Take-home Exam in Advanced Programming

Deadline: Friday, November 12, 15:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2021. This document consists of 22 pages; make sure you have them all. Please read the entire preamble carefully.

The exam consists of 2 questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner. The questions each count for 50%. However, note that you must have both some non-trivial working Haskell and Erlang code to get a passing grade.

In the event of errors or ambiguities in an exam question, you are expected to state your assumptions as to the intended meaning in your report. You may ask for clarifications in the discussion forum on Absalon, but do not expect an immediate reply. If there is no time to resolve a case, you should proceed according to your chosen (documented) interpretation.

What To Hand In

To pass this exam you must hand in both a report and your source code:

- *The report* should be around 5–10 pages, not counting appendices, presenting (at least) your solutions, reflections, and assumptions, if any. The report should contain all your source code in appendices. The report must be a PDF document.
- *The source code* should be in a .ZIP file called `code.zip`, archiving one directory called `code`, and following the structure of the handout skeleton files.

Make sure that you follow the format specifications (PDF and .ZIP). If you don't, the hand in **will not be assessed** and treated as a blank hand in. The hand in is done via the Digital Exam system (`eksamen.ku.dk`).

Learning Objectives

To get a passing grade you must demonstrate that you are both able to program a solution using the techniques taught in the course *and* write up your reflections and assessments of

your own work.

- For each question your report should give an overview of your solution, **including an assessment** of how good you think your solution is and on which grounds you base your assessment. Likewise, it is important to document all *relevant* design decisions you have made.
- In your programming solutions emphasis should be on correctness, on demonstrating that your have understood the principles taught in the course, and on clear separation of concerns.
- It is important that you implement the required API, as your programs might be subjected to automated testing as part of the grading. Failure to implement the correct API may influence your grade.
- To get a passing grade, you *must* have some non-trivial working code in both Haskell and Erlang.

Exam Fraud

This is a strictly individual exam, thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course. Submitting answers (code and/or text) you have not written entirely by yourself, or *sharing your answers with others*, is considered exam fraud.

You are allowed to ask (*not answer*) how an exam question is to be interpreted on the course discussion forum on Absalon. That is, you may ask for official clarification of what constitutes a proper solution to one of the exam problems, if this seems either underspecified or inconsistently specified in the exam text. But note that this permission does not extend to discussion of any particular solution approaches or strategies, whether concrete or abstract.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. However, make sure to use proper and *specific* citations for any material from which you draw considerable inspiration – including what you may find on the Internet, such as snippets of code. Similarly, if you reuse any significant amount of code from the course assignments *that you did not develop entirely on your own*, remember to clearly identify the extent of any such code by suitable comments in the source.

Also note that it is not allowed to copy any part of the exam text (or supplementary skeleton files) and publish it on forums other than the course discussion forum (e.g., StackOverflow, IRC, exam banks, chatrooms, or suchlike), whether during or after the exam, without explicit permission of the author(s).

During the exam period, students are not allowed to answer questions on the discussion forum; *only teachers and teaching assistants are allowed to answer questions.*

Breaches of the above policy will be handled in accordance with the Faculty of Science's disciplinary procedures.

Question 1: An automatic programming system

It is a common observation when working in a typed functional language that there are often not many (non-trivially) different ways to write a specific function with a given type. This is especially true for simple polymorphic functions that don't compute anything interesting by themselves, but only serve as glue for tying actual computations together. Prototypical examples would be the return/bind functions for any particular monad, or general utility combinators, such as Haskell's standard function-composition operator, `(.) :: (b -> c) -> (a -> b) -> (a -> c)`.

Sometimes there is only a single correct way of writing such a function; in other cases, there may be multiple candidates, but the programmer can relatively quickly identify the desired one. There may even be a collection of simple tests (such as the monad laws) that can eliminate unsuitable candidates.

In this task, we will look at a rudimentary system, AutoProg, for *synthesizing* all possible Haskell expressions of a given type; weeding out the functionally incorrect ones is not a part of the task. Here are some usage examples:

```
$ autoprog "(b -> c) -> (a -> b) -> (a -> c)"
\f0 -> \f1 -> \x2 -> f0 (f1 x2)
$ autoprog  "a -> a -> (a,a)"
\x0 -> \x1 -> (x1, x1)
\x0 -> \x1 -> (x1, x0)
\x0 -> \x1 -> (x0, x1)
\x0 -> \x1 -> (x0, x0)
$ autoprog -m 5 "a -> (a -> a) -> a"  # show only the first 5 solutions
\x0 -> \f1 -> x0
\x0 -> \f1 -> f1 x0
\x0 -> \f1 -> f1 (f1 x0)
\x0 -> \f1 -> f1 (f1 (f1 x0))
\x0 -> \f1 -> f1 (f1 (f1 (f1 x0)))
```

The type may also be expressed using (parameterized) *synonyms*:

```
$ autoprog -d "type F x = x -> x" -m 5 "F (F a)"
\f0 -> f0
\f0 -> \x1 -> x1
\f0 -> \x1 -> f0 x1
\f0 -> \x1 -> f0 (f0 x1)
\f0 -> \x1 -> f0 (f0 (f0 x1))
```

Note that `F (F a)` is exactly the same type as `(a -> a) -> a -> a`, which is almost the same as in the previous example, only with the two parameters of the curried function swapped. The candidate expressions are thus very similar to before, but now we also have the extra solution `\f0 -> f0` as a more compact variant of `\f0 -> \x1 -> f0 x1` (which is also found separately).

