Desenvolvendo documentos para linguagem de marcação

João Peres Cassio Silva

Researchers who influenced the type checker design

- Stephan Brandauer,
- Dave Clarke,
- Albert Mingkun Yang, and
- Tobias Wrigstad

Description

This artefact contains an incremental implementation of the type checker as explained in the paper. This document contains the following:

1. information on how to build the library,
2. explanations and documentation of the main functions of the type checking library,
3. unit tests,
4. detailed information on writing your own programs and type checking them,
5. a new section that can be thought as a preliminary approach to Section 7 of the paper, which is a simplification that removes certain abstractions for the non-advanced Haskell reader, and
6. information on how to extend the type checker.

NOTE. We recommend that the reader looks at the HTML version of the `README` file, since it is better formatted and the links point to PDF sections automatically. One can find the HTML version in the downloaded artefact, inside the zip file.

Table of Contents:

    iv) Using a provisioned Virtual Machine
2. Implementation in Haskell
    i) Library code
    ii) Documentation
    iii) Unit tests
    iv) Write your own programs
    v) Phantom Phases
3. How to extend the type checker

0. Folder Structure

The folder structure of this artefact is as follows:

```
.
|--- README.html (Documentation of the artefact in HTML format)
|--- README.pdf (Documentation of the artefact in PDF format)
|--- documentation (auto-generated documention from code)
|   |--- index.html
|   |--- ...
|
|--- assets
|   |--- fonts
|   |--- pandoc.css
|   |--- submitted-version.pdf
|
|--- typechecker-oopl (Type checker)
    |--- stack.yaml
    |--- LICENSE-MIT
    |--- README.md
    |--- Setup.hs
    |--- package.yaml
    |--- src
    |   |--- Applicative (Section 6)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- Backtrace (Section 4)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- Final (Final type checker)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- Initial (Section 2)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
```

```
|   |--- MultiError (Section 6)
|   |   |--- AST.hs
|   |   |--- Typechecker.hs
|   |
|   |--- PhantomFunctors (Section 7)
|   |   |--- AST.hs
|   |   |--- Typechecker.hs
|   |
|   |--- PhantomPhases (Another approach to Section 7,
    ↪ explained later)
|   |   |--- AST.hs
|   |   |--- Typechecker.hs
|   |
|   |--- Reader (Section 3)
|   |   |--- AST.hs
|   |   |--- Typechecker.hs
|   |
|   |--- Warning (Section 5)
|       |--- AST.hs
|       |--- Typechecker.hs
|
|--- stack.yaml
|--- test
|   |--- Spec.hs
|
|--- typechecker-oopl.cabal
```

The instructions are in `README.html` and `README.pdf`. The `assets` folder contains assets for the HTML version and the submitted paper. If you are reading the HTML version, the links to the paper direct you to the appropriate page. If you are using the PDF version, the links only point to the paper.

The implementation can be found under the project folder named `typechecker-oopl`. All the code is documented and the auto-generated documentation can be found under the folder documentation.

As per the paper, the type checker is (mostly) built incrementally, following the order specified in the paper:

1. `Initial` (Paper: Section 2)
2. `Reader` (Paper: Section 3)
3. `Backtrace` (Paper: Section 4)
4. `Warning` (Paper: Section 5)
5. `Multiple Errors` or `Applicative` (Paper: Section 6)
6. `PhantomFunctors` (Paper: Section 7)
7. `Final` (The final version that aggregates all of the previous ones).

As part of an intermediate step to Section 7 (Type State), we have created a simplification of this section, which removes the abstraction over functors. This

is less elegant but an intermediate step for readers who are still learning Haskell and prefer to take a smaller step before jumping into Section 7.

1. Prerequisites

The library is written in the Haskell programming language and has the following dependencies:

- GHC 8.6.3
- `stack`

Below you can find information on how to install these dependencies in OSX, Linux and Windows. We also provide instructions on how to download a ready-to-use virtual machine.

Installing prerequisites on OSX

Type the following command to install `stack`.

```
curl -sSL https://get.haskellstack.org/ | sh
```

To install GHC 8.6.3 and all the dependencies from the project, type:

```
cd typechecker-oopl
stack build
```

If you need further assistance installing Haskell and `stack`, please click here.

Installing prerequisites on Linux

Type the following command to install `stack`.

```
sudo apt-get update
curl -sSL https://get.haskellstack.org/ | sh
```

To install GHC 8.6.3 and all the dependencies from the project, type:

```
cd typechecker-oopl
stack build
```

If you need further assistance installing Haskell and `stack`, please click here.

Installing prerequisites on Windows (non-tested)

Download the `stack` binary from here, and proceed with the normal installation process.

