Modules in Futhark

vfr988

June 10, 2016

0.1 Abstract

mere bedre

1 Introduction

mere bedre

1.1 Problem definition

Is it possible to implement a module system in Futhark, which displays features comparable to the module system implemented in Standard ML? [**?**]

2 Motivation

The implementation of a module system, will greatly expand the usability of Futhark, without having side effects on the performance of the compiled Futhark code. The following subsections contain the features this project is implementing, and examples of why these features are desirable.

To be able to claim, that modules increases usability, we must first setup some metrics of usability.

Readability increases usability:
skriv mere her
Compartmentalization of functionality increases usability:
skriv mere her
A module system increases the ability to share code between developers.
skriv mere her

2.1 Abstraction increases readability.

In example, type aliasing lets us abstract from the actual definition of data types. When writing our source code, we can define our type aliases before writing the rest of our program. If we want to define a sphere in a three dimensional space, we want to define it with **1) a radius**, **2) a position** , and **3) a direction** that it is moving. Let us define a function that multiplies the speed of the sphere by a factor k.

```
fun {f32, {f32, f32, f32}, {f32, f32, f32}} multiply_velocity(
        {f32 r,
        {f32 x_pos , f32 y_pos , f32 z_pos} ,
        {f32 x_dir,f32 y_dir, f32 z_dir}} , f32 k)
          =
    {r, {x_pos, y_pos, z_pos}, {k * x_dir , k * y_dir, k * z_dir }}
```

With type aliasing, we can compartmentalize the data type, and remove the need for explicitally typing out every parameter of function input. Coupled with helper functions, we can now multiply the speed of the sphere like this:

```
type vec3 = {f32, f32, f32}
type position, direction = vec3
type radius = f32
type sphere = {radius, position, direction}

fun vec3 multiply_vector(vec3 {pos_x, pos_y, pos_z}, f32 k) =
  {k * pos_x , k * pos_y , k * pos_z}

fun sphere multiply_velocity(
  sphere {radius, position, direction}, f32 factor)  =
    let new_direction = multiply_vector(direction, factor)
    in {radius, position, new_direction}
```

As we are now using the sphere type as the function argument, we can pattern match on the type aliased values contained in the sphere type. Most importantly, the vectors of the sphere are abstracized into a single variables instead of tuples.
The end result is a shorter, more readable program 3.5.4

2.2 Approximating higher order functionality whilst keeping performance

It is possible to express higher-order functionality in Futhark, without taking a performance hit in the compiled Futhark code. We will reiterate on the concept of modules4, by introducing the concept of functors. First we repeat the three-dimensional vector module from earlier, but without declaring any particular primitive 3.1 type as the contained type of the vector:

```
  structure Vec3 =
    struct
      type vector = {t, t, t}
      fun vector add( ... ) = ...
      fun vector subtract( ... ) = ...
      fun vector multiply ( ... ) = ...
      fun vector divide ( ... ) = ...
    end
```

The structure above cannot be used on its own. Type `t` is not instantiated, and the module cannot be type checked, which causes an error.
We can solve that problem, by instantiating the abstract structure, using a simple functor; the `where`-clause.

```
  structure IntVec3 = Vec3 where type t = int
```

We can now access the structure `IntVec3` throughout the rest of the program.
The structure `IntVec3` is `Vec3`, except all instances of type `t` in `Vec3` is exchanged with type `int`.

To recap: functors allows us to define an abstract implementation of some structure **ONCE**, and lets us instantiate this structure any number of times, each time with our own type. This functionality will be elaborated on in a later section Functors**??**.

From a performance-concerned point of view, the module system is desirable. Every function in the written program, whether it is inside a structure or defined in the top level program, is ultimately accumulated into the same scope. 4.9.1

The entire program is then compiled, and the optimizations that makes FUTHARK `GO FAST` are applied to the included modules as well as the top level declarations.

2.3 Scope of project

The scope of the project is

- to implement a type aliasing system in Futhark
- to implement a module system in Futhark, which has:
  - The definition of structures containing types and functions
  - Nested modules; meaning that any structure can contain a structure
  - A well defined way of referring to types, structures and functions contained in a structure
- to implement functor functionality, so that Futhark supports the definition of abstract structures and concretizations of these structures.

2.4 Success definitions

fyld-ud

3 Type aliases

To make an implementation of functors?? in Futhark, it was necessary to implement type aliases first.

3.1 The language

The initial type system in Futhark supported the following type definitions:

$$
\begin{aligned}
Type = & \quad \texttt{Primitive type} \\
| & \quad (Types) \\
| & \quad [Type]
\end{aligned}
$$

$$
\begin{aligned}
Types = & \quad Type\ ,\ Types \\
| & \quad Type
\end{aligned}
$$

$$
\begin{aligned}
Primitivetype = & \quad UnsignedInteger \\
| & \quad SignedInteger \\
| & \quad FloatType \\
| & \quad Boolean
\end{aligned}
$$

Implementing type aliases expands our type constructions as following:

$$
\begin{aligned}
Type : &\quad \texttt{Primitive type} \\
| &\quad (Types) \\
| &\quad [Type] \\
| &\quad TypeAlias
\end{aligned}
$$

$$
\begin{aligned}
Types : &\quad Type\ ,\ Types \\
| &\quad Type
\end{aligned}
$$

$$
\begin{aligned}
Primitivetype : &\quad \texttt{UnsignedInteger} \\
| &\quad \texttt{SignedInteger} \\
| &\quad \texttt{FloatType} \\
| &\quad \texttt{Boolean}
\end{aligned}
$$

$$
TypeAlias : \quad \texttt{type}\ \mathit{strid} = Type
$$

where *type_id* is short for *string id*, a string identifier.

3.2 Interference rules

A type alias in a Futhark program is done like this: `type` *Strids* = *Type* , where *Type* is as defined in the grammar above.

Given an environment $\Gamma : (Functions, Types),\ Types = \{strid \rightarrow Type\}$
we can define the following interference rules for using type aliases:

$$\frac{\Gamma \vdash typedecl \Rightarrow \Gamma'}{\Gamma \vdash \texttt{type}\ strid\ \texttt{=}\ Type \Rightarrow \{strid \rightarrow Type\} \oplus \Gamma \Rightarrow \Gamma'\}}$$

where

$$\begin{aligned}&\{strid \rightarrow Type\} \oplus \Gamma \\ \Rightarrow & Types(\Gamma) \leftarrow\ Types(\Gamma) \cup \{strid \rightarrow Type\}\end{aligned}$$

iff the type alias declaration does not clash with the current environment. There is no clash, if the declaration obeys the three rules in the implementation subsection3.3.