We can also introduce `newtype`-declarations. For example, to find candidates for the unit and bind functions of the Reader monad, we can run:

```
$ autoprog -d "newtype Reader r a = Rd {runRd :: r -> a}" \
           "a -> Reader r a"
\x0 -> Rd (\x1 -> x0)
$ autoprog -d "newtype Reader r a = Rd {runRd :: r -> a}" \
           "Reader r a -> (a -> Reader r b) -> Reader r b"
\r0 -> \f1 -> Rd (\x2 -> runRd (f1 (runRd r0 x2)) x2)
```

For more complicated types, there may be multiple potential implementations. For example, to find the `fmap` function for the State monad (which is in particular also a Functor):

```
autoprog -d "newtype State s a = St {runSt :: s -> (a,s)}" \
         -m 5 "(a -> b) -> State s a -> State s b"
\f0 -> \r1 -> St (\x2 -> (f0 (fst (runSt r1 x2)), x2))
\f0 -> \r1 -> St (\x2 -> (f0 (fst (runSt r1 x2)), snd (runSt r1 x2)))
\f0 -> \r1 -> St (\x2 -> (f0 (fst (runSt r1 (snd (runSt r1 x2)))), x2))
\f0 -> \r1 -> St (\x2 -> (f0 (fst (runSt r1 x2)), snd (runSt r1 (snd (runSt r1 x2)))))
\f0 -> \r1 -> St (\x2 -> (f0 (fst (runSt r1 (snd (runSt r1 x2)))), snd (runSt r1 x2)))
```

This output is a bit hard to read, because AutoProg in its simple version uses `fst`/`snd` projections instead of a pattern-matching `let` to decompose pairs. When rewritten (manually) to use the latter form, the first three candidates above correspond to:

```
\f0 -> \r1 -> St (\x2 -> let (x3,_) = runSt r1 x2 in (f0 x3, x2))
\f0 -> \r1 -> St (\x2 -> let (x3,x4) = runSt r1 x2 in (f0 x3, x4))
\f0 -> \r1 -> St (\x2 -> let (_,x3) = runSt r1 x2 in
                         let (x4,_) = runSt r1 x3 in (f0 x4, x2))
```

And here, the "intended" solution is evidently the second one.

AutoProg also allows sequences of interdependent type-constructor declarations, as well as `data`-types, though currently only non-recursive and with only a single alternative (i.e., no `|`s).

The system is organized into four modules:

- The Parser, which parses both individual types and type-constructor declarations into the corresponding abstract syntax.
- The Resolver, which converts parsed types into a simple form containing only functions, pairs, and records, but no references to user-declared type constructors.
- The Coder, which systematically enumerates all possible expressions of a given simple type, in a suitable abstract syntax.
- The Driver (main application), which processes command-line options, invokes the three parts above, and pretty-prints the solutions as legal Haskell expressions.

The Driver is already provided, so your task is to implement the other three. The three modules are weighted roughly equally for the purpose of score calculation, but they test mastery of different learning objectives, so you will generally do better overall by collecting

some non-trivial points in each part, rather than, say, (attempting to) develop a near-perfect parser and neglecting the other two modules.

Note that the modules can be developed and tested in any order; and there are clearly identified subsets of each functionality that you are strongly advised to get working first, and then extend with more complex features as time permits.

The detailed specifications of the modules are given in the following sections.

Question 1.1: Parser

AutoProg works with a simplified subset of the full Haskell grammar of type-related constructs. The concrete grammar is shown in Figure 1.

This formal grammar is supplemented with the following stipulations:

Whitespace All tokens of the grammar may be surrounded by whitespace (spaces, tabs, and newlines) and/or comments. Some non-empty whitespace is needed for separating consecutive tokens that would otherwise run together. Comments start with `{-` and end with `-}`, like in Haskell. Unlike in Haskell, end-of-line comments (`--`) are not supported, and neither are nested comments (i.e., an AutoProg comment always ends at the first `-}`). Comments count as whitespace for the purpose of token separation.

Names AutoProg distinguishes between two kinds of names: *constructors* (*cName*) and *variables* (*vName*). Both may contain any mixture of ASCII letters, digits, apostrophes (`'`) and underscores (_). However, a constructor must *start* with an uppercase letter, while a variable must *start* with a lowercase letter. Symbolic (infix) names of either kind are not supported. Also, the AutoProg keywords (`type`, `newtype`, and `data`) cannot be used as names. Other Haskell keywords *are* allowed, but are best avoided, since using them will make any examples incompatible with the Haskell type checker.

Disambiguation Like in Haskell, nested type-constructor applications must be parenthesized; that is, `F G x` parses like `F (G) x` (where `F` is applied to two arguments and `G` to none), not like `F (G x)` (where `F` and `G` have one argument each). Type-constructor application groups tighter than the infix arrow, so that, e.g., `F x y -> z` parses like `(F x y) -> z`, not `F x (y -> z)`; and the arrow is right-associative, so `x -> y -> z` parses like `x -> (y -> z)`, not `(x -> y) -> z`.

Abstract syntax The abstract syntax corresponding to the grammar is as follows (omitting that all the datatypes derive `(Eq, Show, Read)`):

```
data PType = PTVar TVName
           | PTApp TCName [PType]

data TDecl = TDSyn TDHead PType
           | TDRcd TDHead RCName [(FName, PType)]
```

Type ::= *TVar*
| *Type* ‘->’ *Type*
| ‘(’ *Type* ‘,’ *Type* ‘)’
| *TCon* *Typez*
| ‘(’ *Type* ‘)’

Typez ::= ϵ
| *Type* *Typez*

TDeclz ::= ϵ
| *TDeclz* *TDecl*
| *TDeclz* ‘;’

TDecl ::= ‘`type`’ *TDHead* ‘=’ *Type*
| ‘`newtype`’ *TDHead* ‘=’ *RCon* ‘`{`’ *Field* ‘`::`’ *Type* ‘`}`’
| ‘`data`’ *TDHead* ‘=’ *RCon* ‘`{`’ *FDeclz* ‘`}`’

TDHead ::= *TCon* *TVarz*

TVarz ::= ϵ
| *TVar* *TVarz*

FDeclz ::= ϵ
| *FDecls*

FDecls ::= *FDecl*
| *FDecl* ‘,’ *FDecls*

FDecl ::= *Fields* ‘`::`’ *Type*

Fields ::= *Field*
| *Field* ‘,’ *Fields*

TCon ::= *cName*

TVar ::= *vName*

RCon ::= *cName*

Field ::= *vName*

Figure 1: Concrete grammar of AutoProg types and type declarations

```
type TDHead = (TCName, [TVName])

type TCName = String  -- type constructor
type RCName = String  -- record constructor
type TVName = String  -- type variable
type FName = String   -- record field
```

`PType` is the abstract-syntax datatype representing "parser types". It has an extremely simple structure, because we have stipulated that, like in Haskell, the infix notation t_1 `->` t_2 and the mixfix (t_1, t_2) are just syntactic sugar for the equivalent prefix forms `(->)` t_1 t_2 and `(,)` t_1 t_2, respectively. (Note that parenthesized symbolic identifiers are not themselves allowed in AutoProg *inputs*). This means that any parser type can be represented as either a type variable, or a type constructor applied to zero or more further types. For example, the type `F x -> (y, A)` should be parsed into the following abstract syntax:

```
PTApp "(->)" [PTApp "F" [PTVar "x"], PTApp "(,)" [PTVar "y", PTApp "A" []]]
```

The abstract syntax distinguishes only two kinds of declarations: `type`-declarations get parsed into type-synonym declarations, `TDSyn`; while `newtype`- and `data`-declarations are treated as record-type declarations, `TDRcd`. For the purpose of AutoProg, the *only* difference between `newtype` and `data` is that the former can only introduce a record type with exactly one field.