If you need further assistance installing Haskell and `stack`, please click here.

Using a provisioned Virtual Machine

This artefact contains a Virtual Machine (VM) named `TypeChecker.ova`.

You can use your favorite virtualisation software. We have tested this VM using Virtual Box.

After importing the VM, start and login to the VM with the following credentials:

```
user: vagrant
password: vagrant
```

Upon login, a terminal will pop up to receive you. Please type the following command to go to the artefact folder:

```
cd Desktop/TypeChecker
```

If necessary, you can also open this documentation in the VM. Locate the folder `TypeChecker` in the `Desktop`, open it and open the file `README.html`.

2. Implementation in Haskell

This section contains the following information:

- the library code,
- documentation about the library,
- unit tests,
- how to write your own programs, and
- a new section on Phantom Phases, a smaller increment before diving into Section 7.

i) Library Code

The library code can be found in the folder `typechecker-oopl` (where `typechecker-oopl` stands for type checker of an Object-Oriented Language):

```
cd typechecker-oopl
```

As per the paper, the type checker is built incrementally following (mostly) the order specified in the paper:

1. `Initial` (Paper: Section 2)
2. `Reader` (Paper: Section 3)
3. `Backtrace` (Paper: Section 4)
4. `Warning` (Paper: Section 5)
5. `Multiple Errors` or `Applicative` (Paper: Section 6)
6. `PhantomFunctors` (Paper: Section 7)
7. `Final` (The final version that aggregates all of the previous ones).

This is reflected in the structure of the type checker package. Under the `src` folder, there are folders for each of the incremental changes. (Read *Deviations from the paper* section to check for the differences between the paper and the implementation incremental ordering).

```
|--- typechecker-oopl (Type checker)
    |--- stack.yaml
    |--- LICENSE-MIT
    |--- README.md
    |--- Setup.hs
    |--- package.yaml
    |--- src
    |   |--- Applicative (Section 6)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- Backtrace (Section 4)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- Final (Final type checker)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- Initial (Section 2)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- MultiError (Section 6)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
    |   |--- PhantomFunctors (Section 7)
    |   |   |--- AST.hs
    |   |   |--- Typechecker.hs
    |   |
```

```
|   |--- PhantomPhases (Another approach to Section 7,
    ↪ explained later)
|   |   |--- AST.hs
|   |   |--- Typechecker.hs
|   |
|   |--- Reader (Section 3)
|   |   |--- AST.hs
|   |   |--- Typechecker.hs
|   |
|   |--- Warning (Section 5)
|       |--- AST.hs
|       |--- Typechecker.hs
|
|--- stack.yaml
|--- test
|   |--- Spec.hs
|
|--- typechecker-oopl.cabal
```

All the incremental implementations from the paper matches the increments highlighed on a per-section basis.[1]

All the modules, `Initial`, `Backtrace`, etc. contain two files, `AST.hs` and `Typechecker.hs`. The file `AST.hs` contains the definition of the Abstract Syntax Tree (AST) of our language. Since our focus is on type checking, we assume that a parser generates the AST as per our definition. The `Typechecker.hs` module contains the main type checking algorithm, helper functions for type checking, as well as the definition of the `TypecheckM` monad – used throughout the paper.

We recommend the reader to visit section Documentation for checking the documentation as well as the source code, from the browser. More advanced readers can use an IDE to better read the code.

Deviation from the paper We have tried to build the type checker as in the paper, in an incremental way. However, there are minor modifications. For example, the `PhantomFunctor` module (Section 7 (Type State)) is implemented without many of the previous features, such as warnings and backtraces. Since this is arguably the most advanced extension, we wanted to focus on the type state features alone. As argued in the paper though, the order in which we add the extensions is unimportant, and the `Final` module still contains all the extensions to the type checker, as explained in the paper.

Since the extensions for supporting parametric polymorphism (Section 8) and uniqueness types (Section 10) require more significant changes to the language being type checked, this has not been implemented in this artefact. A brief

[1] There are minor deviations, explained in Section: Deviation from the paper.

description of how to extend the type checker with subtyping support (Section 9) is included at the end of this document.

ii) Documentation

We have put emphasis on the code documentation, auto-generated in the folder:

```
TypeChecker/documentation
```

and the following folder in the VM:

```
/home/vagrant/Desktop/TypeChecker/documentation
```

(We refer the reader to the folder structure section in case of doubts).

