Modules in Futhark
Bachelor's thesis

Mikkel Storgaard Knudsen

June 13, 2016

Abstract

This report investigates the benefits and possibilities of implementing a module system in the programming language Futhark. On the basis of the Standard ML module system, both a type aliasing system and a module structure system is succesfully designed (and defined with interference rules), and implemented in Futhark. Both type aliases and structures are tested in up against their interference rules, to confirm that they indeed adhere to their rules. Finally, tentative implementations of module signatures- and functors are proposed on the background of the work and experience gained during the two first feature implementations of the project.

Contents

1 Introduction

This report describes the efforts in defining a concrete method of extending the Futhark Programming Language with a module system.

A module system is here defined as a programming feature, which allows the programmer to package functionality into a discrete code package.

A module can then be shared and utilized in any other program, by including it in the source code. Widely known modules include `NumPy`[1], the Python package for data modelling and computing, and `NLTK`[2], the Natural Language Toolkit (also for Python.)

A well written module can become very popular. In some cases, a programmers choice of programming language for a given project, can be decided by the number and quality of available, project relevant modules.

The module system design for Futhark has been inspired by the module system implemented in Standard ML[1].

Not only does the Standard ML modules enable structures with predefined implementations. It also defines a system of *functors*[3], which defines abstract implementations of structures.

These abstract structures are applied on an argument of a fitting signature, returning a realized structure. This makes it possible to write structures that does not commit to one certain type, but instead inherits the functionality defined in the given argument.

This spares the developers for writing a lot of almost identical code. Refer to (Figure 4) for a simple example.

Reader expectations: The reader is expected to have experience with statically typed languages, functional programming, and preferable the Haskell programming language.

[1] `http://www.numpy.org/`
[2] `http://nltk.org`
[3] Not to be confused with neither mathematical functors or functors from Haskell

```
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 }}
```

Figure 1: A function polluted by all the references to `f32`

1.1 Motivation

The Futhark language a young language[4], and is in continous developement. Extending this language to support modules would increase the usability of Futhark, which could be a factor in expanding the user base and user retention of Futhark.

The following subsections contain the features this project is implementing, and examples of why these features are desirable.

1.1.1 Abstraction increases readability.

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. Take for example a sphere in three dimensional space: It has **1) a radius**, **2) a position** , and **3) a direction** that it is moving in.
Let us define a function that multiplies the speed of the sphere by a factor k: (Figure 1)

[4]It's official web site `http://futhark-lang.org` was launched in March 2016.

```
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}
```

Figure 2: Type aliasing abstracts type definitions into names that are more fitting for the function declaration at hand

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 sphere2 without being explicit about the actual types at each function call.

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.

1.1.2 Compartmentalization of functionality increases usability:

Splitting code functionality into multiple files, will allow the programmer to compile and type check these modules individually. The programmer can edit and contribute to these files independently of the programs, which includes these modules.

1.1.3 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 structures3, by introducing the concept of functors.
First we repeat the three-dimensional vector structure from earlier, but without declaring any particular primitive type (Section 2.1) as the contained type of the vector (Figure 3).

The structure3 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 applying the abstract structure on a concrete structure.

```
functor Vec3 ( number_type : NumType) {
      type t = number_type.t
      type vector = {t, t, t}
      fun vector add( ... ) = ...
      fun vector subtract( ... ) = ...
      fun vector multiply ( ... ) = ...
      fun vector divide ( ... ) = ...
  }
```

Figure 3: A functor defining a vector structure

```
struct Int {
  type t = int
} : NumType
struct IntVec3 = Vec3(Int)
```

Figure 4: The IntVec3 structure is an instance of Vec3.

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.

From a performance-concerned point of view, the module system is desirable, as the Futhark compiler handles all used and included structures at compile time.

1.2 Problem definition

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

1.2.1 Scope of project

The scope of the project is

- to define and implement a type aliasing system in Futhark
- to define and 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 research the possibility of implementing functor and signature functionality, so that Futhark supports the definition of abstract structures and concretizations of these structures. A suggested design for the implementation of functors should be part of this project.