Like in Haskell, if multiple fields have the same type, they can be listed together; for example `C {f1, f2 :: a -> a}` should parse to exactly the same as `C {f1 :: a -> a, f2 :: a -> a}`.

In the Resolver, we will require that the type variables in a *TDHead*, and the field names in a record, are distinct. It is *not* the job of the Parser to enforce these restrictions.

Finally, note that AutoProg is *not* indentation-sensitive. This normally makes little difference, but for better interoperability with Haskell in both single-line and multi-line inputs, top-level AutoProg declarations may be *optionally* interspersed with semicolons, as seen in the concrete grammar for *TDeclz*. Any such semicolons are ignored for the purpose of constructing the AST for the parse result.

Parser API

The grammar actually has two top-level entry points, one for types and one for type declarations. Accordingly, the Parser module exports the following two functions (only):

```
parseStringType :: String -> EM PType
parseStringTDeclz :: String -> EM [TDecl]
```

where `type EM a = Either String a` is the AutoProg-wide monad for reporting all kinds of (user) errors. For implementing your parser, you may use either ReadP or Parsec. If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators.Parsec`); in particular you are *disallowed* to use

`Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`. As always, don't worry about providing informative syntax-error messages if you use ReadP.

In the report, explain at least how you disambiguated *Type* (for example, by showing the transformed grammar), as well as any other non-trivial design and implementation choices you made. ***

Question 1.2: Resolver

The primary task of the Resolver is to convert the parser types, which heavily involve type constructors (both user-defined and built-in) to the following grammar of *simple* types:

```
data SType = STVar TVName
           | STProd SType SType
           | STArrow SType SType
           | STRcd RCName [(FName, SType)]
```

Simple types can contain type variables as before, but now the product (tuple) and arrow (function) types have dedicated constructors. The record type can be thought of as generalized product type; it consists of a record-constructor name (used for building values of the type), as well as all the field names (used for extracting components) together with their types. For example, if the declaration list had included

```
data T a = C { x :: a, f :: a -> a }
```

then resolving the result of parsing the type `T (b,b)` should give this simple type:

```
STRcd "C" [("x", STProd (STVar "b") (STVar "b")),
           ("f", STArrow (STProd (STVar "b") (STVar "b"))
                         (STProd (STVar "b") (STVar "b")))]
```

(Assuming that the parser-type variable `"b"` corresponds to the simple-type variable with the same name.)

To resolve parser types to simple types, we need to know what all the type constructors occurring in the former correspond to in the latter. This information is kept in a *type-constructor environment*:

```
type TCEnv = [(TCName, [SType] -> EM SType)]

tce0 :: TCEnv
tce0 =
  [("(,)",
    \ts -> case ts of [t1,t2] -> return $ STProd t1 t2
                      _ -> Left "bad args for tycon (,)"),
   ("(->)",
    \ts -> case ts of [t1,t2] -> return $ STArrow t1 t2
                      _ -> Left "bad args for tycon (->)")]
```

That is, the environment is represented as a list of bindings, where each type constructor is bound to a *function* mapping a list of (already resolved) type arguments to the resulting type (or possibly an error message).

The constant `tce0` is the *initial* type-constructor environment, giving just the bindings for the built-in type constructors for product and function types. If one of these type constructors is applied to fewer or more than 2 arguments, it reports an error. This cannot actually happen with the current Parser, because it can only produce parse results with a `PTApp "(,)"` *ts* or `PTApp "(->)"` *ts* in which *ts* is a two-element list; but if we were to extend the parser to allow prefix-application of symbolic type constructors, we would be able to parse a meaningless type like `(->) x y z`, and we would want an informative error message about that from the Resolver.

Thus, an important secondary task of the Resolver is to ensure that types and type declarations are *semantically* well formed. This includes checking that type constructors are applied to the correct number of arguments, but also that that there are no undefined or multiply-defined names in various places.

Like in Haskell, all type constructors, record(-value) constructors, and field names introduced in a declaration list must be distinct. However, since type constructors and record constructors live in different namespaces, they are allowed to share the same (capitalized) name, regardless of whether they are related or not; for example `newtype T a = T {t :: a}` is fine, because the first `T` is used for constructing types, and the second for values. Similarly, because type variables and field names have non-overlapping uses, this declaration could have been equivalently (if somewhat confusingly) written as `newtype T t = T {t :: t}`.

All type parameters in a single declaration head must be distinct variables. For example, `type T a b a = b` is semantically illegal, even though the multiply introduced parameter `a` is not used on the RHS of the `=`, so there is no actual ambiguity. Further, since field names can be used as selection/projection functions in programs, they must be unique across *all* record definitions, not just each one separately. Moreover, we stipulate that, to avoid confusion with projections from pairs, the names `fst` and `snd` must not be used for record-field names (but they are OK for type variables, i.e., they are not actual keywords).

Unlike in Haskell, we require that all declarations are non-recursive, and in fact may refer only to strictly previous declarations in the sequence, regardless of whether the reference introduces an actual circularity. Thus, `type T a = a -> U a; type U x = x` is illegal, even if `T` is never used.