The following list links pdf sections with their implementation increments:

- Global Index
- 2. A Small Object-Oriented language – Typechecker.hs and AST.hs
- 3. Refactoring: Removing Boilerplate – Typechecker.hs and AST.hs
- 4. Extension: Support for backtraces – Typechecker.hs and AST.hs
- 5. Extension: Addition of Warnings – Typechecker.hs and AST.hs
- 6. Extension: Support Multiple Errors
  - i) Approach 1. Typechecker.hs and AST.hs
  - ii) Approach 2. Typechecker.hs and AST.hs
- 7. Refactoring: Type State Phases – Typechecker.hs and AST.hs

One can access the Haskell auto-generated documentation, by clicking in the top-level `Source` button (between `Quick Jump` and `Contents`). (As an example, the reader can click on this link, which opens up the documentation for the warnings module, and try the clicking of the `Source` button). The documentation also links to its implementation on a per function basis (click the `#Source` link of any function).

iii) Unit tests

There is a single file that contains unit tests for all the increments, located in `typechecker-oopl/test/Spec.hs`. Upon execution of the unit test, the reader will observe how each increment improves the type checker.

In the terminal, enter to the folder:

```
cd typechecker-oopl
```

Inside this terminal, run the tests typing:[2]

[2] `stack test` builds the modules and run the tests.

```
stack test
```

You should see a long output, similar to:

```
-- Possible recompilation of modules

****************************************
**                                    **
** Simple compile test suite          **
**                                    **
****************************************

Welcome to the test suite.
The type checker is going to run some tests and show
the output according to new features added to the compiler.

All versions of the type checker will type check the following
program, which contains 3 errors:
1. The class ``Foo'' is not defined
2. The class ``Bar'' is not defined
3. The class variable ``x'' is not defined

class C
  val f: Foo


class D
  val g: Bar
  def m(): Int
    x

************************************************
1. Initial Version.
Showing a program with 3 errors:
- type checker only catches one error
- there is not support for backtrace

Output:

Unknown class 'Foo'

************************************************

************************************************
2. Refactoring to use the Reader monad -- no extensions.
Showing a program with 3 errors:
- type checker only catches one error
- there is not support for backtrace

```

```
Output:

Unknown class 'Foo'


***********************************************

***********************************************
3. Add backtrace information.
Showing a program with 3 errors:
- type checker only catches one error
- support for backtrace

Output:

 *** Error during typechecking ***
Unknown class 'Foo'
In type 'Foo'
In field 'val f : Foo'
In class 'C'

...
```

The reader can check how, at each increment, the type checker becomes more powerful. There are two exceptions: the refactoring of boiler plate (Section 2) and the type state of phases (Section 7). The former is a refactoring that does not add any new features; the latter can be tested by simply trying to write an AST in Haskell that does not match the expected phase, since it will be rejected statically by GHC.

iv) Write your own programs

There are two ways to play with the library:

1. Test existing programs
2. Write your own program

Before we discuss how to play with the library (write your own programs), we explain how to fire up a REPL. Then we will continue with how to test existing programs and how to write your own programs.

Starting a REPL Both of them involve running a REPL. To start a REPL, the terminal must be inside the folder `typechecker-oopl`. Then, type:

```
stack ghci
```

Once the REPL is ready, load any modules that you plan on using. For example, let's load the `Warning` module:

```
:m Warning.Typechecker Warning.AST
```

From now on, you can use all the functions defined in the module `Warning`, i.e., Warning.Typechecker and Warning.AST.

Test existing programs To test existing programs, please read on how to start a REPL. Each of the increments contain basic programs that simply define an `AST` node. For example, lets assume that the REPL has loaded the module `Warning`. Then, we can test existing programs such as `testClass1`, `testClass2`, and `testClass3` by fully qualifying them as follows:[3]

```
:m Warning.Typechecker Warning.AST
let program = Program [Warning.Typechecker.testClass1]
```

We bind a `Program` AST node `Warning.Typechecker.testClass1` to the `program` variable (the implementation of `testClass1` is defined here).

To type check `program`, type:

```
tcProgram program
```

for which we get an error (which we expected, since class `Foo` is not defined):

```
Left  *** Error during typechecking ***
Unknown class 'Foo'
In type 'Foo'
In field 'val f : Foo'
In class 'C'
```

A more complex example can be simply defined as:

```
let program = Program [testClass2, testClass1]
```

To type check, type:

```
tcProgram program
```

which throws the following error:

[3]By fully qualifying existing programs we prevent rebinding of existing programs from users, i.e., users naming their programs the same as the examples which could end up with programs mixing ASTs from different modules.

```
Left  *** Error during typechecking ***
Unknown class 'Bar'
In type 'Bar'
In field 'val g : Bar'
In class 'D'
```