1.3 Related work

Futhark modules are designed with Standard MLs module system in mind. Thus, *The Definition of Standard ML '97*[5] has been a source of inspiration for the behaviour of the Futhark module system.
However, I have not read the code implementation of SML modules. My code supervisor and I did not find it necessary to lean on the SML implementation of modules:
The Futhark compiler could readily support extensions to the language, and it was not difficult to figure out a way to implement the module system.

Implementing SML modules in other languages is not a unique idea:
Yaraw Amin[7] has shown a way of implementing SML modules in Scala language. However, he was able to exploit functionality that was already available in Scala, to create his module system.

Mainly, he was able to use Scala `functions` as functors to create new Scala `classes` (comparable to structures), just as Scala `traits` were used for implementing signatures.

Futhark had neither functions (for structures), structures or signatures at the beginning of the project, which made these necessary to add to the compiler itself. Therefore, the following project is not necessarily original research (as modules are a well known concept,) but an original implementation.

2 Type aliases

To make an implementation of functors5 in Futhark, it was necessary to implement type aliases first. Otherwise, it would not be possible to abstract a type away in an alias.

2.1 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:

$$
\begin{array}{rl}
\mathit{Type}: & \texttt{Primitive type} \\
\mid & (\mathit{Types}) \\
\mid & [\mathit{Type}] \\
\mid & \mathit{TypeAlias}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Types}: & \mathit{Type}\ ,\ \mathit{Types} \\
\mid & \mathit{Type}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Primitivetype}: & \texttt{UnsignedInteger} \\
\mid & \texttt{SignedInteger} \\
\mid & \texttt{FloatType} \\
\mid & \texttt{Boolean}
\end{array}
$$

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

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

2.1.1 Interference rules

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

Given an environment $\Gamma : \{FE, TE\}$, $Types = \{type\_id \to Type\}$, $Type \to \tau$[a] we can define the following interference rules for using type aliases:

$$\frac{\Gamma \vdash typedecl \Rightarrow \Gamma' \qquad \Gamma \vdash Type \to \tau \qquad \Gamma \nvdash type\_id \to Type}{\Gamma \vdash \texttt{type}\ type\_id\ =\ Type \Rightarrow \{\emptyset, \{type\_id \to Type\}\} \oplus \Gamma}$$

where

$$\begin{aligned} &\{\emptyset, \{type\_id \to Type\}\} \oplus \Gamma \\ \Rightarrow &Types(\Gamma) \leftarrow\ Types(\Gamma) \cup \{type\_id \to Type\} \end{aligned}$$

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 subsection2.2.

[a] We will let τ designate a primitive type in Futhark

2.2 Implementation

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

1. The alias $type\_id_i$ is not already declared[5] in the current local environment. I.e. the example below:

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

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

2. The alias $type\_id_i$ 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 $type\_id_i$. 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 subsection2.3.2.

   ```
   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 here2.2

2.3 Parsing a type alias

2.3.1 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.

2.3.2 Adding resolved types to scope

```
1   type TypeAliasTableM =
2   ReaderT (HS.HashSet LongName) (StateT Scope TypeM)
3
4  typeAliasTableFromProg :: [TypeDefBase NoInfo VName]
5                         -> Scope
6                         -> TypeM Scope
7  typeAliasTableFromProg defs scope = do
8    checkForDuplicateTypes defs
9    execStateT (runReaderT (mapM_ process defs) mempty) scope
10   where
11         findDefByName name = find ((==name) . typeAlias) defs
12
13         process :: TypeDefBase NoInfo VName
14                 -> TypeAliasTableM (StructTypeBase VName)
15         process (TypeDef name (TypeDecl ut NoInfo) _) = do
16           t <- expandType typeOfName ut
17           modify $ (addType name t)
18           return t
19
20         typeOfName :: LongName -> SrcLoc
21                    -> TypeAliasTableM (StructTypeBase VName)
22         typeOfName (prefixes, name) loc = do
23           inside <- ask
24           known <- get
25           case typeFromScope (prefixes, name) known of
26             Just t -> return t
27             Nothing
28               | (prefixes, name) `HS.member` inside ->
29                   throwError $ CyclicalTypeDefinition loc name
30               | Just def <- findDefByName name ->
31                   local (HS.insert (prefixes, name)) $ process def
32               | otherwise ->
33                   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 snippet2.3.2. `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 $type\_id_a$ leads to another alias $type\_id_b$[6], we add $type\_id_a$ 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.