Finally, in a correct type-constructor *declaration* (whether `type` or `newtype`/`data`), all type variables occurring on the RHS of the `=` must have been explicitly introduced on the LHS. Thus, a declaration like `type T a = b -> a`, or `type T = a -> a` is illegal, regardless of how (or whether) it's subsequently used. On the other hand, when we search for expressions of a given (polymorphic) type, such as `a -> b -> a`, that type will usually contain unbound type variables.

Resolver API

Your Resolver implementation should export two functions:

```
resolve :: TCEnv -> (TVName -> EM SType) -> PType -> EM SType

declare :: [TDecl] -> EM TCEnv
```

In `resolve` *tce tve pt*, the argument *tce* is a type-constructor environment *tce* (represented as an association list, as discussed above), *tve* is a type-variable environment (represented as a function), and *pt* is the parser type to resolve. The function returns either the resulting simple type or an error message.

The environment *tce* can be just `tce0`, but will usually have been extended with further bindings arising from type declarations. *tve* gives the bindings for any type variables that may occur in *pt*. If `resolve` is being used to prepare a goal type for the Coder, all type variables represent just themselves; but when `resolve` is used on the RHS of a type declaration (such as in the implementation of `declare`), the type variables will be bound to specific simple types, obtained from the arguments that the type constructor was applied to.

Conversely, `declare` *tds* builds a type-constructor environment from a list of type declarations *tds*, or gives an error message if there is problem with one or more of those declarations. (If there is more than one error in a single declaration, it doesn't matter which one gets reported; but later declarations should not even be looked at if there's an error in an earlier one.)

The resulting environment should extend `tce0` with bindings for all type constructors declared in the collection. (The relative order the bindings in the returned environment does not matter, since multiple bindings of the same type constructor are not allowed.)

For example, if `td` is the result of parsing the declaration `data T a = C {x :: a, f :: a -> a}` from above, i.e.,

```
td = TDRcd ("T", ["a"]) "C" [("x", PTVar "a"),
                             ("f", PTApp "(->)" [PTVar "a", PTVar "a"])]
```

then `declare [td]` should successfully return a *tce*, such that `lookup "T"` *tce* finds a function *fT*, and for all simple types *t*, the application *fT* `[`*t*`]` returns

```
STRcd "C" [("x", t), ("f", STArrow t t)]
```

On the other hand, applying *fT* to a non-singleton list should return a suitable error message, for example `Left "Bad args for tycon T"`.

Remember that the declarations should be checked individually. For example processing the declaration sequence

```
newtype T a = C1 {f :: a}; data U b = C2 {x :: b, f :: b -> b}
```

should give an error for `U` because the field name `f` has already used been used for `C1`, which is inherently problematic, regardless of whether we use the type constructor `U` for anything later. A similar problem would occur if we had reused the record constructor `C1` in the declaration of `U`.

More subtly, in the sequence

```
type T a = a; type U x = T x x
```

the declaration of `U` will only cause an actual problem if `U` is ever used (directly or indirectly) in a goal type. However, since Haskell is a statically type-checked language, we want to complain about `U` in any case, because it contains a latent bug that will be triggered if we ever apply it to anything.

Be explicit about separation of concerns in `declare`, for example by using a monad (with a suitable combination of reader/write/state/error components) to organize the code. Briefly explain your *overall* design choices for this function in the report. ***

Question 1.3: Coder

The final component, the Coder, is responsible for enumerating the expressions of a given simple type. Possible expressions are taken from the following informal grammar of a suitable Haskell subset:

$$\begin{aligned} \mathit{Exp} &::= \mathit{Var} \mid \mathit{Exp}\ \mathit{Exp} \mid \texttt{\textbackslash}\ \mathit{Var}\ \texttt{->}\ \mathit{Exp} \mid \texttt{(}\mathit{Exp}\texttt{,}\ \mathit{Exp}\texttt{)} \mid \texttt{fst}\ \mathit{Exp} \mid \texttt{snd}\ \mathit{Exp} \\ &\quad \mid \mathit{RCName}\ \texttt{\{}\mathit{Field}\ \texttt{=}\ \mathit{Exp}\texttt{,}\ \ldots\texttt{,}\ \mathit{Field}\ \texttt{=}\ \mathit{Exp}\texttt{\}} \mid \mathit{Field}\ \mathit{Exp} \\ \mathit{Var} &::= \mathit{vName} \end{aligned}$$

In practice, the Coder will simply produce expressions in abstract syntax; a suitable pretty-printer is already provided in the Driver module. (Note that, for compactness, this printer omits the braces and field names in record constructors with less than two fields.) The abstract syntax for expressions is as follows:

```
data Exp = Var VName
         | App Exp Exp
         | Lam VName Exp
         | Pair Exp Exp
         | Fst Exp
         | Snd Exp
         | RCons RCName [(FName, Exp)]
         | RSel FName Exp
```

In the following presentation, however, we will generally use the concrete syntax for expressions, as it's much more compact for non-trivial examples.

While the search space for well typed expressions is enormous, we can cut it down very considerably by only considering *irreducible* expressions, i.e., those that do not redundantly construct an expression of a more complicated type, and then immediately take it apart again, possibly throwing large parts away. Specifically, we don't need to consider expressions that anywhere contain one or more of the following patterns:

- `fst (`e_1`,`e_2`)` or `snd (`e_1`,`e_2`)`
- `(\`x `->` e_1`)` e_2
- f_i C`{`f_1 `=` e_1`,` `...,` f_n `=` e_n`}`

There are a few other redundancies possible, such as `(fst p, snd p)` (which is equivalent to just `p`), or `\x -> f x` (which is equivalent to `f`), but we will not be concerned about those, because they do not significantly increase the numbers of possible well typed expressions.

In general, when generating expressions of a target type, we may have already introduced some lambda-bound variables of other types, which we may use in the expression. Thus, in general, to produce an expression of type t, we may either *build* it out of expressions of strictly simpler types than t, or *extract* it from an available variable of a more complex type t' that contains t (or is just t itself).