However, we may want to see all the errors. To check this fact, we are going to remove existing bindings by reloading the project modules, load the `MultiError` module, and rebind the `program` variable to the `AST` of the `MultiError` module, as follows:

```
:reload
:m MultiError.Typechecker MultiError.AST
let program = Program [MultiError.Typechecker.testClass2,
    ↪ MultiError.Typechecker.testClass1]
tcProgram program
```

which throws now 3 type checking errors:

```
Left  *** Error during typechecking ***
Unknown class 'Bar'
  In type 'Bar'
  In field 'val g : Bar'
  In class 'D'
Unbound variable 'x'
  In expression x
  In method 'm'
  In class 'D'
Unknown class 'Foo'
  In type 'Foo'
  In field 'val f : Foo'
  In class 'C'
```

These examples are easy to understand. The reader can write more complex examples, while being careful of creating a valid AST node (which a parser would usually generate).

Write your own program To create your own programs, one needs to import the AST module of the increment that should be tested. There are examples of AST programs in each of the `Typechecker.hs` module that can serve as inspiration, and most if not all of them are almost the same.

Information: Convenience copy-and-paste

If the reader wants to simply copy-and-paste code from the snippets to the REPL, then use this multiline notation:

```
:{
<write your
  multiline example>
:}
```

For example, lets look at the `MultiError/AST.hs` module (here) and create a class definition:

```
:reload
:m MultiError.Typechecker MultiError.AST

-- :{ and :} are for multiline examples that
-- can be copy-pasted directly from the documentation
-- into the REPL
:{
let cls =  ClassDef {cname = "D"
                    ,fields = [FieldDef {fmod = Val, fname = "g
                        ↪ ", ftype = ClassType "Bar"}]
                    ,methods = [MethodDef {mname = "m", mparams
                        ↪  = [], mtype = IntType, mbody =
                        ↪ VarAccess Nothing "x"}]}
:}
```

To type check the program, wrap it in a `Program` AST node and call the main type checking function:

```
let program = Program [cls]
tcProgram program
```

Below we show helper functions, that the reader can copy-paste, and we build 5 examples of ASTs, where one of them also updates the compiler to throw multiple exceptions when doing binary operations, and show their equivalence in pseudo-code:

(Do not forget to check tips on how to avoid mixing ASTs)

- Helper Functions
- 1. Class with unbound variable
- 2. Class with two methods with unbound variable and unknown field errors
- 3. Updating the compiler to throw multiple errors in binary operations
- 4. Creating a new instance of a class that does not exist
- 5. Testing PhantomFunctors module

Helper Functions These helper functions are merely shorthand functions for not writing specific AST nodes. We believe these could be useful for less well-

versed Haskell developers. We write also the type signature of these functions, so that the reader can look in the according `AST` from the module at test.[4]

```haskell
-- Class Factory:
classFactory :: String -> [FieldDef] -> [MethodDef] -> ClassDef
classFactory name fields methods = ClassDef name fields methods

-- Field factory:
fieldFactory :: Mod -> Name -> Type -> FieldDef
fieldFactory modif name ty = FieldDef name ty modif

-- Method factory
methodFactory :: String -> [Param] -> Type -> Expr -> MethodDef
methodFactory name params ty body = MethodDef name params ty
    ↪ body

-- Parameter factory
paramFact :: String -> Type -> Param
paramFact name ty = Param name ty

-- Field Access
fieldAccess :: Expr -> String -> FieldAccess
fieldAccess expr name = FieldAccess Nothing expr name

-- Variable access
varAccess :: String -> VarAccess
varAccess nam = VarAccess Nothing nam

-- Access to `this`.
thisAccess :: VarAccess
thisAccess = VarAccess Nothing thisName

-- Binary operation
binaryOp :: Op -> Expr -> Expr -> BinOp
binaryOp op lhs rhs = BinOp Nothing op lhs rhs
```

The reader should not copy these functions in the REPL, since the 5 following examples copy-paste the helper functions definitions[5]

1. Class with unbound variable This example tries to perform the sum on a variable (`y`) that has not been declared:

```
class Object
  val x: Int -- immutable field
```

[4]We have used the `String` type, instead of `Name`, because the implementation is just an alias and think that it could be more helpful for the reader.