2.3.3 Converting UserType to TypeBase

```
1  expandType :: (Applicative m, MonadError TypeError m) =>
2                (LongName -> SrcLoc -> m (StructTypeBase VName))
3             -> UserType VName
4             -> m (StructTypeBase VName)
5
6  expandType look (UserTypeAlias longname loc) =
7    look longname loc
8  expandType _ (UserPrim prim _) =
9    return $ Prim prim
10 expandType look (UserTuple ts _) =
11   Tuple <$> mapM (expandType look) ts
12 expandType look (UserArray t d _) = do
13   t' <- expandType look t
14   return $ arrayOf t' (ShapeDecl [d]) Nonunique
15 expandType look (UserUnique t loc) = do
16   t' <- expandType look t
17   case t' of
18     Array{} -> return $ t' `setUniqueness` Unique
19     _       -> throwError $ InvalidUniqueness loc $ toStructural t'
```

2.3.4 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.

[6] line 27-33

2.3.5 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.

2.4 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[7].

However, it was also necessary to write tests to specifically confirm, that the implementation respects the rules defined in 2.2. The original pull request for working type aliases can be found here: [3].

2.4.1 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
```

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

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

This program fails as expected.

2.4.2 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.

2.4.3 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.

2.4.4 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[8].

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.
The two nbody implementations, one with type aliasing and one without, are available in appendix 7.2 and 7.3.

2.5 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[9] has been declared, but not defined, until the containing structure is instantiated using a functor.
This has not been implemented yet.

[8] `https://en.wikipedia.org/wiki/N-body_problem`
[9] in the form of a *type_id*

3 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 [10], the user runs a risk of running into errors like `MulipleDefinitionError`[11], 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:

$$\begin{aligned} \mathit{LongName} : \quad & \texttt{prefix.}\mathit{LongName} \\ \mid \quad & \texttt{identifier} \end{aligned}$$

In Futhark we will be using string ids for declarations, and longnames for the accessing types and functions in structures.

3.0.1 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

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

$$\frac{\Gamma \vdash fundecl \Rightarrow \Gamma' \qquad \Gamma \nvdash fun\_id \rightarrow funexpr}{\Gamma \vdash \texttt{fun}\ \ fun\_id\ =\ funexpr \Rightarrow \{\{fun\_id \rightarrow funexpr\}, \emptyset, \emptyset, \emptyset, \emptyset\} \oplus \Gamma}$$

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 5: Rule for adding a function to the environment

$$\frac{\Gamma \vdash Decl_1 \Rightarrow \Gamma' \qquad \Gamma \nvdash id \rightarrow decltype\ Decl_2}{\Gamma \vdash \texttt{decltype}\ id\ =\ decl \Rightarrow \{id \rightarrow Decl\}}$$

Figure 6: A generalized rule for adding declarations to the environment

$\Gamma = (FE, TE, SIGE, STRUCTE, FUNCTE)$, where the additions are a type alias environment, a signature environment, a structure environment and a functor environment.
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 $\{strid \rightarrow \Gamma_{strid}\}$ is a mapping in $STRUCTE$. Any Γ_{strid} contains its own function-, typealias-, signature-, structure- and functor declarations.

3.1 Interference rules

Now that we can discern between the global 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 four new elements, we can define the addition of any of these four elements follow a generalized rule: Please note, that several declarations can actually share name *id*, as long as they don't share declaration type.

3.1.1 Interference rules for adding multiple declarations

Finally, we want to create a rule for which environment we get, when we state multiple declarations in a row. This rule is covers the run of an entire program, since we can set the initial declaration in the program as $Decl_1$, use rule 7 to handle the first declaration in the pair $Decl_1$ and $Decl_2$, and afterwards using the same rule again on $Decl_2$, $Decl_3$.
This procedure is repeated until the rule reaches the two last statements, $Decl_{n-1}$, $Decl_n$.

$$\frac{\Gamma \vdash Decl_1 \Rightarrow \Gamma_1 \qquad \Gamma \oplus \Gamma_1 \vdash Decl_2 \Rightarrow \Gamma_2}{\Gamma \vdash Decl_1, Decl_2 \Rightarrow \Gamma_1 \oplus \Gamma_2}$$

For any Γ, $\Gamma_a \oplus \Gamma_b \Rightarrow \Gamma_a \cup \Gamma_b$

Figure 7: The behaviour of multiple declarations in a row.

$$\frac{\Gamma \vdash strdecl \Rightarrow \Gamma' \qquad \Gamma \vdash strdecls \Rightarrow \Gamma_{strdecls} \qquad \Gamma \cup \Gamma_{strdecls} \vdash \Gamma_{strid}}{\Gamma \vdash \texttt{struct}\ strid\ \{\ strdecls\ \} \Rightarrow \{strid \rightarrow \Gamma_{strid}\}}$$

If there is a shared identifier in both Γ and $\Gamma_{strdecls}$, the definition in $\Gamma_{strdecls}$ is the definition in Γ_{strid}.

Figure 8: The behavior of constructing a structure.

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.

3.1.2 Rule for adding a structure to the local environment

The consequence of defining Γ_{strid} as a union of new declarations together with the old environment menas, that Futhark exhibits variable shadowing, where definitions in a structure are valid in its nested structures, unless they are redefined in the nested structure.

3.1.3 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
```