In more detail, the possibilities for building an expression e of type t are:

- If $t = (t_1, t_2)$ for some t_1 and t_2 (i.e., a product type), we can take $e = (e_1, e_2)$, where e_1 is an expression of type t_1, and e_2 is an expression of type t_2.
- If $t = t_1 \texttt{->} t_2$, we can take $e = \texttt{\textbackslash}x \texttt{ -> } e_2$, where e_2 is an expression of type t_2, but which may also make use of the (freshly named) variable x of type t_1.
- If $t = C\,\{f_1 :: t_1, \ \ldots, \ f_n :: t_n\}$, we can take $e = C\,\{f_1 = e_1, \ \ldots, \ f_n = e_n\}$, where each e_i is an expression of type t_i.
- If $t = a$, where a is a type variable, *building* an expression of this type is not possible. (But we may be able to *extract* it from one of previously introduced variables.)

To *extract* an expression e of type t from an expression (possibly just a variable) e' of type t', we consider the possible forms of t':

- If $t' = t$ then we can simply take $e = e'$.
- If $t' = (t'_1, t'_2)$, we can try to extract the desired e from `fst` e', which has the simpler type t'_1; or we can try to extract it from `snd` e', which has type t'_2.
- If $t' = t'_1 \texttt{->} t'_2$, we can try to extract e from $e'\ e_1$ of the simpler type t'_2, where e_1 is an expression of type t'_1. (Note that the choice of e_1 does not affect t'_2, so we don't have to pick a specific e_1 before we have checked that it is in fact possible to extract a t-typed expression from a t'_2-typed one.)
- If $t' = C\,\{f_1 :: t'_1, \ \ldots, \ f_n :: t'_n\}$, we can pick any i between 1 and n and try to extract e from $f_i\ e'$ of type t'_i, which again is simpler than t'.
- If $t' = a$, where a a type variable different from t (the case $t' = t$ was already covered above), there's no way extract a t-typed e from e'.

In several places in the above description, we have to make a choice, most notably whether to build or extract (and in the latter case, which variable to extract from). Moreover, when extracting, we may need to keep making choices; for example, there are two distinct ways to extract an expression of type `a` from an available variable `p` of type `((a,b),a)`, namely `fst (fst p)` or `snd p`. And, further, for extracting an expression of type `c` from an `f` of type `a -> c`, we can use either one of those possibilities for the function argument, i.e, `f (fst (fst p))` or `f (snd p)`. (On the other hand, it's hopeless to extract a `c`-typed expression from a variable `g` of type `a -> d`, so we don't need to consider all the potential `a`-typed function arguments in that case.)

Each choice in the above may lead to zero or more distinct final solutions. To keep track of the choices and explore them in a systematic way, we organize them into a *search tree*. We use the following general, polymorphic type of trees:

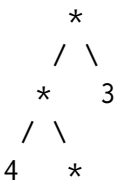
```
data Tree a = Found a
            | Choice [Tree a]
```

That is, a tree is either a leaf node containing a successful solution of type `a`, or a choice node with a number (possibly zero) of subtrees. A choice node with zero children is also called a *failure* node. The list in each `Choice` will always be finite, but the tree may still be infinitely *deep*, because it will only be constructed on demand, exploiting Haskell's lazy evaluation. An infinite-depth tree may contain finitely (possibly zero) or infinitely many solution nodes.

For example, a simple (finite) value of type `Tree Int` would be

```
tr0 = Choice [Choice [Found 4, Choice []], Found 3]
```

This contains two success nodes and three choice nodes, of which one is a failure node. It can be graphically represented as follows:



(where each `*` represents a choice node). The Coder will construct a search tree for expressions of a desired type, with a success node for each correct solution, and a choice node whenever there's a finitary choice to be made, such as which variable to extract from, or whether to extract from the first or second component of a product. When no meaningful choices are possible, such as when attempting to extract from a type variable different from the desired type, the tree will contain a failure node.

Once we have the search tree, we can enumerate all the solution nodes in it as a (potentially infinite) list. To make sure we consider all possible choices, and in particular to avoid getting stuck in infinite subtrees (whether or not they contain any solutions), we explore the trees in a *breadth*-first manner: first the root node, then all children (if any) of the root node, from left to right, then the grandchildren (again in sequence), and so forth. For example in the sample tree `tr0` above, we would first find the solution 3, and then 4. (In fact, the integers in the tree happen to exactly match the number of the corresponding node in the breadth-first traversal). In the example, there are no further nodes to consider after the fifth (which happens to be a failure), but in general, the search may go on forever. To avoid infinite recursion, we will usually impose a limit on the total number of nodes to visit in the search, and/or a limit on the number of successes we want to see.

Coder API

The module `Coder` should export the following:

```
instance Monad Tree

pick :: [a] -> Tree a
solutions :: Tree a -> Int -> Maybe a -> [a]

produce :: [(VName, SType)] -> SType -> Tree Exp
```

That is, you should provide the `return`/`>>=` functions to make `Tree` a monad of nondeterministic computations, similarly to the List monad, but keeping the individual choice points explicit in the result. An effect-free computation contains just a success node, while *tr* `>>=` *f* should return the tree resulting from replacing each success node in the tree *tr* with the tree resulting from applying *f* to the contents of that node.

The only effectful operation in the monad is `pick`, which represents a single choice from some finite list of possibilities. `pick` should avoid creating trees with singleton `Choice` nodes.

`solutions` *tr n d* returns a lazy list of (the contents of) all the success nodes in *tr*, when traversed breadth-first. The parameter *n* (which you may assume to be non-negative) represents a limit on the total number of nodes in *tr* to visit. If this limit is hit (i.e., if there are still unexplored nodes left after the first *n*), and *d* is `Just` *a*, then *a* is added as a final element in the output list, so that the caller can see that the search was cut short. (If *d* is `Nothing`, the result list will just contain the solutions from the first *n* nodes.)

In the report, *briefly* explain how you expressed the breadth-first tree search, and how you implemented the node-limit check. ***

Finally, `produce` *g t* constructs a search tree for expressions of type *t*, possibly containing variables from *g* with their associated types. The success nodes in the tree should contain precisely the irreducible expressions satisfying the requirements, and no expression should occur more than once in the tree.

The bound variables in any constructed `Lam`-expressions should not shadow each other, or those in *g*. For example, you should never produce an expression like `\x -> \x -> x`, but only, e.g., `\x -> \x' -> x'` or `\x0 -> \x1 -> x1`. Independent uses of the same name in different scopes, e.g., `\f -> f (\x -> x) (\x -> x)` are fine, though. For simplicity, you need not ensure that variables do not clash with record field names.