[5]The helper function definitions in some cases do not actually match the expected types, especially in the case of `PhantomFunctors` (and therefore also `Final`). However, we provide a specific example for this module.

```
  def foo(): int
    this.x == y
  end
end
```

The following functions create the corresponding AST, which we test on the e.g., `Backtrace` module (feel free to copy-paste this in the REPL):

```
:reload
:m Backtrace.Typechecker Backtrace.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 1 above:
paramsExample = []
bodyExample = binaryOp Add (fieldAccess thisAccess "x") (
    ↪ varAccess "y")
methodsExample = [methodFactory "foo" paramsExample IntType
    ↪ bodyExample]
fieldsExample = [fieldFactory Val "x" IntType]
classesExample = classFactory "Object" fieldsExample
    ↪ methodsExample
programExample1 = Program [classesExample]

-- type checking of the program
tcProgram programExample1
```

which outputs:

```
Left  *** Error during typechecking ***
Unbound variable 'y'
In expression y
In expression this.x + y
In method 'foo'
In class 'Object'
```

2. Class with two methods with unbound variable and unknown field errors This example declares a class and two methods: `foo` and `bar`. The methods try to access an unbound variable and an unknown field.

```
class Object
  def foo(): int
    this.x
  end
  def bar(): bool
    y
  end
end
```

As before, we the code below reloads the module, loads the `Backtrace` module, (re)defines the helper functions in the loaded module, and encodes the AST:

```
:reload
:m Backtrace.Typechecker Backtrace.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 2 above:
paramsExample2 = []
methodsExample21 = methodFactory "foo" paramsExample2 IntType (
    ↪ fieldAccess thisAccess "x")
methodsExample22 = methodFactory "bar" paramsExample2 IntType (
    ↪ varAccess "y")
classesExample2 = classFactory "Object" [] [methodsExample21,
    ↪ methodsExample22]
programExample2 = Program [classesExample2]

-- type checking of the program
tcProgram programExample2
```

The error is:

```
Left  *** Error during typechecking ***
Unknown field 'x'
In expression this.x
In method 'foo'
In class 'Object'
```

However, we can observe how there are actually two errors. To get the compiler to report all the errors, we can test the output of loading the `MultiError` module:

```
:reload
:m MultiError.Typechecker MultiError.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 2 above:
paramsExample2 = []
methodsExample21 = methodFactory "foo" paramsExample2 IntType (
    ↪ fieldAccess thisAccess "x")
methodsExample22 = methodFactory "bar" paramsExample2 IntType (
    ↪ varAccess "y")
classesExample2 = classFactory "Object" [] [methodsExample21,
    ↪ methodsExample22]
programExample2 = Program [classesExample2]

-- type checking of the program
tcProgram programExample2
```

which now outputs both errors:

```
Left  *** Error during typechecking ***
Unknown field 'x'
  In expression this.x
  In method 'foo'
  In class 'Object'
Unbound variable 'y'
  In expression y
  In method 'bar'
  In class 'Object'
```

3. Updating the compiler to throw multiple errors in binary operations In the current type checker, we only throw multiple exception in some cases. The reader can play with the code to extend it to places where one would expect to observe multiple errors. Let us handle throwing multiple errors when doing a binary operation, such as the following one:

```
class Object
  def foo(): int
```

```
    this.x + y
  end
end
```

We would expect two errors, one for the unbound field `this.x` and one for the unbound variable `y`. Currently, we only throw one exception even when we load the `MultiError` module (please copy-paste the code below in the REPL):

```
:reload
:m MultiError.Typechecker MultiError.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 3 above:
paramsExample3 = []
bodyExample3 = binaryOp Add (fieldAccess thisAccess "x") (
    ↪ varAccess "y")
methodsExample3 = [methodFactory "foo" paramsExample3 BoolType
    ↪ bodyExample3]
classesExample3 = classFactory "Object" [] methodsExample3
programExample3 = Program [classesExample3]

-- type checking of the program
tcProgram programExample3
```

but the output only shows one error:

```
Left  *** Error during typechecking ***
Unknown field 'x'
  In expression this.x
  In expression this.x + y
  In method 'foo'
  In class 'Object'
```

Lets update the type checking function on binary operations to handle this case. In the module `MultiError.Typechecker.hs` [here], we have the following type checking implementation for `BinOp`:

```
  doTypecheck e@(BinOp {op, lhs, rhs}) = do
    lhs' <- hasType lhs IntType
```

```
    rhs' <- hasType rhs IntType
    return $ setType IntType e{lhs = lhs'
                               ,rhs = rhs'}
```

Adding multiple errors is as simple as using the forking combinator as per the paper (Sec. 6). The type signature for `<&>` is:

```
(<&>) :: (Semigroup e, MonadError e m) => m a -> m b -> m (a, b
    ↪ )
```

Since the function `hasType` (`hasType :: Expr -> Type -> TypecheckM Expr`) returns a monad, we can just apply two monadic actions (i.e., the call to `hasType` to the left-hand side and right-hand side) and aggregate the errors as follows:

```
  doTypecheck e@(BinOp {op, lhs, rhs}) = do
    (lhs',rhs') <- hasType lhs IntType <&>
                   hasType rhs IntType
    return $ setType IntType e{lhs = lhs'
                               ,rhs = rhs'}
```