We can assign the same type to several string ids simultaneously, as long as we don't declare the same type alias twice.

$$\frac{\Gamma \vdash typedecl \Rightarrow \Gamma' \qquad \text{Strids} = strid_1, ..., strid_n}{\Gamma \vdash \texttt{type}\ \ Strids\ \texttt{=}\ \ \ Type \Rightarrow \{strid_1 \rightarrow Type, .., strid_n \rightarrow Type\} \oplus \Gamma \Rightarrow \Gamma'\}}$$

where

$$\begin{aligned}&\{strid_1 \rightarrow Type, \cdots, strid_n \rightarrow Type,\} \oplus \Gamma \\ \Rightarrow & Types(\Gamma) \rightarrow\ Types(\Gamma) \cup \{strid_1 \rightarrow Type, \cdots, strid_n \rightarrow Type\}\end{aligned}$$

3.3 Implementation

A type alias $strid_i \rightarrow Type$ declaration is succesful, if three rules are followed:

1. The alias $strid_i$ is not already declared[1] in the current local environment. I.e. the example below:

```
type foo = i32
type foo = f32
```

[1]This is by convention of only being able to define a value *once*

2. The alias $strid_i$ refers to a type (or a type alias), that is either already defined in the current environment (including structure environments), or is in the same declaration chunk as $strid_i$. In the example below, `foo` refers to `bar`, but `bar` is in the same chunk as `foo`.

   Therefore, `foo` can be resolved by resolving `bar`. The implementation of this described in a later subsection3.4.2.

   ```
   type foo = bar
   type bar = {f32, i32}
   ```

   There are no hard limit to the number of type aliases that has to be checked, before a type alias is resolved:

   ```
   type foo = bar
   type bar = {f32, baz, i32}
   type baz = [{bee, bang, boo}]
   type boo ...

   ...

   type bep = i32
   ```

   Such a chain of type aliases is allowed, as long as the last of the three rules is obeyed:

3. The alias being resolved cannot be cyclically defined.
   Imagine that some type `type my_type = foo` is in the chain in the type aliasing example2 above.

   In this case, the compiler is first trying to resolve `foo` by resolving `bar`, and trying to resolving `bar` by resolving `baz`, et cetera.
   At some point, the compiler encounters `my_type`, and must resolve `foo` to continue - which creates a cycle, because `foo` is resolved by `bar`, and so on.

   To keep track of this, the compiler maintains the set of aliases that has been visited in the attempt to resolve some type alias. Every time another type alias has to be checked, the compiler first checks the set to find out, whether this alias is already on the list of aliases that needs to be resolved.

   If so, the compiler returns an error. The implementation of this can be read here3.3

3.4 Parsing a type alias

3.4.1 Data types for describing a type

Initially, the types parsed in a Futhark program were always represented as instances of the following datatype `TypeBase`:

```
data TypeBase shape as vn = Prim PrimType
                          | Array (ArrayTypeBase shape as vn)
                          | Tuple [TypeBase shape as vn]

data ArrayTypeBase shape as vn =
      PrimArray PrimType (shape vn) Uniqueness (as vn)
    -- ^ An array whose elements are primitive types.
    | TupleArray [TupleArrayElemTypeBase shape as vn] (shape vn) Uniqueness
    -- ^ An array whose elements are tuples.

data TupleArrayElemTypeBase shape as vn =
    PrimArrayElem PrimType (as vn) Uniqueness
  | ArrayArrayElem (ArrayTypeBase shape as vn)
  | TupleArrayElem [TupleArrayElemTypeBase shape as vn]
```

These are also the data types that are used in the internal Futhark program.
Before this project, Futhark parsed types from Futhark source code directly to TypeBases. However, we decided to add an intermediate data type between raw source code and TypeBases, to make type aliases available.

```
data UserType vn = UserPrim PrimType SrcLoc
                 | UserArray (UserType vn) (DimDecl vn) SrcLoc
                 | UserTuple [UserType vn] SrcLoc
                 | UserTypeAlias LongName SrcLoc
                 | UserUnique (UserType vn) SrcLoc

    deriving (Show)
```

The parser was changed, so type declarations in Futhark source would now be parsed as UserTypes and not typebases.

3.4.2 Adding resolved types to scope

```
  type TypeAliasTableM =
  ReaderT (HS.HashSet LongName) (StateT Scope TypeM)

typeAliasTableFromProg :: [TypeDefBase NoInfo VName]
                       -> Scope
                       -> TypeM Scope
typeAliasTableFromProg defs scope = do
  checkForDuplicateTypes defs
  execStateT (runReaderT (mapM_ process defs) mempty) scope
  where
        findDefByName name = find ((==name) . typeAlias) defs

        process :: TypeDefBase NoInfo VName
                -> TypeAliasTableM (StructTypeBase VName)
        process (TypeDef name (TypeDecl ut NoInfo) _) = do
          t <- expandType typeOfName ut
          modify $ (addType name t)
          return t

        typeOfName :: LongName -> SrcLoc
                   -> TypeAliasTableM (StructTypeBase VName)
        typeOfName (prefixes, name) loc = do
          inside <- ask
          known <- get
          case typeFromScope (prefixes, name) known of
            Just t -> return t
            Nothing
              | (prefixes, name) `HS.member` inside ->
                  throwError $ CyclicalTypeDefinition loc name
              | Just def <- findDefByName name ->
                  local (HS.insert (prefixes, name)) $ process def
              | otherwise ->
                  throwError $ UndefinedAlias loc name
```

TypeAliasTableM is a monad stack that is used to resolve a list of type alias declarations in a declaration chunk.
It is a reader monad transformer that contains a state monad transformer, that contains the TypeM monad.

The reader monad is used to contain a HashSet as its environment. This environment is used to keep check of cyclical definitions as described in 3.

For each type alias, we use the function `process` to resolve the type, and modify the scope contained within the state monad of the ReaderT.

Resolving a type from a type aliasing is done using the function `typeOfName` in the code snippet3.4.2. `typeOfName` tries to resolve the type by name by retrieving the scope contained in the transformed State monad inside the reader.

If this is not immediately possible, we must either continue our search for the type, throw an error due to a cyclical type definition, or throw an error because the type alias has not been defined yet.
If our attempt to resolve $strid_a$ leads to another alias $strid_b$[2], we add $strid_a$ to our reader environment using the function `local`, and process $alias_b$ instead.

Acknowledgement: The initial design of `expandType` and the addition of type aliases to the scope was initially much larger, but was reduced in size by Troels Henriksen, who rewrote the process to use monads, and reduced some code duplication.