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:

$$\frac{\Gamma \vdash longname \rightarrow fun\_definition}{\Gamma \vdash letx = longname() \Rightarrow x :=\downarrow longname}$$

where

$$\downarrow longname \;=\; return\ getFromEnv(longname,\ \Gamma)$$

.

Figure 9: Resolving a longname in the environment

3.2 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.

3.2.1 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`.

3.2.2 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
\\
  checkProg' :: [DecBase NoInfo VName] -> TypeM [DecBase Info VName]
  checkProg' decs = do
    _ <- checkForDuplicateDecs decs
    (_, decs') <- checkDecs decs
    return decs'
```

3.2.3 Checking for duplicates

As described in rule 6, 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
```

3.2.4 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 figure10
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)
```

```
      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

Figure 10: An example of scope behaviour

3.2.5 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 3.1.
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.

3.2.6 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 rule8.

```
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)
```

checkMod first adds a "breadcrumb" of the structure's name (*strid*) to the local environment.

After the environment has been given its name, the internal declaration of the structure is read using `checkDecs`.

```
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
```

Figure 11: The Haskell code which resolves a longname

3.2.7 Resolving the application of a longname

Resolving the application for a longname is done through two short recursive functions. Figure 11 shows the implementation of the longname rule 9:

3.2.8 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 3.5 for a possible future solution.

3.2.9 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.

3.3 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.
Having writting the structure implementation myself, I have been able to consider my testing options. I have had the choice of writing either *white box* tests, *black box* tests, or both.

In the context of these futhark structures, white box tests are defined as tests that are designed by a programmer, who has intricate knowledge of the code implementation of the program features, he is supposed to test.
I.e., this means trying to write programs, which tests whether there can be ambiguities in the parsing of the program, or other general errors in the code.

Black box tests are written without knowledge of the code behind the program. In the context of Futhark structures, this limits the test writer to test, whether the rules which are specified in 3.1 holds in the actual implementation.

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

3.3.1 Testing rule for multiple declarations [...]

The following tests are implemented to test whether rule 6 hold; that we can have several declarations of the same name, as long as they don't share declaration type.
`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. This is a white box test to ensure, that the implementation handles type, struct and fun declarations in seperate tables.
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.

3.3.2 Testing structures can be called as expected

The following tests are implemented to test that calling structures works as defined in rule 9.
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.

