcM (HsExpr GhcTcId)`.

The main differences with our approach is that GHC's design uses the `Reader` monad and mutable references to apply type changes on specific scopes and to accumulate errors and warnings. This is in contrast to our usage of the `Reader` monad in all type checking places and the `Writer` monad to accumulate warnings. Our design, which is also simpler, does not make use of the **`IO`** monad, given that we use the `Writer` and `State` monads to accumulate errors and warnings. Finally, the monadic structure of our type checking monad can also be reused in other phases of the compiler.

11.2 Comparison with Object-Oriented Languages

In this section we compare our current design with the Scala compiler.

Scala Compiler Scala is an object-oriented language with an advanced type system, with support for functional programming [6]. It provides subtyping via inheritance, traits, and type classes.

Regarding the implementation of the compiler's AST, Scala implements a DSL to generate AST nodes. Their encoding uses classes, traits, case classes, and implicit conversions; to create flexible *AST* node transformations, their design uses inheritance and subtyping. This design is much more advanced than our simple encoding of ASTs.

Scala's design to handle errors and warnings uses inheritance and subtyping. This design decision favours the creation of new subclasses to provide different implementations, such as a storing class that accumulates errors (similar to the one in this article), to a side-effecting class that prints errors. Our implementation is more principled due to Haskell's type system, where side-effects are captured at the type-level. However, as is common in functional programming, adding a new data value implies the explicit management of this data value – this is the problem known as the Expression Problem [14, 29, 36] (to which several solutions exist, *e.g.,* [16, 17, 32, 33]).

12 Discussion and Lessons Learned

In this section we discuss the project on a higher level and list some of the things we learned in the process.

Our Background When this project started, we were a group of researchers familiar with functional programming as a whole; we had not previously developed more than toy programs in Haskell. Some of us were familiar with Haskell as a general purpose language, but not so much with its advanced typing features and abstractions, such as generalised algebraic data types, monads, or monad transformers.

We were not used to thinking in terms of algebraic structures, *e.g.,* semigroups and monoids. We were also not familiar with GHC extensions, such as functional dependencies and others, which are invaluable for writing Haskell code.

The decision to use Haskell was more or less arbitrary, but it turned out to be the right choice in terms of maintainability and flexibility of the compiler.

What Didn't Work So Well As with any new language, there is a learning curve and it took some time for new people joining the project to be familiar and productive with the language. Debugging was not easy, and we still mainly rely on prints from the `trace` function in module `Debug.Trace`.

When we started this project, we could not find explanations for how to create a compiler using Haskell's features, putting together advanced concepts in small steps, as we do here. We were able to follow the functional literature, but we had to put the pieces back together on our own.

The intuition to type state using phantom types and the kinding system is rough to grasp at the beginning, and during the implementation we felt we were guided by gut feeling, rather than deep knowledge. For this reason, we believe that an experience report like this can shed some light on how to do simple type-level programming in the context of a compiler, in Haskell.

The type state implementation was developed specifically for the type checker of the language presented in this paper, and has yet to be added to Encore. We do not think that this involves substantially more work than the changes explained in this paper.

What Can You Learn From This Experience? We hope that this experience report can serve as a solid basis for future researchers and practitioners interested in developing static tools or compilers in Haskell.

We believe that this report shines light on how to write a compiler (static analysis tool), leveraging high-level (type) abstractions. The related work on functional languages confirms that this design is also applicable to compilers of functional languages. We do not claim the originality of the techniques, but we believe that this report can still be helpful in spreading the gospel of the monadic design to developers of object-oriented languages.

Brief Comparison to the Encore Compiler This paper presents a subset of the full Encore language. In addition to the features we have seen so far, Encore supports concurrency based on actors and tasks, together with a capability-based type system for ensuring the absence of data-races [12]. There are also arrays, global and local functions, mutable and immutable variables, algebraic data types and pattern matching, and other features that naturally make the compiler more complicated. This means that the source code of the Encore compiler is more complicated than the code presented in this paper. Still, we argue that the *overall design* is the same for both compilers.

As an example of the differences between our presentation and the real thing, we include the type checking code for assignments from the Encore compiler, and discuss some of the differences from the code in Figure 2:

```
doTypecheck assign@(Assign {lhs = lhs@VarAccess{qname}, rhs}) =
  do eLhs ← typecheck lhs
     varIsMutable ← asks $ isMutableLocal qname
     varIsLocal ← asks $ isLocal qname
     unless varIsMutable $
           if varIsLocal
           then tcError $ ImmutableVariableError qname
           else pushError eLhs NonAssignableLHSError
     eRhs ← hasType rhs (AST.getType eLhs)
     return $ setType unitType assign {lhs = eLhs, rhs = eRhs}

doTypecheck assign@(Assign {lhs, rhs}) =
  do eLhs ← typecheck lhs
     unless (isLval eLhs) $
           pushError eLhs NonAssignableLHSError
     context ← asks currentExecutionContext
     case context of
       MethodContext mtd →
         unless (isConstructor mtd && isThisFieldAccess eLhs) $
           assertNotValField eLhs
       _ → assertNotValField eLhs
     eRhs ← hasType rhs (AST.getType eLhs)
     return $ setType unitType assign {lhs = eLhs, rhs = eRhs}
```

Since Encore has global variables in the form of global functions, we separate type checking of assignment into two cases. If we are assigning a variable, we ensure that it is a mutable, local variable (and raise an informative error if it is not). Note the use of `pushError`, which pushes an expression onto the backtrace before throwing an error. This is used to make an error message focus on a child of the current expression.