3.4.3 Converting UserType to TypeBase

```
expandType :: (Applicative m, MonadError TypeError m) =>
               (LongName -> SrcLoc -> m (StructTypeBase VName))
            -> UserType VName
            -> m (StructTypeBase VName)

expandType look (UserTypeAlias longname loc) =
  look longname loc
expandType _ (UserPrim prim _) =
  return $ Prim prim
expandType look (UserTuple ts _) =
  Tuple <$> mapM (expandType look) ts
expandType look (UserArray t d _) = do
  t' <- expandType look t
  return $ arrayOf t' (ShapeDecl [d]) Nonunique
expandType look (UserUnique t loc) = do
  t' <- expandType look t
  case t' of
    Array{} -> return $ t' `setUniqueness` Unique
    _       -> throwError $ InvalidUniqueness loc $ toStructural t'
```

3.4.4 Why we added UserType instead of extending TypeBase

Adding UserType and then resolving these into TypeBases whilst running the program through the TypeChecker, removes the need of handling UserAliases after the type check, where these aliases are resolved.

Furthermore, not all information about a TypeBase declaration can actually be claimed already at program parse time. Some information about i.e. IKKE SIKKER HER array dimensionality in regards to aliased arrays, is decided within the type checker as well.

[2] line 27-33

3.4.5 The slip from type aliases to realized types

Since every type alias is resolved in the type checker, the UserTypes are not used after the type check.

The data type for a type declaration is this:

```
data TypeDeclBase f vn =
  TypeDecl { declaredType :: UserType vn
                             -- ^ The type declared by the user.
           , expandedType :: f (StructTypeBase vn)
                             -- ^ The type deduced by the type checker.
           }
```

An unresolved type looks like this:

```
TypeDecl userType NoInfo
```

After resolve, NoInfo has been filled out with a variable of type `Info TypeBase`, giving us the following TypeDecl:

```
TypeDecl usertype (Info typebase).
```

At any point after the type check, only the `expandedType` of TypeDecl is used.

3.5 Results

The addition of type aliases works without any issues. To verify this, Futhark has been tested to pass all of the tests in futharks test suite[3].

However, it was also necessary to write tests to specifically confirm, that the implementation respects the rules defined in 3.3

3.5.1 A type cannot be defined twice in the same environment

From `alias-error3.fut` in `futhark/src/data/tests/types`:

```
-- You may not define the same alias twice.
--
-- ==
-- error: Duplicate.*mydup

type mydup = int
type mydup = f32

fun int main(int x) = x
```

This program fails as expected.

[3] located in folder `futhark/data/tests`

3.5.2 A type alias cannot be defined, if it refers to a type alias that has not been defined

From `alias-error4.fut` in `futhark/src/data/tests/types`:

```
-- No undefined types!
--
-- ==
-- error: .*not defined.*

type foo = bar

fun foo main(foo x) = x
```

This program fails as expected.

3.5.3 A type alias cannot be cyclically defined

From `alias-error5.fut` in `futhark/src/data/tests/types`:

```
-- No tricky circular types!
--
-- ==
-- error: .*cycl.*

type t0 = [t1]
type t1 = {int, float, t2}
type t2 = t0

fun t1 main(t1 x) = x
```

This program fails as expected.

3.5.4 Example of planet simulations being simplified by type aliases

A nice example of the benefits of type aliasing, is the N-body simulation (`nbody`), which is a simulation over the n-body problem[4].

The original Futhark implementation of the simulation contained function arguments of tuples with arity up to 10. Whilst it is still necessary to bring all the arguments throughout the program, type aliasing makes the program itself much more readable:

```
 -- N-body simulation based on the one from Accelerate:
-- https://github.com/AccelerateHS/accelerate-examples/tree/master/examples/n-
   body
--
-- Type descriptions:
--
-- type mass = f32
-- type position = {f32, f32, f32}
-- type acceleration = {f32, f32, f32}
-- type velocity = {f32, f32, f32}
-- type body = (position, mass, velocity, acceleration)
--            =~ {f32, f32, f32, -- position
--                f32,           -- mass
--                f32, f32, f32, -- velocity
--                f32, f32, f32} -- acceleration
--

fun {f32, f32, f32}
  vec_add({f32, f32, f32} v1,
             {f32, f32, f32} v2) =
  let {x1, y1, z1} = v1
  let {x2, y2, z2} = v2
  in {x1 + x2, y1 + y2, z1 + z2}

fun {f32, f32, f32}
  vec_subtract({f32, f32, f32} v1,
               {f32, f32, f32} v2) =
  let {x1, y1, z1} = v1
  let {x2, y2, z2} = v2
  in {x1 - x2, y1 - y2, z1 - z2}

fun {f32, f32, f32}
  vec_mult_factor(f32 factor,
                  {f32, f32, f32} v) =
  let {x, y, z} = v
  in {x * factor, y * factor, z * factor}

fun f32
  dot({f32, f32, f32} v1,
      {f32, f32, f32} v2) =
  let {x1, y1, z1} = v1
  let {x2, y2, z2} = v2
  in x1 * x2 + y1 * y2 + z1 * z2
```