3.3.3 Testing rule for variable shadowing

The following tests are implemented to test whether the rule 8 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()
```

3.4 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 rules3.1.

To answer the problem definition, it is definitely possible to add module functionality to Futhark, in terms of structure support.

The extensions to the Futhark compiler in the TypeChecker.hs-module have had perfect compatibility with the rest of the Futhark, as all of the type aliases, structures and so on, are dealt with in the TypeChecker, which returns a typechecked Futhark program to the Futhark internaliser.

Besides two small fixes in Internalise.hs, it has not been necessary to change any part of the Futhark compiler pipeline, after the TypeChecker step.

The original pull request for working type aliases can be found here: [2].

3.5 Future work

As mentioned in 3.2.8, 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.

Another potential effort worth of mentioning is the commence the definition of a standard library for futhark. Futhark already has built-ins mathematical functions like log10(float a), so defining a standard library of structures which define i.e. Integer-functions and Float-functions, could be the next step.

4 Signatures

Just like a type signature of a function is the definitions of the argument- and return types of a function, we can define module signatures similarly:

A module signature specifies the type signatures for a set of declarations.
Let us define a signature called `hasPlus`:

```
sig hasPlus {
  type t
  val plus : (t,t) -> t
}
```

Then, after this declaration, we can define some structure, and append this structure declaration with a signature:

```
include hasPlus

struct M0 {
  ...
} : hasPlus
```

With this signature, we promise that the structure `M0` matches the declarations in signature `hasPlus`. In the instance of `M0`, we now require, that `M0` has a declaration, where alias `t` is assigned.`M0` must also have a function, that takes two arguments of type `t`, and returns a value of type `t`.

In the following example, two structures are written. `M0` adheres to the specifications in `hasPlus` , but `M1` does not, which ultimately results in an error at compile time:

```
include hasPlus

struct M0 {
  type t = int
  fun t plus(t a, t b) = a + b
} : hasPlus

struct M1 {
  type t = (int, int)
  fun int plus(t tuple) =
    let (a, b) = tuple
    in a + b
} : hasPlus
```

4.1 Purpose of module signatures

By themselves, the ability to add a signature to a structure does not add any significicant usefulness to a language. Signatures are useful together with *functors*5, so I will refrain from doing further explanation outside of functor context.

4.2 Implementation

This section is not complete, as I have not implemented a complete and working signature system at this point.

We need a data type which represents a signature. I chose to define it as a table from some declaration name to a signature binding:

```
type Signature = HM.HashMap Name SigBinding

data SigBinding = FunSignature (Maybe (TypeDeclBase Info VName)) (TypeDeclBase
    Info VName)
                  -- ^ A function may take arguments, but might also be a
                  constant.
                  -- It always has a return type.

                | TypeSignature (TypeDeclBase Info VName)
                  -- A type signature must resolve to a type.

                | ModSignature LongName
                -- A structure of signature sig, can itself be demanded to
                -- contain a structure of some ModSignature
```

With the Signature defined, we need to extend the data type of the structure to accomodate the possibility of adding a signature to a structure:

```
data ModDefBase f vn = ModDef { modName :: Name
                              , modDecls :: [DecBase f vn]
                              , modDefLocation  :: SrcLoc
                              , modSignature :: Maybe LongName
                              }
```

The parsed structure from the futhark program now has either `modSignature Nothing` , or `modSignature (Just` *signature*`)`.

We update checkMod from 3.2.6 accordingly¸ by calling an extra function `verifyMod`:

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

verifyMod :: [DecBase Info VName] -> Maybe LongName -> TypeM ()
verifyMod _ Nothing _ = return ()
verifyMod decs (Just sig) loc = do
  sigtable <- asks (sigFromScope sig)
  case sigtable of
    Nothing -> throwError $ UndefinedLongName sig loc
    Just t ->
      foldM_ verifyDec decs $ HM.toList t
```

For each entry in the signature table, we call `verifyDec` on this entry to verify, that decs contain a declaration that satisfies the signature table.

As I have not implemented `verifyDec` completely yet, this implementation is not finished.