Exactly how to name the variables (as long as they are lexically correct *vNames*) is not specified. (The examples in the introduction use names roughly derived from their types, but that's not a requirement.) Note that expressions differing only by a consistent renaming of bound variables are considered the same, and should only be found once each.

It is also not specified in which order the choices should be made, or how the alternatives within a single choice are ordered. In particular, there is no expectation that you produce the solutions in the same order as in the examples in the introduction.

Your tree should *preferably* not contain patently futile subtrees. For example, if the goal type is `a->(a->a)->b`, any possible expression would necessarily have the shape `\x -> \f -> ?`, where the function body `?` is of type `b`; but it is clearly impossible to extract such an expression from either `x::a` or `f::a->a`, and in particular, it is not productive to enumerate the infinitely many possible `a`-typed arguments to `f`. On the other hand, for `(a->a)->a`, there might a priori be a solution of the form `\f -> f ?`, where `?` was itself of type `a`, as in

`\f -> f (f ?)`, and so on. To detect such cycles (which can be much subtler than in this example) and prune them from the search tree *not* a part of the task; it's perfectly fine to just construct an infinite tree in this case, and let the solution search hit the limit.

Hint: Unless you have a better idea, we recommend that you define `produce` mutually recursively with a companion function,

```
extract :: [(VName, SType)] -> SType -> SType -> Tree (Exp -> Exp)
```

Its specification is that `extract` g t' t constructs a search tree for extracting a t-typed result from a t'-typed expression. The success nodes in this tree should be *access paths*, detailing which sequence of projections and/or applications should be applied to the input expression. For example if t' is `((a,b),a)`, and t is `a`, then the resulting tree should contain exactly two success nodes, with Haskell functions (equivalent to) `\e' -> Fst (Fst e')` and `\e' -> Snd e'`. `extract` also gets a copy of all the available variables g. This is so that, e.g., when t' is `(a,a) -> b` and t is `b`, and g contains a variable `"x"` with type `a`, we can construct access paths like `\e' -> App e' (Pair (Var "x") (Var "x"))`. Neither `produce` nor `extract` should use `solutions` (or the `Tree` constructors), only the monad operations and `pick`.

In the report, *briefly* explain the structure of your `generate`/`extract` function, focusing on how you dealt with arrow types in both. (If you don't use `extract`, you should of course explain how your alternative solution works.) ***

How to get started

It is ultimately your decision which parts of the task to work on, and in what order. However, we recommend the following multi-pass strategy for maximizing your score-for-effort and coverage of the learning objectives:

- For the Parser, get `parseStringType` to (mostly) work, and test it (by adding relevant cases to the test suite). If there are parts or aspects that you can't easily get to work exactly right, such as the finer details of comments, or correct precedences/associativities of unparenthesized type phrases, make a note to fix it later (and a corresponding failing test case to remind you), and move on.
- For the Builder, implement and test `resolve`, as well as a stub implementation `mkTCEnv` for the special case of an empty declaration list.
- For the Coder, implement and test the `Monad` instance for `Tree`, `pick`, and `solutions`. Don't sweat any off-by-one issues with the node limit, but do make a note to check them later.

Now you have a workable base solution to expand on, possibly with some known deficiencies. For each of the modules, you can then add more features and/or bug fixes, in roughly the following order for each (but in a breadth-first way, i.e. alternating between working on the various subtasks, prioritizing the ones where you can make most progress, rather than, e.g., trying to completely finish the parser first). Add relevant test cases to the suite as you go.

- For the Parser, add `parseStringTDeclz`; implement parsing of first `type`-declarations, then `newtype`, and finally `data`.
- For the Resolver, first implement `declare` for `TDSyn`, then also for `TDRcd`. Prioritize producing correct results for well formed inputs over detecting and reporting problems for incorrect ones (but don't forget completely about the latter).
- For the Coder, first implement `produce` and `extract` for types involving only type variables and arrows; then add products, and finally records. Initially, don't worry about keeping down the size of the generated trees.

General instructions

All your code should be put into the provided skeleton files under `code/autoprog/`, as indicated. In particular, the actual code for each module `Mod` should go into `src/ModImpl.hs`. Do not change any of the other files, especially `Defs.hs`, which contains the common type definitions presented above; nor should you modify the type signatures of the exported functions in the APIs. Doing so will likely break our automated tests, which may affect your grade. Be sure that your codebase builds against the provided `app/Driver.hs` with `stack build`; if any parts of your code contain syntax or type errors, be sure to comment them out for the submission. Note that `stack.yaml` specifies the LTS-18.13 version of the Haskell platform (including GHC 8.10.7) for the exam, to hopefully avoid the stray dependency problems on Windows.

Place your main tests in `tests/BlackBox.hs`, so that they are runnable with `stack test`. As the name suggests, these should be black-box tests only, i.e., not depend on any internals of your own implementation, but test only against the API specifications given above. In particular, we may run your black-box tests against our own sample implementations (correct and/or incorrect). If you want to directly test some of your non-exported auxiliary functions, and/or unspecified behaviors, you can put those in `tests/suite1/WhiteBox.hs` (and uncomment the suite in `package.yaml`).

In your report, you should describe your main design and implementation choices for each module in a separate (sub)section. Be sure to cover *at least* the specific points for each module asked about in the text above (and emphasized with *** in the margin), as well as anything else you consider significant. Low-level, technical details about the code should normally be given as comments in the source itself.

For the assessment, we recommend that you use the same (sub)headings as in the weekly Haskell assignments (Completeness, Correctness, Efficiency, Robustness, Maintainability, Other). Feel free to skip aspects about which you have nothing significant to say. You may assess each module separately, or make a joint assessment in which you assess each aspect for all modules (but then be clear about which module you are referring to, when discussing any specific issues).

Question 2: A Cache of Fun

The task in this question is to implement a memory-based cache called Frappe. Frappe is based on *cost-based capacity*, thus inserting a heavy item in the cache can displace many lightweight items.

General comments

This question consists of two sub-questions: Question 2.1 about implementing an API for starting a Frappe server and for manipulating the cache, and Question 2.2 about writing QuickCheck tests against this API. Question 2.1 counts for 70% and Question 2.2 counts for 30% of this question. Note that Question 2.2 can be solved with a simple partial (or even without an) implementation of the `frappe` module from Question 2.1. In Appendix A you can find examples on how to use the API.