[4] `https://en.wikipedia.org/wiki/N-body_problem`

```
fun {f32, f32, f32}
  accel(f32 epsilon,
        {f32, f32, f32} pi,
        f32 mi,
        {f32, f32, f32} pj,
        f32 mj) =
  let r = vec_subtract(pj, pi)
  let rsqr = dot(r, r) + epsilon * epsilon
  let invr = 1.0f32 / sqrt32(rsqr)
  let invr3 = invr * invr * invr
  let s = mj * invr3
  in vec_mult_factor(s, r)

fun {f32, f32, f32}
  accel_wrap(f32 epsilon,
             {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32} body_i,
             {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32} body_j) =
  let {xi, yi, zi, mi, _, _, _, _, _, _} = body_i
  let {xj, yj, zj, mj, _, _, _, _, _, _} = body_j
  let pi = {xi, yi, zi}
  let pj = {xj, yj, zj}
  in accel(epsilon, pi, mi, pj, mj)

fun {f32, f32, f32}
  move(f32 epsilon,
       [{f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}] bodies,
       {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32} body) =
  let accels = map(fn {f32, f32, f32} ({f32, f32, f32, f32, f32, f32, f32, f32
      , f32, f32} body_other) =>
                     accel_wrap(epsilon, body, body_other),
                   bodies)
  in reduceComm(vec_add, {0f32, 0f32, 0f32}, accels)

fun [{f32, f32, f32}]
  calc_accels(f32 epsilon,
              [{f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}] bodies) =
  map(move(epsilon, bodies), bodies)

fun {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}
  advance_body(f32 time_step,
               {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32} body) =
  let {xp, yp, zp, mass, xv, yv, zv, xa, ya, za} = body
  let pos = {xp, yp, zp}
  let vel = {xv, yv, zv}
  let acc = {xa, ya, za}
  let pos' = vec_add(pos, vec_mult_factor(time_step, vel))
  let vel' = vec_add(vel, vec_mult_factor(time_step, acc))
  let {xp', yp', zp'} = pos'
  let {xv', yv', zv'} = vel'
  in {xp', yp', zp', mass, xv', yv', zv', xa, ya, za}

fun {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}
  advance_body_wrap(f32 time_step,
                    {f32, f32, f32, f32, f32, f32, f32, f32, f32, f32} body,
```

```
                    {f32, f32, f32} accel) =
  let {xp, yp, zp, m, xv, yv, zv, _, _, _} = body
  let accel' = vec_mult_factor(m, accel)
  let {xa', ya', za'} = accel'
  let body' = {xp, yp, zp, m, xv, yv, zv, xa', ya', za'}
  in advance_body(time_step, body')

fun [{f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}, n]
  advance_bodies(f32 epsilon,
                 f32 time_step,
                 [{f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}, n]
                    bodies) =
  let accels = calc_accels(epsilon, bodies)
  in zipWith(advance_body_wrap(time_step), bodies, accels)

fun [{f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}, n]
  advance_bodies_steps(i32 n_steps,
                       f32 epsilon,
                       f32 time_step,
                       [{f32, f32, f32, f32, f32, f32, f32, f32, f32, f32}, n]
                           bodies) =
  loop (bodies) = for i < n_steps do
    advance_bodies(epsilon, time_step, bodies)
  in bodies

fun {[f32, n], [f32, n], [f32, n], [f32, n], [f32, n], [f32, n], [f32, n], [
   f32, n], [f32, n], [f32, n]}
  main(i32 n_steps,
       f32 epsilon,
       f32 time_step,
       [f32, n] xps,
       [f32, n] yps,
       [f32, n] zps,
       [f32, n] ms,
       [f32, n] xvs,
       [f32, n] yvs,
       [f32, n] zvs,
       [f32, n] xas,
       [f32, n] yas,
       [f32, n] zas) =
  let bodies = zip(xps, yps, zps, ms, xvs, yvs, zvs, xas, yas, zas)
  let bodies' = advance_bodies_steps(n_steps, epsilon, time_step, bodies)
  in unzip(bodies')
```

to

nbody using type aliases:

```
type mass = f32
type vec3 = (f32, f32, f32)
type position, acceleration, velocity = vec3

type body = (position, mass, velocity, acceleration)

fun vec3 vec_add(vec3 v1, vec3 v2) =
  let (x1, y1, z1) = v1
  let (x2, y2, z2) = v2
  in (x1 + x2, y1 + y2, z1 + z2)

fun vec3 vec_subtract(vec3 v1, vec3 v2) =
  let (x1, y1, z1) = v1
  let (x2, y2, z2) = v2
  in (x1 - x2, y1 - y2, z1 - z2)

fun vec3 vec_mult_factor(f32 factor, vec3 v) =
  let (x, y, z) = v
  in (x * factor, y * factor, z * factor)

fun f32 dot(vec3 v1, vec3 v2) =
  let (x1, y1, z1) = v1
  let (x2, y2, z2) = v2
  in x1 * x2 + y1 * y2 + z1 * z2

fun velocity accel(f32 epsilon, vec3 pi, f32 mi, vec3 pj, f32 mj) =
  let r = vec_subtract(pj, pi)
  let rsqr = dot(r, r) + epsilon * epsilon
  let invr = 1.0f32 / sqrt32(rsqr)
  let invr3 = invr * invr * invr
  let s = mj * invr3
  in vec_mult_factor(s, r)

fun vec3 accel_wrap(f32 epsilon, body body_i, body body_j) =
  let (pi, mi, _ , _) = body_i
  let (pj, mj, _ , _) = body_j
  in accel(epsilon, pi, mi, pj, mj)

fun position move(f32 epsilon, [body] bodies, body this_body) =
  let accels = map(fn acceleration (body other_body) =>
                     accel_wrap(epsilon, this_body, other_body),
                   bodies)
  in reduceComm(vec_add, (0f32, 0f32, 0f32), accels)

fun [acceleration] calc_accels(f32 epsilon, [body] bodies) =
  map(move(epsilon, bodies), bodies)

fun body advance_body(f32 time_step, body this_body) =
  let (pos, mass, vel, acc) = this_body
  let pos' = vec_add(pos, vec_mult_factor(time_step, vel))
  let vel' = vec_add(vel, vec_mult_factor(time_step, acc))
  let (xp', yp', zp') = pos'
```

```
  let (xv', yv', zv') = vel'
  in (pos', mass, vel', acc)

fun body advance_body_wrap(f32 time_step, body this_body, acceleration accel)
    =
  let (pos, mass, vel, acc) = this_body
  let accel' = vec_mult_factor(mass, accel)
  let body' = (pos, mass, vel, accel')
  in advance_body(time_step, body')

fun [body, n] advance_bodies(f32 epsilon, f32 time_step, [body, n] bodies) =
  let accels = calc_accels(epsilon, bodies)
  in zipWith(advance_body_wrap(time_step), bodies, accels)

fun [body, n] advance_bodies_steps(i32 n_steps, f32 epsilon, f32 time_step,
                                    [body, n] bodies) =
  loop (bodies) = for i < n_steps do
    advance_bodies(epsilon, time_step, bodies)
  in bodies

fun body wrap_body (f32 posx, f32 posy, f32 posz,
                    f32 mass,
                    f32 velx, f32 vely, f32 velz,
                    f32 accx, f32 accy, f32 accz) =
  ((posx, posy, posz), mass, (velx, vely, velz), (accx, accy, accz))

fun (f32, f32, f32, f32, f32, f32, f32, f32, f32, f32) unwrap_body(body
   this_body) =
  let ((posx, posy, posz), mass, (velx, vely, velz), (accx, accy, accz)) =
      this_body
  in (posx, posy, posz, mass, velx, vely, velz, accx, accy, accz)



fun ([f32, n], [f32, n], [f32, n], [f32, n], [f32, n], [f32, n], [f32, n], [
   f32, n], [f32, n], [f32, n])
  main(i32 n_steps,
       f32 epsilon,
       f32 time_step,
       [f32, n] xps,
       [f32, n] yps,
       [f32, n] zps,
       [f32, n] ms,
       [f32, n] xvs,
       [f32, n] yvs,
       [f32, n] zvs,
       [f32, n] xas,
       [f32, n] yas,
       [f32, n] zas) =
  let bodies  = map(wrap_body, zip(xps, yps, zps, ms, xvs, yvs, zvs, xas, yas,
      zas))
  let bodies' = advance_bodies_steps(n_steps, epsilon, time_step, bodies)
  let bodies'' = map(unwrap_body, bodies')
   in unzip(bodies'')
```

3.6 Future work

To make functors work, it must be possible to declare an abstract type alias in a structure.