If we reload the module and re-run the example:

```
:reload
:m MultiError.Typechecker MultiError.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 3 above:
paramsExample3 = []
bodyExample3 = binaryOp Add (fieldAccess thisAccess "x") (
    ↪ varAccess "y")
methodsExample3 = [methodFactory "foo" paramsExample3 BoolType
    ↪ bodyExample3]
classesExample3 = classFactory "Object" [] methodsExample3
programExample3 = Program [classesExample3]

-- type checking of the program
tcProgram programExample3
```

the type checker now captures multiple errors even in this case:

```
Left  *** Error during typechecking ***
Unknown field 'x'
  In expression this.x
  In expression this.x + y
  In method 'foo'
  In class 'Object'
Unbound variable 'y'
  In expression y
  In expression this.x + y
  In method 'foo'
  In class 'Object'
```

4. Creating a new instance of a class that does not exist This example shows how to create a new instance of a class inside a method, which requires the use of a `Let` expression. In the surface language we do not expect developers to use `Let` expressions directly, and the parser would actually generate these bindings. The code below instantiates a non-existing class:

```
class Object
  def foo(): Object
    let x = new C
    in x
  end
end
```

The code below reloads the module, (re)defines helper functions in the loaded module, and encodes the AST of the program above.

```
:reload
:m MultiError.Typechecker MultiError.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 4 above:
paramsExample4 = []
bodyExample4 = Let Nothing "x" (New Nothing (ClassType "D")) (
    ↪ VarAccess Nothing "x")
```

```
methodsExample4 = [methodFactory "foo" paramsExample4 (
    ↪ ClassType "Object") bodyExample4]
classesExample4 = classFactory "Object" [] methodsExample4
programExample4 = Program [classesExample4]

-- type checking of the program
tcProgram programExample4
```

The expected error is:

```
Left  *** Error during typechecking ***
Unknown class 'D'
  In type 'D'
  In expression new D
  In expression let x = new D in x
  In method 'foo'
  In class 'Object'
```

The reader can fix the current issue by creating a class that contains no fields nor methods, and named `D`.

```
class D
end

class Object
  def foo(): Object
    let x = new D
    in x
  end
end
```

Below we reload the module and show the resulting AST:

```
:reload
:m MultiError.Typechecker MultiError.AST

-- Helper functions being bound to the current AST module
classFactory name fields methods = ClassDef name fields methods
fieldFactory modif name ty = FieldDef name ty modif
methodFactory name params ty body = MethodDef name params ty
    ↪ body
paramFact name ty = Param name ty
fieldAccess expr name = FieldAccess Nothing expr name
varAccess nam = VarAccess Nothing nam
thisAccess = VarAccess Nothing thisName
binaryOp op lhs rhs = BinOp Nothing op lhs rhs

-- Actual encoding of the example 4 above:
paramsExample4 = []
```

```
bodyExample4 = Let Nothing "x" (New Nothing (ClassType "D")) (
    ↪ VarAccess Nothing "x")
methodsExample4 = [methodFactory "foo" paramsExample4 (
    ↪ ClassType "Object") bodyExample4]
classesExample4 = classFactory "Object" [] methodsExample4
programExample4 = Program [classesExample4, classFactory "D" []
    ↪  []]

-- type checking of the program
tcProgram programExample4
```

The expected error is that the returned type of the method differs from the expected returned type, `Object != D`:

```
Left  *** Error during typechecking ***
Type 'D' does not match expected type 'Object'
  In method 'foo'
  In class 'Object'
```

5. Testing PhantomFunctors and Final module Testing the `PhantomFunctors` (and `Final` modules) requires a bit more work because we do no get the right algebraic data type in its expected `'Parsed` kind. Reusing example 4, which is the following program:

```
class Object
  def foo(): Object
    let x = new C
    in x
  end
end
```

We show below the encoding of the AST, which requires the extension `-XDataKinds`, and the explicit type signatures that are given to the helper functions:

```
:reload
:set -XDataKinds
:m Data.Proxy PhantomFunctors.Typechecker PhantomFunctors.AST

:{
classFactory :: Name -> [FieldDef 'Parsed] -> [MethodDef '
    ↪ Parsed] -> ClassDef 'Parsed
classFactory name fields methods = ClassDef name fields methods

fieldFactory :: Mod -> Name -> Type 'Parsed -> FieldDef 'Parsed
fieldFactory modif name ty = FieldDef name ty modif