4.3 Results

As the signature implementation is *incomplete*, I cannot claim that it is possible to implement module signatures in Futhark, by referring to tests.
However, the current implementation lacks **only** a correct implementation of `verifyDec` before it works, barring some corrections of bugs that might show up during tests.
The ongoing work on this feature can be followed on the `signatures`-branch in the public Futhark GitHub repo.

To answer the question in the problem statement; whether it is possible to implement module signatures in Futhark:
Yes, it is.

5 Functors

A function takes a number of arguments of specified types, and returns a value of some return type. A **functor** takes a structure3 of a specified signature, and returns a structure of some signature.

The concept is best explained through an example:

```
--
--==
-- input {}
-- output { 4 }

sig NumType {
  type t
  val add : (t,t) -> t
  val subtract : (t,t) -> t
  val divide : (t,t) -> t
  val multiply : (t,t) -> t
}

struct Int {
  type t = int
  fun t add(t a, t b) = a + b
  fun t subtract(t a, t b) = a - b
  fun t divide(t a, t b) = a / b
  fun t multiply(t a, t b) = a * b

} : NumType

functor Doubler(number : NumType){
  type t = number.t
  fun t add(t a, t b) = number.add
  fun t double(t a) = add(a,a)
}

struct IntDoubler = Doubler(Int)

fun int main() = IntDoubler.double(2)
```

We first declare a signature NumType, which defines the number type. We declare it so that there is a type t, that has four standard operations (add, subtract, divide and multiply.).

We then declare a structure Int, which adheres to the NumType signature: The add function in Int uses the built-in functionality of the plus operator in Futhark, but could in principle be defined as whatever the programmer would like *a*s long as the type checks correctly.

Then, the functor is defined: The functor takes a structure as an argument, and can then refer to this structure in the declarations of the functor, using dot-notation. The functor contains a list of type-, function- and structure declarations, just like the structure does.

However, none of these declarations has to be defined inside the functor: any of the declarations can be loaded from the functor's argument instead.

In the example above, the Doubler functor contains a function called `double`. But `double` is not defined inside the functor, but will instead become whatever the `double` function is defined as, inside the `number` argument of the functor. At the application of the Doubler functor, creating IntDoubler, we can guarantee, that our types still hold. This is because the function definitions in the functor has an abstract type[12], which is not instantiated until the functor can get the type definition from the structure argument.

To boot, we can even bind a signature to functor if we want to. In that case, we will have to create a functor, that returns a structure of signature *sig*, when it is applied.

5.1 The reason for functors

As explained in the introduction of this report1.1.3, functors allows the programmer to write modules with abstract structures, defining some functionality. Functors does not give us actual polymorphism in our programs, but it *does* allow us to generalize functionality, when it is not tied to any specific type traits.

Riccardo Pucella demonstrates in his "Notes on Programming SML/NJ"[4, pp. 59-60], how to implement a data structure `Queue`, using a functor that takes a structure of signature `Stack` as an argument. The queue is a widely used data structure, which makes an abstract implementation an excellent alternative to implementing a queue for each data type we need.

5.2 Tentative implementation of functors

Implementing a simple functor which instantiates a structure on application, should not be very difficult.

On a base level, building a structure from a functor and its argument is a question of parsing the functor's declaration in the right order.
At the time of the functor declaration of some functor `functor Funct(`*argument* `:` *signature*`){` *`Functdecls`* `}` we do not know anything about the types of the declarations in *Functdecls*.

We cannot type check this functor yet, so we will instead build a functor, which contains similar declarations to what the structure contains. However, we allow for abstract declarations of both types, functions and structures, such as:

```
functor S(str : sig){
  type a = str.t
  fun a foo(a left, a right) = str.bar(left,right)
  struct M0 = str.mystruct
```

[12]type t

```
}
```

We also allow for function declarations with calls to abstract functions:

```
functor S(str : sig){
  fun int baz = 2 + str.function()
}
```

Thus we will not handle the functor any further, until it is attempted to be applied to a structure somewhere in the program. At the application of the functor on some structure *str*, *str* is assumed to be in scope.