That will allow for a structure definition, where a type variable[5] has been declared, but not defined, until the containing structure is instantiated using a functor.
This has not been implemented yet.

[5] in the form of a *strid*

4 Structures

Introduction to structures: Without structures, every function and type in futhark shares the same scope. Implementing modules lets us create functions that are alike, but keeps distinctions between them. Take this example of a program with two different vector types:

```
type vec3 = {f32, f32, f32}
type vec4 = {f32, f32, f32, f32}

fun vec3 vec3_plus(
  vec3 {a_1, ... , a_3},
  vec3 {b_1, ..., b_3}
  ) = {a_1 + b_1, ... , a_3 + b_3}

fun vec4 vec4_plus(
  vec4 {a_1, ... , a_4},
  vec4 {b_1, ..., b_4}
  ) = {a_1 + b_1, ... , a_4 + b_4}
```

Let us try compartmentalizing a vector type and its functions into a structure.
In the following example, we have defined two different modules, each containing a structure, and a futhark program which includes and utilizes these modules:

```
Vec3Float.fut:
  structure Vec3Float =
    struct
      type vector = {f32, f32, f32}
      fun vector plus( ... ) = ...
      fun vector minus( ... ) = ...
      fun vector multiply ( ... ) = ...
    end

Vec4Float.fut:
  structure Vec4Float =
    struct
      type vector = {f32, f32, f32, f32}
      fun vector plus( ... ) = ...
      fun vector minus( ... ) = ...
      fun vector multiply ( ... ) = ...
    end
```

myprogram.fut:

```
include Vec3Float
include Vec4Float

type vec3 = Vec3Float.vector
type vec4 = Vec4Float.vector

fun vec4 foo(vec3 vector) =
  let {a, b, c} = Vec3.plus(vector, vector)
  in Vec4.multiply({a, b, c, 1.0f} , 4.0f)
```

Whilst it *is* possible to create libraries without a module implementation [6], the user runs a risk of running into errors like `MulipleDefinitionError`[7], if any of the library functions uses any of the names, that the local user is using as well.

The module system removes this hazard, as application of functions and types are done using **longnames**, which adds prefixes to names. This way, functions can have the same name, as long as they do not share the same prefix.

> **Longnames:**
> A longname consists of any amount of prefixes followed by a dot, followed by the string id of the desired function or type:
>
> $$\begin{array}{lr} \mathit{LongName}: & \texttt{prefix.}\mathit{LongName} \\ | & \texttt{identifier} \end{array}$$
>
> In Futhark we will be using string ids for declarations, and longnames for the accessing types and functions in structures.

4.1 Accessing types and functions within structures

To work with type aliases and modules, we need to define the internal environment of Futhark during compile time.

Before starting this project, the environment of Futhark could be described like this:
$\Gamma = (FE)$, where FE is a function environment, mapping function ids to functions:
$\{funid \rightarrow funexpr\}$.

It is the goal of this project to expand the environment of Futhark, so that
$\Gamma = (FE, TE, SIGE, STRUCTE, FUNCTE)$, where the additions are a type alias environment, a signature environment, a structure environment and a functor environment.

[6]By including libraries that adds functions to the top level environment
[7]Multiple functions of same name defined

Type aliases, signatures, structures and functors are described in their respective sections. A structure can be regarded as a structure name and an environment contained in the structure, so that $\{strid \to \Gamma_{strid}\}$ is a mapping in $STRUCTE$.

$$\frac{\Gamma_{local} \vdash fundef \Rightarrow \Gamma'_{local}}{\Gamma_{local} \vdash fundef_{strid} \Rightarrow \Gamma_{local} \oplus fundef_{strid} \Rightarrow \Gamma'_{local}}$$

where

$$\begin{gathered} fundef_{strid} \oplus \Gamma_{local} \\ \Rightarrow \Gamma_{local} \leftarrow Functions(\Gamma_{local}) \cup \{strid \rightarrow fundef\}) \end{gathered}$$

iff the function the function body is otherwise well-formed[a].

[a] I will not go further into the functionality of functions, as it is not within scope of this project.

Figure 1: Rule for adding a function to the environment

$$\frac{\Gamma_{local} \vdash typedecl \Rightarrow \Gamma'_{local}}{\Gamma_{local} \vdash \texttt{type}\ strid\ \texttt{=}\ Type \Rightarrow \{strid \rightarrow Type\} \oplus \Gamma_{local} \Rightarrow \Gamma'_{local}\}}$$

where

$$\begin{gathered} \{strid \rightarrow Type\} \oplus \Gamma_{local} \\ \Rightarrow \Gamma_{local} \leftarrow Types(\Gamma) \cup \{strid \rightarrow Type'\} \end{gathered}$$

and

$$Type' \leftarrow Type\ resolved\ in \Gamma_{local}$$

as defined in 3.2.

Any Γ_{strid} contains its own function-, typealias-, signature-, structure- and functor declarations.

4.2 Interference rules

Now that we can discern between the glocal environment, and any number of environments in structures, we must redefine the environment behaviour of Futhark at compile time.

After expanding the environment to contain the three new elements, we can define the addition of any of these three elements follow the following rule:

$$\frac{\Gamma_{local} \vdash fundecl \Rightarrow \Gamma'_{local}}{\Gamma_{local} \vdash \texttt{fun}\ strid\ \texttt{=}\ Function \Rightarrow \{strid \to Function\} \oplus \Gamma_{local} \Rightarrow \Gamma'_{local}\}}$$

where $\{strid \to Function\} \oplus \Gamma_{local} = \Gamma_{local} := Functions_\Gamma \cup \{strid \to Function\}$ and $Function' := Functionresolvedin\Gamma_{local}$ as defined in 3.2.

$$\frac{\Gamma_{local} \vdash sigdecl \Rightarrow \Gamma'_{local}}{\Gamma_{local} \vdash \texttt{fun}\ strid\ \texttt{=}\ Signature \Rightarrow \{strid \to Signature\} \oplus \Gamma_{local} \Rightarrow \Gamma'_{local}\}}$$

where $\{strid \to Signature\} \oplus \Gamma = \Gamma := Signatures_\Gamma \cup \{strid \to Signature\}$ as defined in 3.2.
Note, that Signatures should be independently defined, and not refer to any other signatures.

4.2.1 Adding declarations to the environment

4.3 Interference rules for structure definitions

We must take a closer look at what happens, when a structure is defined. The definition of a structure in Futhark has the following behaviour.