```

```
methodFactory :: Name -> [Param 'Parsed] -> Type 'Parsed ->
    ↪ Expr 'Parsed -> MethodDef 'Parsed
methodFactory name params ty body = MethodDef name params ty
    ↪ body

paramFact :: Name -> Type 'Parsed -> Param 'Parsed
paramFact name ty = Param name ty

fieldAccess :: Expr 'Parsed -> Name -> Expr 'Parsed
fieldAccess expr name = FieldAccess Proxy expr name

varAccess :: Name -> Expr 'Parsed
varAccess nam = VarAccess Proxy nam

thisAccess :: Expr 'Parsed
thisAccess = VarAccess Proxy thisName

binaryOp :: Op -> Expr 'Parsed -> Expr 'Parsed -> Expr 'Parsed
binaryOp op lhs rhs = BinOp Proxy op lhs rhs

-- Actual encoding of the example 5 above:
paramsExample5 = []
bodyExample5 = Let Proxy "x" (New Proxy (ClassType "D")) (
    ↪ varAccess "x")
methodsExample5 = [methodFactory "foo" paramsExample5 (
    ↪ ClassType "Object") bodyExample5]
classesExample5 = classFactory "Object" [] methodsExample5
programExample5 = Program [classesExample5]
:}
tcProgram programExample5
```

which outputs:[6]

```
Left Unknown class 'D'
```

Information: how to avoid mixing ASTs Upon loading a module, such as:

```
:m MultiError.Typechecker MultiError.AST
```

one creates programs that are tied to the `MultiError.AST` nodes. It is possible to mistakenly create new definitions that refer to different `AST`s, which will only fail upon trying to mix them. We recommend to remove all existing bindings when moving to a new module, and use the up-down arrows of the keyboard (in

[6] There is no backtrace/stack trace because this module does not include the additions from the `Backtrace` module.

the REPL) to loop through entered definitions and reuse them (alternatively, one can have a file to keep these). For example:

```
:reload
:m Warning.Typechecker Warning.AST
let program = Program [testClass1]

-- User-defined function
testClass1 = ClassDef {cname = "C",fields = [FieldDef {fmod =
    ↪ Val, fname = "f", ftype = ClassType "Foo"}],methods = []}

-- add `testClass1` into the `Program` AST node.
let program = Program [testClass1]

-- Testing the function in the `Warning` module.
tcProgram program
-- some output of the errors.
-- ...


-- User would like to test output of a different
-- type checker. The reader should reload project
-- when moving to another module, which
-- removes existing bindings.
:reload
:m MultiError.Typechecker MultiError.AST

-- use up and down arrows until one finds
-- the existing definition to re-test, here
-- the `testClass1`
testClass1 = ClassDef {cname = "C",fields = [FieldDef {fmod =
    ↪ Val, fname = "f", ftype = ClassType "Foo"}],methods = []}

-- add `testClass1` into the `Program` AST node.
let program = Program [testClass1]

-- Test type checker, errors, warnigns, etc.
tcProgram program
```

v) Phantom Phases

This section can be thought as an intermediate step before Section 7. Refactoring: Type State Phases, and has been written as such. Its implementation is in `typechecker-oopl/src/PhantomPhases`, under Typechecker.hs and AST.hs

6.5. Refactoring: Phantom Phases

It is really simple to introduce bugs in the compiler. For example, can you spot the error in the following function?

```
instance Typecheckable Expr where
  typecheck e@(FunctionCall {target, args}) = do
    target' <- typecheck target
    let targetType = getType target
    unless (isArrowType targetType) $ throwError $
        ↪ NonArrowTypeError targetType
    let paramTypes = tparams targetType
        resultType = tresult targetType
    args' <- zipWithM hasType args paramTypes
    return $ setType resultType e {target = target', args =
        ↪ args'}
```

The error is in the line

```
let targetType = getType target
```

which tries to get the type of the current function from a node named `target` ↪ . However, `target` has not been type checked and does not have any typing information. Instead, the line should refer to the already type checked node, `target'`, which has been decorated with the typing information. A similar kind of bug is forgetting to decorate an expression with its type, which could cause errors in later stages of the compiler.