Our goal now is to convert the applied functor into a structure. This will happen during checkDecs. After applying the functor, we will not have any need for the functor declaration. Instead, we need a ModDec in the rest of the program, because the Futhark internalises the program functions on basis of function declarations, which are read from the actual function declarations `FunDec`, which are stored either in the top level decl list, or in the declaration list contained in a Futhark structure.

In the following tentative implementation of functors in the Futhark typechecker, we have added a Functor definition to the Futhark syntax.
Furthermore, ModDef have been expanded to allow functor application, checkMod have been expanded to allow functor application, and checkFunctor has been added.

```
data ModDefBase f vn = ModDef { modName :: Name
                              , modDecls :: ModInternal f vn
                              , modDefLocation  :: SrcLoc
                              , modSignature :: Maybe LongName
                              }

data ModInternal f vn = ModDecls :: [DecBase f vn]
                      | Functor :: ( functorName :: LongName
                                   . argumentName :: LongName
                                   )

data FunctorDefBase = FunctorDef { functorName :: Name
                                 , functorArg :: LongName
                                 , functorArgSig :: LongName
                                 , functorDecls :: [FunctorDeclBase f vn]
                                 , functorDefLocation :: SrcLoc
                                 }

...

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

```
checkMod :: ModDefBase NoInfo VName -> TypeM (Scope , ModDefBase Info VName)
checkMod (ModDef name (Functor (functor, applicant) loc sig) =
  local (`addBreadcrumb` name) $ do
    applyFunctor functor applicant