4.3.1 Rule for multiple declarations of same name in same local environment:

Building a structure from a list of *structdecls* allows for several different declarations in the same environment of the same name, as long as it is different types of declarations:

For $decl : (name, type), type \in \{Function, TypeAlias, Structure, Signature\}$:

$$\frac{\Gamma_{local} \vdash (decl ::{}^{a} decls) \Rightarrow \Gamma'_{local}}{\Gamma_{local} \vdash structdecl \Rightarrow structdecl \oplus \Gamma_{local}\Gamma'_{local}}$$

where $decl(name, type) \oplus \Gamma_{local}$ adds the declaration to Γ_{local} , iff $(name, type) \notin decls$.

This works, because Functions, TypeAliases, Structures and Signatures are stored in their respective tables, instead of storing all of them in the same table.

[a] this is the list constructor operator

$$\frac{\Gamma_{local} \vdash structdecl \Rightarrow \Gamma'_{local} \qquad \{structdecl_{strid} \rightarrow decls_{strid}\}}{\Gamma_{local} \vdash \texttt{struct}\ strid = Declarations\ end \Rightarrow \Gamma_{strid} = \Gamma_{local} \oplus Declarations; \Gamma'_{local} = \Gamma_{strid} \otimes \Gamma_{local}}$$

.

where $\Gamma_{local} \oplus structdecls_{strid}$ builds an environment Γ_{strid}, from $structdecls_{strid}$ interpreted in the environment Γ_{local}.
If $structdecls_{strid}$ contains a new function, type or structure, with a name that is already used in Γ_{local}, the fun- or type definition in Γ_{strid} is will be the defined in $structdecls_{strid}$

and $\Gamma_{strid} \otimes \Gamma_{local} = \Gamma_{local}Structures := \Gamma_{local}Structures \cup strid \rightarrow \Gamma_{strid}$.
Please note, that Γ_{local} does not have a reference $Structure\{strid \rightarrow \Gamma_{strid}\}$, until the entire structure $structdef_{strid}$ has been parsed.

For $decl : (name, type), type \in \{Function, TypeAlias, Structure, Signature\}$:

$$\frac{\Gamma_{local} \vdash decl_{new}(name, type) \Rightarrow \Gamma'_{local} \qquad (name, type) \downarrow \Gamma_{local} \rightarrow decl_{current}}{\Gamma_{local} \vdash decl \Rightarrow \Gamma' := decl_{new} \oplus \Gamma}$$

where $(name, type) \downarrow \Gamma_{local}$ returns the declaration of (name, type) as defined in Γ_{local} and $decl_{new} \oplus \Gamma$ replaces the current declaration of (name, type) in Γ_{local}

4.3.2 Rule for adding a structure to the local environment

4.3.3 Rule for variable shadowing

The following interference rule defines the variable shadowing behaviour of the Futhark module system.

4.4 Interference rules for interpreting functions and types in an environment with structures

There are three cases where it is necessary to resolve a function or a type from a longname:
1) When applying a function as an expression in a function expression; i.e.

```
let myNumber = Constants.numberFour()
```

2) When using a function as an argument in a currying function; i.e.

```
let numbers = [1, 2, 3, 4] in
let sum = reduce(MathLib.plus , 0 , numbers)
```

3) When using a type definition from a structure; i.e.

```
type int_pair = Pairs.Int.t
```

$$\frac{\Gamma_{local} \vdash \texttt{apply}longname \Rightarrow \texttt{apply}function}{\Gamma_{local} \vdash letx = longname() \Rightarrow x :=\downarrow longname}$$

where

$$\downarrow longname = \qquad returngetFromEnv(longname, \Gamma_{local})$$

.

We can define the interference rule for using a longname in the three different cases. As all three cases handles resolving a longname the same way, the rule will only be called for the first case:

4.5 Implementation

A program in the Futhark type checker is defined as a list of unchecked declarations. I will not describe the process in details, but will instead explain specific parts of the code. I will specifically focus on parts of the code, that corresponds to the interference rules described earlier.

4.5.1 The Scope datatype

To describe the environment in the Futhark compiler, we use the Scope datatype:

```
data Scope  = Scope { envVtable :: HM.HashMap VName Binding
                    , envFtable :: HM.HashMap Name FunBinding
                    , envTAtable :: HM.HashMap Name TypeBinding
                    , envModTable :: HM.HashMap Name Scope
                    , envBreadcrumb :: LongName
                    }
```

Given $\Gamma = (Functions, TypeAliases, Structures)$, then *Functions* is implemented in `envFtable`, *TypeAliases* in `envTAtable` and *Structures* in `envModTable`.

4.5.2 checkProg

The type check is initiated in the functions checkProg and checkProg'. checkProg' checks the initial top level of declaratations for duplicates.

```
checkProg :: UncheckedProg -> Either TypeError (Prog, VNameSource)
checkProg prog = do
  checkedProg <- runTypeM initialScope src $ Prog <$> checkProg' (progDecs
     prog')
  return $ flattenProgFunctions checkedProg
  where
    (prog', src) = tagProg' blankNameSource prog
```

```
      fun int four() = 4
\\
      structure M0 =
        struct
          fun int double(int a) = a + a
        end
\\
      fun int main() =
        let x = four() in
        M0.double(x, x)
```

`four()` does not have `M0` available in its local environment, but `main()` does, because `M0` HAS already been parsed, at the point where `main()` is declared

```
\\
  checkProg' :: [DecBase NoInfo VName] -> TypeM [DecBase Info VName]
  checkProg' decs = do
    _ <- checkForDuplicateDecs decs
    (_, decs') <- checkDecs decs
    return decs'
```

4.5.3 Checking for duplicates

As described in rule **??**, we allow multiple declarations of same name, as long as they don't share type: In example, checking for two function definitions of the same type is done in the first case of checkForDuplicateDecs:

```
checkForDuplicateDecs :: [DecBase NoInfo VName] -> TypeM ()
checkForDuplicateDecs decs = do
  _ <- foldM_ f HM.empty decs
  return ()
  where
    f known dec@(FunOrTypeDec (FunDec (FunDef _ (name,_) _ _ _ loc))) =
      case HM.lookup name known of
        Just dec'@(FunOrTypeDec (FunDec FunDef{})) ->
          throwError $ DupDefinitionError name loc $ decLoc dec'
        _ -> return $ HM.insert name dec known
```

4.6 Dividing a Futhark program into chunks

To facilitate variable shadowing, it was necessary to split a parsed Futhark program into chunks.
A structure declaration alters the enviroment drastically, by enabling the following program declarations following the structure to access the environment of the structure. Therefore we divide any list of declaration into type- and function declarations, and structure declarations. An example is given in figure**??**
This chunking is done recursively on a list of declarations:

```
chompDecs :: [DecBase NoInfo VName]
          -> ([FunOrTypeDecBase NoInfo VName], [DecBase NoInfo VName])
chompDecs decs = f ([], decs)
  where f (foo , FunOrTypeDec dec : xs ) = f (dec:foo , xs)
        f (foo , bar) = (foo, bar)
```