Remember that it is possible to make a partial implementation of the API that does not support all features. If there are functions or features that you don't implement, then make them return the atom `not_implemented`.

There is a section at the end of this question, on page 21, that lists expected topics for your report.

Terminology

A *cache server*, or just *cache*, maintains a *key*/*value* store, that is an association of *keys* to *values*. We call the association of a key to a value an *item*. A cache has a *capacity* which is a positive integer (meaning strictly larger than zero). Each item in the cache has a *cost*, which is also a positive integer. The *capacity invariant* is that the sum of the cost of all items in the cache is smaller than the capacity.

In this question we use a cost-based variant of the *least recently used* (LRU) algorithm for cache replacement. That is, when we add a new item to the cache we might have to remove some items from the cache to make room for the new item (to maintain the capacity invariant). Thus, to insert a new item that has a cost that would take us above the capacity of the cache, we'll remove least recently used items until we have room for the new item. See Appendix A for some example of how the cache is expected to behave. Successful read and write operations are considered a *usage of a key*.

We can apply a *transformation* to the value of a key in the cache, or for a key that we want in the cache. Keys and values can be any Erlang term, thus we use the type `transformation()` for transformation functions:

```
-type key() :: term().
-type value() :: term().
-type cost() :: pos_integer().
-type transformation() :: fun(({existing, value()} | new) ->
                                {new_value, value(), cost()} | any()).
```

See the API description of the `upsert/3` function for an explanation of the arguments and return type of `transformation()`. Your library must be robust against erroneous transformations. In particular, your solution should be able to deal with slow or non-terminating transformation functions. You may assume that it is safe to exit a process that is executing a transformation function, as long as you use an exit reason different from `kill`.

The cache should be *key coherent*, meaning that two *write operations* to the *same* key in the cache, should be done in sequential order as they are received by the cache and should only be observable in the order that they are completed.

The cache should be *key concurrent*, meaning that a write to one key should not block that another key can be written.

Question 2.1: The `frappe` module

Implement an Erlang module `frappe` with the following API.

- `fresh(Cap)` for starting a Frappe server with capacity `Cap`. Where `Cap` should be a positive integer. Returns `{ok, FS}` on success, or `{error, Reason}` if some fault occurred.
- `set(FS, Key, Value, C)` unconditionally saves `Value` under `Key` with cost `C`. This is a write operation. Returns `ok` on success; or `{error, Reason}` if `C` is an invalid cost with respect to `FS`.
- `read(FS, Key)` gets the value saved for `Key`. Returns `{ok, Val}` if a value can be found for `Key`; otherwise returns `nothing`. Note that this function should return the value of the last *completed* write operation for the key, and should not wait for ongoing write operations. This function count as a usage of the key `Key`.
- `insert(FS, Key, Value, C)` saves `Value` under `Key` with cost `C`, if there is no value associated with `Key`. This is a write operation. Returns `ok` on success; or `{error, Reason}` if `C` is an invalid cost with respect to `FS` or there already is an association for the key.
- `update(FS, Key, Value, C)` saves `Value` under `Key` with cost `C`, if there is already is a value associated with `Key`. This is a write operation. Returns `ok` on success; or `{error, Reason}` if `C` is an invalid cost with respect to `FS` or there is not an association for the key.
- `upsert(FS, Key, Fun)` for read-and-modify a cached item or for inserting a new item. This is a write operation.

  The argument for the transformation function `Fun` is:

  - `{existing, Value}` if `Value` is associated with `Key`;
  - `new` if there is no value associated with `Key`.

  The transformation function `Fun` should either:

  - return the term `{new_value, Val, C}` for associating the value `Val` with cost `C` for `Key`;
  - return a value `X` different from `{new_value, _, _}`, in which case the existing cache item (if any) remains unchanged;
  - throw the exception `{new_value, Val, C}` for associating the value `Val` with cost `C` for `Key`;
  - throw an exception different from `{new_value, _, _}`, exit or provoke an error, in which case the existing cache item (if any) remains unchanged;

  `upsert/3` returns `ok` if an item is successfully inserted or updated; or `{error, Reason}` if an item with invalid cost with respect to `FS` is attempted to be inserted or updated.

- `stable(FS, Key, Ref)` gets the value associated with `Key` asynchronously. When there is a value, `Val`, associated for `Key` and no ongoing write operation for that key, send the message `{Ref, Val}` to the calling process. When the message is sent, this counts as a usage of the key `Key`.

- `all_items(FS)` returns all the items at the server. Returns a `Items`, where `Items` is list of triples `{K, V, C}` where each triple is the key, `K`, value, `V`, and cost, `C`, of an item in the cache. Note that this function should return the value of the last *completed* write operations for the keys, and should not wait for ongoing write operations. This function does not change the order of usage of the keys.

- `stop(FS)` stops a Frappe server, including all (if any) helper processes. Returns `ok` when all processes are stopped.

Note that the write operation `upsert/3` can make the write operations `insert/4` and `update/4` block indefinitely (when writing to the same key), because they depend on the result of the transformation given to `upsert/3`. However, `set/4` does not depend on the result of `upsert/3` and should therefore *not* block, instead it should cancel any ongoing (and pending) write operations and just write the value.

How to get started

Start by implementing the `frappe` API except for the functions `upsert/3` and `stable/3`. This subset of the API will also allow you to write a meaningful (but limited) version of the properties in Question 2.2. Then extend your implementation with the `upsert` function, to start with perhaps without support for concurrent write of different keys. Save the function `stable/3` for last.

As the keys in a LRU cache are removed in an somewhat *first in, first out* manner, you might find the standard module `queue` useful. (However, note that it is not a requirement to use the `queue` module.)

Consider developing your test-suite along side your implementation. This will help you greatly, should you see the need for refactoring your code.

Question 2.2: Testing `frappe`

Make a module `test_frappe` that uses `eqc` QuickCheck for testing a `frappe` module. We evaluate your tests with various versions of the `frappe` module that contains different planted bugs and checks that your tests find the planted bugs. Thus, your tests should only rely on the API described in the exam text.