applyFunctor :: LongName -> LongName -> TypeM (Scope, ModDefBase Info VName)
applyFunctor functorname structname =
  functor <- functorFromEnv functorname
  argScope <- scopeFromEnv structname
  verifyScope argScope $ functorArgSig functor
  (local (addModule argScope)) $ do
    decs <- resolveFunctorDecs $ functorDecls functor
    checkMod (ModDef structname (ModDecls decs) (functorSig functor) (
       functorDefLocation functor))
```

As our results from structures3.4 showed, that structures in Futhark behaved as we would expect, I will claim that this tentative solution works as well. applyFunctor doesn't do much else than resolving the abstract declarations of the functor, into a concrete set of declarations, which we DO know how to handle, using `checkDecs`.

6 Discussion of results

Both the type aliasing system and the basic structure implementation developed during this project, has been designed around a set of interference rules. Tests were designed to measure whether these rules were obeyed by the actual implementation of the type system. For both type aliasing and structures, the final iterations of program showed that the implementations *did* work as defined in the interference rules.

As my final implementations of the two first features failed neither any of the feature-specific tests, nor any test in the Futhark test suite, or any of the Futhark benchmark programs[6], I *will* claim, that it is possible to extend the Futhark language, and that I am able to succesfully add features to the code base.

These results suggests that the two remaining features should be quite possible to implement as well:
Although the module signatures were not implemented in time for this report, all the information needed to perform a signature check at compile time, is present. A correct implementation is indeed just a question of writing a function, that verifies the type signature of a declaration correctly.
The implementation of the functor functionality itself was not completed either, but a tentative implementation has been described in this report. As the tentative implementation has been described from a good background knowledge of the relevant parts of the Futhark implementation, I see no immediate reason, that this implementation should not work *b*arring a total redesign of the Futhark implementation, outside of my control.

6.1 Method

The developement of the features implemented during this project has generally followed three phases.
For each of the features, I have initially sat down with both my supervisor and my "code supervisor" to discuss both the behaviour, and the implementation of the features. After having decided on which features we would like each part of the project to exhibit, we have shortly discussed a strategy for implementing this.

After these initial design meetings, I have been implementing the designs in the Futhark compiler itself. As the extensions I have made to Futhark have been implemented all the way into the Futhark parser, the step from initial code changes to a test-driven developement has been slightly long.
In the beginning of this project, it took me several weeks from the first change to the program, until I had code that would compile again. This was mostly caused by me being very new to the futhark code base. Over time, this was definitely alleviated by me becoming much more confident in Futhark developement.
At a certain point in the developement cycle of each feature, the implementation reached

an almost finished state: The code would compile, and the initial parsing tests for the given feature would compile correctly. At this point, I could commence actual test driven developement:

My method was now to define tests that corresponded to the interference rules which were declared for the feature in question. Whilst the code *would* compile at this stage, I reiterated on the implementation to fix mistakes in the code. I.e. type checking would perform correctly in a test program, but the program would exhibit behaviour that was not defined by an interference rule, nor desirable in the context of program.
Any number of tests could be written to test a feature implementation, but I concentrated mainly on white box testing, which are attempts to write tests that breaks the implementation, or makes the implementation show undefined behaviour.

After completing the implementation of a feature, I would document the addition to the Futhark language in the Futhark language reference documents, and merge the git feature branch with master.

6.1.1 Alternatives to the chosen method

The implementation of Futhark modules was chosen to be done directly in the Futhark code base. However, we considered another option in the beginning of the project. Futhark modules could have been implemented as a prototype instead, using an intermediate compiler for the modules specifically.

Instead of writing modules directly in a Futhark source file, the user would write futhark modules in an intermediate language. This would then be compiled into a normal Futhark source file, without any structures, modules or type aliases.
The benefits of this method would have been a larger degree of freedom in the implementation: I would have been able to define my an intermediate language for Futhark modules, and I could have implemented the intermediate language in a language of my own choice, other than Haskell.

I decided against this, because seperating the module compiler from the Futhark compiler would demand continued maintenance on the module compiler, whenever the Futhark language definition changed.
Furthermore, I decided to develop the modules in Futhark itself, so I could benefit from Troels Henriksens knowledge of the Futhark code base, and get more comfortable with advanced functional programming concepts, such as monads.

6.2 Conclusion

Implementing a simple Standard ML style module system is not only theoretically possible, but should be within reach of the Futhark project, given a couple of weeks' more of devel-

opement time.
At the time of this report, it has not been finished solely due to time constraints.

6.3 Acknowledgements

First and foremost, I would like to thank my supervisor *Martin Elsman*, who have been helpful with advice, theoretical education in relation to defining environments, and the interference rules hereof.

I would like to thank *Troels Henriksen*, my "code supervisor" on this project. As Troels is one of, if not *the*, head developer of Futhark, he has been an invaluable source of sound design advice and code reviews on my Futhark development.

Finally, I like would like to thank *Niels Gustav Westphal Serup*, who has been helpful with giving me Haskell advice, at times where I have been stuck in developement.

nå, hej hej

7 Appendices

7.1 Futhark Module Language

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

7.2 nbody before type aliasing

```
 -- 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

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')
```

7.3 nbody using type aliasing

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'')
```

8 References

References

[1] Danny Gratzer. A Crash Course on ML Modules. http://jozefg.bitbucket.org/posts/2014-00-08-modules.html. Online; accessed 12.06.2016.

[2] Mikkel Storgaard Knudsen. Git commit containing complete structure feature. https://github.com/HIPERFIT/futhark/commit/7e39b9c759587e3f8a1d5c3b0de68b1400baa4c1. Online; accessed 12.06.2016.

[3] Mikkel Storgaard Knudsen. Original git commit containing complete type aliasing feature. https://github.com/HIPERFIT/futhark/commit/8c0034d6764a71482b50db7694c9bbc2ac4cf7b2. Online; accessed 12.06.2016.

[4] Riccardo Pucella. Notes on Programming SML/NJ. http://www.cs.cornell.edu/riccardo/smlnj.html. Online; accessed 12.06.2016.

[5] Robin Milner and Mads Tofte and Robert Harper and David McQueen. *The Definition of Standard ML*. MIT Press, 1997.

[6] Various. Futhark Benchmarks. https://github.com/hiperfit/futhark-benchmarks. Online; accessed 12.06.2016.

[7] Yawar Amin. Scala Modules. https://github.com/yawaramin/scala-modules. Online; accessed 12.06.2016.