4.7 Checking function- and type declarations

```
checkDecs :: [DecBase NoInfo VName] -> TypeM (Scope, [DecBase Info VName])
checkDecs decs = do
\\
    let (funOrTypeDecs, rest) = chompDecs decs
    scopeFromFunOrTypeDecs <- buildScopeFromDecs funOrTypeDecs
    local (const scopeFromFunOrTypeDecs) $ do
      checkedeDecs <- checkFunOrTypeDec funOrTypeDecs
      (scope, rest') <- checkDecs rest
      return (scope , checkedeDecs ++ rest')
\\
checkDecs [] = do
  scope <- ask
  return (scope, [])
```

1) The current chunk of declarations is built using chompDecs.
2) A scope is built from this chunk, by using the three interference rules in section 4.2.1.
3) The scope from step 2 is now the local scope in the following function execution.
4) checkFunOrTypeDec does the actual typechecking of the function declaration.
5) The remainder of the declarations chunked in chompDecs is checked using checkDecs. Note, that the first element of `rest` must be a structure declaration, due to the implementation of chompDecs.

4.8 Checking structure declarations

Please note that structures are currently called modules inside Futharks compiler.

```
checkDecs (ModDec modd:rest) = do
  (modscope, modd') <- checkMod modd
  local (addModule modscope) $
    do
      (scope, rest') <- checkDecs rest
      return (scope, ModDec modd' : rest' )
```

When checkDecs encounters a `ModDec`, `checkMod` is called to resolve the `ModDec`. The ModDec defines an environment of functions and type declarations called modscope, which we add to the local environment by calling `local (addModule modscope)`
This part of the code is the implementation of rule**??**.

```
checkMod :: ModDefBase NoInfo VName -> TypeM (Scope , ModDefBase Info VName)
checkMod (ModDef name decs loc) =
  local (`addBreadcrumb` name) $ do
    _ <- checkForDuplicateDecs decs
    (scope, decs') <- checkDecs decs
    return (scope, ModDef name decs' loc)
```

```
type LongName = ([Name], name)

typeFromScope :: LongName -> Scope -> Maybe TypeBinding
typeFromScope (prefixes, name) scope = do
  scope' <- envFromScope prefixes scope
  let taTable = envTAtable scope'
  HM.lookup name taTable

funFromScope :: LongName -> Scope -> Maybe FunBinding
funFromScope (prefixes, name) scope = do
  scope' <- envFromScope prefixes scope
  let taTable = envFtable scope'
  HM.lookup name taTable


envFromScope :: [Name] -> Scope -> Maybe Scope
  envFromScope (x:xs) scope =
    case HM.lookup x (envModTable scope) of
      Just scope' -> envFromScope xs scope'
      Nothing -> Nothing
envFromScope [] scope = Just scope
```

checkMod first adds a "breadcrumb" of the structure's name (*strid*) to the local environment. This is where the transformation in rule **??** of $\Gamma_{local} \Rightarrow \Gamma_{strid}$ is implemented.
After the environment has been given its name, the internal declaration of the structure is read using `checkDecs`.

4.9 Resolving the application of a longname

Resolving the application for a longname is done through two short recursive functions.
Figure **??** shows the implementation of the longname rule **??**:

4.9.1 Including structures, functions and types from other files

The Futhark `include`-statement lets the user include other files into the current program.

This is implemented by letting the Futhark compiler combine the source code from all the included files, with the declarations in the main program. The combined program is then sent passed on through the rest of the compiler.

As the Futhark program with an arbitrary number of includes is merged into a single program within the compiler, we can compartmentalize a program into discrete files.

However, this creates a hazard: there might be declarations in the included code, which has names that overlap with names in the remaining code, so that a `DuplicateDefinitionError` is triggered. the included code has any declarations that results in a duplicate definition error. Refer to **??** for a possible future solution.

4.9.2 Keeping track of function names

Futharks variable shadowing made it necessary to extend the type of the function declaration type.
Initially, a function had only the declared name of the function as an identifier. However, it was necessary to extend this name into a pair:

`type FunName = (declaredName :: Name, expandedName :: LongName).`

When a function is added to the function table during `buildFtable`, it is added to the function table together with it's longname. The function's longname contains the function's name, and the name of the (potentially) nested structures it was defined in.

4.10 Tests

To verify that the implementation of structures was correct, a series of test programs were written to verify, whether the interference rules defined for the structures, actually held in an executed Futhark program.

In the following tests, programs which break the rules are expected to return with an error.

4.10.1 Testing rule for multiple declarations [...]

The following tests are implemented to test whether rule **??** holds. `duplicate_def0.fut`:

```
-- This test is written to ensure that the same name can be used
-- for different declarations in the same local environment, as long as their
   types does not overlap.
-- ==
-- input { 4 }
-- output { 4 }

type foo = int
fun foo foo(int a) = a + a
struct foo
  {
    fun int one() = 1
  }

fun int main(int x) = x
```

This test passes.
In the following test, the structure foo contains declarations of name foo also. `duplicate_def1.fut`:

```
-- The struct opens a new environment, which lets us use names again, which
   were used
-- in a previous scope.
-- ==
-- input { }
-- output { (1 , 2.0) }

type foo = int
struct foo
```

```
  {
    fun int foo() = 1
    struct foo
      {
        type foo = float
        fun foo foo() = 2.0
      }
  }

fun (foo, foo.foo.foo) main() = ( foo.foo() , foo.foo.foo())
```

This test passes.
In the following tests, the programmer has attempted to declare functions and types in the same environment, twice.

`duplicate_error0.fut`:

```
-- This test fails with a DuplicateDefinition error.
-- ==
-- error: .*Dup.*

fun int bar() = 1
struct foo
  {
    fun foo foo() = 1
  }
fun int bar() = 2

fun int main() = 0
```

This test passes.
In the following test, the structure foo contains declarations of name foo also. `duplicate_error1.fut`:

```
-- This test fails with a DuplicateDefinition error.
-- ==
-- error: .*Dup.*

type foo = int

struct foo
  {
    fun foo foo() = 1
  }
type foo = float

fun int main() = 0
```

This test passes.

4.10.2 Testing structures can be called as expected

The following tests are implemented to test that calling structures works as defined in rule **??**.