The case where we are not assigning a variable is mostly similar to the example in Figure 2. The main difference is that we use the backtrace to get the current execution context (a method, a function, or a closure) and check if we are currently inside a constructor, rather than storing this information directly as a flag in the environment as in Section 2.

13 Conclusion

A huge part of writing code is finding the right abstractions. Haskell proved to be the right tool for us for writing a compiler for our concurrent, object-oriented language. As shown in this report, we monotonically added compiler extensions to a core type checker, without any substantial changes to the original code.

In this report we also show how to add language-level features, such as parametric polymorphism and subtyping. We finished by showing that the monadic design can be reused for new phases doing static analysis, and that these phases immediately gets access to an environment, and the warning and exception systems, for free.

Acknowledgments

Kiko Fernandez-Reyes' work on this paper was funded by the SCADA project. Elias Castegren's work on this paper was funded by KTH Royal Institute of Technology. We are grateful to Viktor Palmkvist for pointing out the usage of `ApplicativeDo` to support multiple errors, and to the anonymous reviewer who suggested an improvement to the type state implementation. We are also grateful to Stephan Brandauer, Tobias Wrigstad, Dave Clarke, and Albert Mingkun Yang who influenced the design of the Encore compiler.

References

[1] 2019. The F# Compiler. https://github.com/fsharp/fsharp. GitHub repository.

[2] 2019. The Glasgow Haskell Compiler. https://www.haskell.org/ghc.

[3] 2019. The Java Compiler. https://openjdk.java.net/groups/compiler/.

[4] 2019. Mirror of official clang repository. https://github.com/llvm-mirror/clang.

[5] 2019. The OCaml Compiler. https://github.com/ocaml/ocaml. GitHub repository.

[6] 2019. The Scala Compiler. https://github.com/scala/scala. GitHub repository.

[7] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1015–1022. https://doi.org/10.1145/1639950.1640073

[8] Henry G. Baker. 1995. "Use-once" Variables and Linear Objects: Storage Management, Reflection and Multi-threading. *SIGPLAN Not.* 30, 1 (Jan. 1995), 45–52. https://doi.org/10.1145/199818.199860

[9] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 183–200. https://doi.org/10.1145/286936.286957

[10] Stephan Brandauer et al. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming - SFM 2015*. https://doi.org/10.1007/978-3-319-18941-3_1

[11] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An Extension of System F with Subtyping. In *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings (Lecture Notes in Computer Science)*, Takayasu Ito and Albert R. Meyer (Eds.), Vol. 526. Springer, 750–770. https://doi.org/10.1007/3-540-54415-1_73

[12] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[13] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816

[14] William R. Cook. 1990. Object-Oriented Programming Versus Abstract Data Types. In *Foundations of Object-Oriented Languages, REX School/Workshop, Proceedings (Lecture Notes in Computer Science)*, J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg (Eds.), Vol. 489. Springer, 151–178. https://doi.org/10.1007/BFb0019443

[15] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422. https://doi.org/10.1145/2491956.2462161

[16] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

[17] Erik Ernst. 2004. The expression problem, Scandinavian style. *ON MECHANISMS FOR SPECIALIZATION* (2004), 27.

[18] Matthew Fluet and Riccardo Pucella. 2006. Phantom types and subtyping. *J. Funct. Program.* 16, 6 (2006). https://doi.org/10.1017/S0956796806006046

[19] Phil Freeman. 2017. PureScript by Example.

[20] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014). https://doi.org/10.1145/2629609

[21] Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Tutorial Text*. https://doi.org/10.1007/3-540-59451-5_4

[22] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 1995*. https://doi.org/10.1145/199448.199528

[23] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(August 2019)* (2010).

[24] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's Do-notation into Applicative Operations. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. https://doi.org/10.1145/2976002.2976007

[25] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*. https://doi.org/10.1145/1159803.1159811

[26] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[27] Benjamin C Pierce and David N Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000).

[28] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

[29] JC Reynolds. 1994. User-defined types and procedural data as complementary approaches to data abstraction. Theoretical Aspects of Object-Oriented Programming.

[30] Rust Programming Language [n.d.]. The Rust Programming Language. https://www.rust-lang.org. Accessed June 2019.

[31] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *ECOOP 2003*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-45070-2_12

[32] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[33] Mads Torgersen. 2004. The Expression Problem Revisited. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference (Lecture Notes in Computer Science)*, Martin Odersky (Ed.), Vol. 3086. Springer, 123–143. https://doi.org/10.1007/978-3-540-24851-4_6

[34] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular type inference with local assumptions. *J. Funct. Program.* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

[35] Philip Wadler. 1992. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi*. https://doi.org/10.1007/978-3-662-02880-3_8

[36] Philip Wadler. 1998. The expression problem. Posted on the Java Genericity mailing list.

[37] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283