Your `test_frappe` module should contain, at least, the following:

- `mktrans(Opr, Args)` a helper function that generates an terminating transformation function from the atom `Opr` and the arguments `Args`. You decide which atoms are valid for `Opr` and what `Args` should be. We will not call this function directly, only through symbolic calls generated by your generators.
- A parameterised QuickCheck generator `terminating_transformation(KeyGen)` that takes a QuickCheck generator, `KeyGen`, as argument and then generates a tuple where the first element is generated by `KeyGen` and the second element is a symbolic call which can be evaluated to a transformation function.

  For instance, six samples from this generator could be:

  ```
  {key1, {call,test_frappe,mktrans,[add,5]}}
  {fresh, {call,test_frappe,mktrans,[set_cost, 420]}}
  {56, {call,test_frappe,mktrans,[fact_of,dummy]}}
  {"cut grass", {call,test_frappe,mktrans,[mult,10]}}
  {key1, {call,test_frappe,mktrans,[read_key,fresh]}}
  {key2, {call,test_frappe,mktrans,[both,[[set_cost,mult],[13,5]]}}}
  ```

  Again, remember that the arguments to `mktrans/2` are up to you to decide. The given samples are just for inspiration.

  Likewise, an example usage of this function could be:

  ```
  terminating_transformation(eqc_gen:nat())
  ```

  if you want to use small natural numbers as keys.

- A QuickCheck property `prop_cache_under_capacity/0` that checks that a cache behaves like a normal key/value store as long as we are under the capacity of the cache.

  If you have not implemented the full API from Question 2.1, then you may also make a version of this property called `prop_cache_under_capacity_limited/0`, that only exercise the subset of the API you have implemented. Then we'll check both the limited and the full version with our implementations of Frappe.

- A QuickCheck property `prop_capacity_invariant/0` that checks that all sequences of calls of API functions from Question 2.1 will maintain the capacity invariant.

  If you have not implemented the full API from Question 2.1, then you may also make a version of this property called `prop_capacity_invariant_limited/0`, that only exercise the subset of the API you have implemented. Then we'll check both the limited and the full version with our implementations of Frappe.

- A `test_all/0` function that runs all your tests and only depends on the specified `frappe` API. Your tests should involve the required properties in this module, but should also test aspects and functionality not covered by the required properties.
- We also evaluate your tests on your own implementation, for that you should export a `test_everything/0` function (that could just call `test_all/0`).

You may want to put your tests in multiple files (especially if you use both `eqc` and `eunit` as they both define a `?LET` macro, for instance). If you use multiple files, they must all start with the prefix `test_` (or `apqc_` if you use that). Remember that the code called from `test_all/0` must only depend on the specified `frappe` API and your (if any) `test_` modules.

You are welcome (even encouraged) to make more QuickCheck properties than those explicitly required. Properties that only depend on the specified `frappe` API should start with the prefix `prop_`. If you have properties that are specific to *your* implementation of the `frappe` library (perhaps they are related to an extended API or you are testing sub-modules of your implementation), they should start with the prefix `myprop_`, so that we know that these properties most likely only work with your implementation of `frappe`.

Topics for your report for Question 2

Your report should document:

- For Question 2.1:
  - Clearly explain if you support the complete API or not. If not, which parts you don't support.
  - If you have made any assumptions, especially simplifying assumptions. In particular if you have made any assumptions about transformation functions.
  - How many processes your solution uses, what role each process has, and how they communicate.
  - What data your processes maintain.
  - Overall, the interesting parts of Question 2.1 are the interactions of the write operations with each other and with the function `stable/3`. You should make sure that your report cover these parts.
- For Question 2.2:
  - Clearly explain if you have implemented all required parts or not. If not, which parts you don't support.
  - Explain the idea behind and how your properties works.
  - The quality or limitations of your testing. Especially the QuickCheck parts explicitly required in Question 2.2. Explain how you have measured this quality.
  - Summarise how you have tested aspects and functionality not covered by the required properties in Question 2.2.

    Likewise, as always, remember to include your tests and the result of running your test in your report (perhaps in an appendix).

Appendix A: Example use of `frappe`

The following examples demonstrates how to use the `frappe` API.

Appendix A.1: Use all capacity

The function `small_capacity` makes a cache with capacity 5. Then the keys `key1` and `key2` are inserted, after that `key1` is looked up. Thus, at this point in the program the items in the cache use 4 out of the capacity of 5, and `key1` is the most recent used key. Finally, the key `big` is inserted, and to make room for that, the item with `key2` is removed from the cache.

```
small_capacity() ->
  {ok, FS} = frappe:fresh(5),
  ok = frappe:set(FS, key1, 42, 2),
  io:format("When looking up ~w I got ~w, and I expected {ok, 42}~n",
            [key1, frappe:read(FS, key1)]),
  ok = frappe:set(FS, key2, 69, 2),
  {ok, 42} = frappe:read(FS, key1),
  ok = frappe:set(FS, big, 300, 3),
  io:format("When looking up ~w I got ~w, and I expected {ok, 42}~n",
            [key1, frappe:read(FS, key1)]),
  io:format("When looking up ~w I got ~w, and I expected nothing~n",
            [key2, frappe:read(FS, key2)]),
  frappe:stop(FS).
```

Appendix A.2: Using `upsert`

The function `small_upsert` makes a cache with capacity 5. Then the keys `key1` and `key2` are inserted. Finally, the value and cost of the item for `key2` is updated with `upsert/3`, to make room for the updated item, the item with `key1` is removed from the cache.

```
small_upsert() ->
  {ok, FS} = frappe:fresh(5),
  ok = frappe:set(FS, key1, [a,b], 2),
  io:format("When looking up ~w I got ~w, and I expected {ok, [a,b]}~n",
            [key1, frappe:read(FS, key1)]),
  ok = frappe:set(FS, key2, [a,b], 2),
  ok = frappe:upsert(FS, key2,
                     fun({existing, Val}) ->
                         New = Val ++ [c,d],
                         {new_value, New, length(New)}
                     end),
  io:format("When looking up ~w I got ~w, and I expected nothing~n",
            [key1, frappe:read(FS, key1)]),
```