`calling_nested_module.fut`:

```
-- ==
-- input {
--   10 21
-- }
-- output {
--   6
-- }

type t = int
struct NumLib  {
    fun t plus(t a, t b) = a + b
  struct BestNumbers
    {
      fun t four() = 4
      fun t seven() = 42
      fun t six() = 41
    }
  }


fun int localplus(int a, int b) = NumLib.plus (a,b)

fun int main(int a, int b) =
  localplus(NumLib.BestNumbers.four() ,   2)
```

This test passes.
Currying functions such as map and reduce works as expected:
`map_with_structure0.fut`:

```
-- Testing whether it is possible to use a function
-- from a struct in a curry function (map)
-- ==
-- input {
--   [1, 2, 3 ,4, 5, 6, 7, 8, 9, 10]
-- }
--  output {
--  55
-- }

struct f  {
    fun int plus(int a, int b) = a+b
  }

fun int main([int] a) = reduce(f.plus , 0 , a)
```

This test passes.
Using structures from an include works as expected:
`Vec3.fut`:

```
-- ==
-- tags { disable }
struct Vec3
  {
    struct F32
```

```
    {
      type t = ( f32 , f32 , f32 )
      fun t add(t a , t b) =
        let (a1, a2, a3) = a in
        let (b1, b2, b3) = b in
        (a1 + b1, a2 + b2 , a3 + b3)

      fun t subtract(t a , t b) =
        let (a1, a2, a3) = a in
        let (b1, b2, b3) = b in
        (a1 - b1, a2 - b2 , a3 - b3)

      fun t scale(f32 k , t a) =
        let (a1, a2, a3) = a in
        (a1 * k, a2 * k , a3 * k)

      fun f32 dot(t a , t b) =
        let (a1, a2, a3) = a in
        let (b1, b2, b3) = b in
        a1*b1 + a2*b2 + a3*b3
    }

  struct Int
    {
      type t = ( int , int , int )
      fun t add(t a , t b) =
        let (a1, a2, a3) = a in
        let (b1, b2, b3) = b in
        (a1 + b1, a2 + b2 , a3 + b3)

      fun t subtract(t a , t b) =
        let (a1, a2, a3) = a in
        let (b1, b2, b3) = b in
        (a1 - b1, a2 - b2 , a3 - b3)

      fun t scale(int k , t a) =
        let (a1, a2, a3) = a in
        (a1 * k, a2 * k , a3 * k)

      fun int dot(t a , t b) =
        let (a1, a2, a3) = a in
        let (b1, b2, b3) = b in
        a1*b1 + a2*b2 + a3*b3
    }
}
```

```
-- ==
-- input {
--   (10, 21, 21) (19, 12, 5)
-- }
-- output {
--   547
-- }
```

```
include Vec3

type vec3 = Vec3.Int.t
fun int main(vec3 a, vec3 b) = Vec3.Int.dot(a , b)
```

This test passes.

4.10.3 Testing rule for variable shadowing

The following tests are implemented to test whether the rule **??** holds: Simple shadowing for functions holds:
`shadowing0.fut`:

```
-- M1.foo() calls the most recent declaration of number, due to M0.number()
-- being brought into scope of M1, overshadowing the top level definition of
-- number()
-- ==
-- input {
-- }
-- output {
--  2
-- }

fun int number() = 1
struct M0
  {
    fun int number() = 2
    struct M1
      {
        fun int foo() = number()
      }
  }

fun int main() = M0.M1.foo()
```

Simple shadowing for types holds:
`shadowing1.fut`:

```
-- M1.foo() calls the most recent declaration of number, due to M0.number()
-- being brought into scope of M1, overshadowing the top level definition of
-- number()
-- ==
-- input {
-- }
-- output {
--  (6.0, 6, 6)
-- }

type best_type = float
fun best_type best_number() = 6.0
struct M0
  {
    type best_type = int
    fun best_type best_number() = 6
```

```
    struct M1
      {
        fun best_type best_number() = 6
      }
  }

fun (float, int, int) main() = (best_number() , M0.best_number() , M0.M1.
   best_number())
```

This test shows, that structures are only read into scope, after they are fully parsed. `shadowing2.fut`:

```
-- M0.foo() changes meaning inside M1, after the previous declaration of M0
-- is overshadowed.
--
-- ==
-- input {
-- }
-- output {
--  12
-- }

struct M0
  {
    fun int foo() = 1
  }

struct M1
  {
    fun int bar() = M0.foo()
    struct M0
      {
        fun int foo() = 10
      }
    fun int baz() = M0.foo()
  }

fun int main() = M0.foo() + M1.bar() + M1.baz()
```

and `undefined_structure_err0.fut`:

```
-- We can not access a struct before it has been defined.
-- ==
-- error: .*Unknown.*

fun int try_me() = M0.number()
struct M0
  {
    fun int number() = 42
  }

fun int main() = try_me()
```

4.11 Results

The implementation of structures works, and I am satisfied with the results. As shown in the tests, the structures behave as prescribed in the rules4.2.

4.12 Future work

As mentioned in **??**, it is possible to trigger a `DuplicateDefinitionError` from including files into the main program.

Given more time, I would like to extend the Futhark `include` statement to support an `as` statement, so that the inclusion

```
include some_module as M0
```

will include the declarations from `some_module` into the futhark program, but not into the top level declarations. Instead, the declarations will be loaded into a structure `M0`, which can then be accessed throughout the rest of the program as any other module.

Language part	language construct	definition	example
Core:			
	dec ::=	`type` *t* = *type_def*	`(* type Status = int *)`
	$\mid$	`fun` *f args* = *exp*	`(* fun foo n = n + 5 *)`
	$\mid$	`val` *x* = *exp*	`(* val eleven = 11 *)`
Module:			
	topdec ::=	*sigdec*	
	$\mid$	*moddec*	
	$\mid$	*topdec topdec*	
	sigdec ::=	`signature` *X* = *sigexp*	`(* signature foo = ... *)`
	sigexp ::=	`sig` *sigspec* `end`	`(* sig val bar : int end` `*)`
	$\mid$	*X*	`(*{foo, bar, One, ... }*)`
	sigspec	`val x :` type	`(* val bar : string *)`
	$\mid$	`module` *X* `:` *sigexp*	`(* module Bar :` `Numberable *)`
	$\mid$	sigspec sigspec	
	moddec ::=	*dec*	
	$\mid$	`module` *X* = *modexp*	`(* module One = ... *)`
	$\mid$	moddec moddec	`(* val x = 1 val y = 2 *)`

	modexp ::=	`struct` *moddec* `end`	`(* struct`
			`module Adder = PlusOp`
			`val one = 1`
			`end *)`

	$\mid$	*modexp* `:` *sigexp*	`(* struct`
			`val bar = 1`
			`end : foo *)`

	$\mid$	*X*	`(* {One, Numberable,`
			`Counter, Queue, ... }`
			`*)`

Module access:			
	LongIdent ::=	*module*`.`*field*	`(* One.bar *)`
	$\mid$	*module*`.`*LongIdent*	`(* Numberable.One.bar *)`