In this section, we propose one solution to statically avoid these kind of bugs. The main idea is to use the Haskell type system to let the `AST` nodes track whether they have been type checked or not, so that we can prevent the use of undecorated `AST` nodes where we do not expect them. This will be implemented using phantom types.[7]

A full compiler goes through multiple phases. The type checker receives an undecorated `AST` node from the parsing phase, which then gets decorated during the type checking phase. Let us therefore reify the current phase in a data type:

```
data Phase = Parsed | Checked
```

Since we are aiming to use `Parsed` and `Checked` as phantom type parameters, the data type `Phase` needs to be lifted to the kind level. This can be done with the GHC extension `DataKinds`. With the extension `KindSignatures`, we can update the `AST` data types to take a phantom type of kind `Phase`:

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

[7] Matthew Fluet and Riccardo Pucella. 2006. Phantom types and sub-typing. J. Funct. Program. 16, 6 (2006). https://doi.org/10.1017/S0956796806006046

```
  | UnitType
    deriving (Eq)

newtype Program (p :: Phase) =
  Program [ClassDef p] deriving (Show)

data ClassDef (p :: Phase) =
  ClassDef {cname   :: Name
            ,fields  :: [FieldDef p]
            ,methods :: [MethodDef p] }

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

Note that a class definition is only considered `Checked` if all its fields and classes are `Checked`, and similarly for an expression and its subexpressions. Intuitively, the type of the type checking function for an expression is now going to be `Expr 'Parsed -> Expr 'Checked`, ensuring that we do not forget to check any expressions. Similarly, we change the type of a function like `getType` to `Expr ' ↪ Checked -> Type 'Checked`, ensuring that we only ever try to get the type of a decorated `AST` node.

The entry point to the type checker must be updated, since it takes an undecorated program (with phantom type `Parsed`) and returns either an error or a decorated program (with phantom type `Checked`). An important change is that the environment used for type checking must only contain well-formed types. For example, if we use the environment to look up the type of a field, this field must not have an undefined type. Because of this, generating the environment can now fail if it finds an undefined type. Hence, we now require the use of the exception monad when building the environment:

```
tcProgram :: Program 'Parsed -> Either TCError (Program '
    ↪ Checked)
tcProgram p = do
  env <- runExcept $ runReaderT (genEnv p) (generatePreEnv p)
  runExcept $ runReaderT (doTypecheck p) env
```

Note that due to a chicken-and-egg problem, the environment `Env` 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. 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 where
  precheck :: a -> TypecheckM b
```

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:

```
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)}

genEnv :: Program 'Parsed -> TypecheckM Env
genEnv (Program classes) = do
  classEntries <- 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 classEntries}

```

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

After pre-checking, we have a well-formed environment that we can use to type check the program just as before. The `Typecheckable` type class changes into:

```
class Typecheckable a where
  typecheck :: a 'Parsed -> TypecheckM (a 'Checked)
```

Thanks to phantom types, the Haskell compiler now helps us ensure that our type checking functions indeed return `AST` nodes which have been checked, and will statically notify us about the usage of undecorated `AST` nodes when one expects them to have typing information. Once more, *the original implementation of the type checker did not change notably*, we just added phantom types to some definitions and changed how we generate environments.

3. How to extend the type checker

Here we document an overview on how to extend the type checker with subtyping, closely following the explanations of the paper (Section 9).

The first thing to consider is whether we need a new `AST` node to represent traits, which we do. Hence, we declare the `TraitDecl` node as well as their new dependencies (`Requirement` AST node):

```
data TraitDecl = Trait {
  tname :: Type,
  treqs :: [Requirement],
  tmethods :: [MethodDecl]
} deriving (Show)

data Requirement = RequiredField { rfield :: FieldDecl }
  | RequiredMethod { rmethods :: MethodDecl } deriving(Show)
```

The `TraitDecl` has a field for the name of the trait, a list of requirements expected from the trait, and method declarations. The requirements could be imposed on fields (`RequiredField`) or methods (`RequiredMethod`). Now the parser can read a trait declaration and produce a `TraitDecl` AST node.

The next thing to do is to extend the environment, so that the type checker can statically check subtyping properties between classes and traits. To do this, we extend the environment as follows (highlighted in blue in HTML):

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

After this change, we can type check a class and check that the required fields and methods of its trait are present, as per the following outline in the `doTypecheck` function:

```
doTypecheck c@(Class {cname, cfields, cmethods, ctraits}) = do
  local addTypeVars $ mapM_ typecheck ctraits
  mapM_ isTraitType ctraits

  mapM_ (meetRequiredFields cfields) ctraits
  meetRequiredMethods cmethods ctraits
  ensureNoMethodConflict cmethods ctraits
  ...
```

The following lines checks that traits are well-formed:

```
local addTypeVars $ mapM_ typecheck ctraits
mapM_ isTraitType ctraits
```

After that, we check that the requirements of the traits apply to the current class, ensuring that there are no method conflicts (e.g.). As an example, this could be written as follows.

```
mapM_ (meetRequiredFields cfields) ctraits
meetRequiredMethods cmethods ctraits
ensureNoMethodConflict cmethods ctraits
```