Chapter 1

The **FShark** language

LANGUAGE REFERNCE

$$
\begin{array}{lcll}
t & ::= & \texttt{int} & \text{(Integers)} \\
 & | & \texttt{float} & \text{(Floats)} \\
 & | & \texttt{bool} & \text{(Booleans)} \\
 & | & (t_0 \times \ldots \times t_n) & \text{(Tuples)} \\
 & | & \{\texttt{id}_0 : t_0; \ldots ; \texttt{id}_n : t_n\} & \text{(Records)} \\
 & | & t \text{ array} & \text{(Arrays)} \\
 & & & \\
k & ::= & n & \text{(Integer)} \\
 & | & f & \text{(Float)} \\
 & | & b & \text{(Boolean)} \\
 & | & (k_0, \ldots, k_n) & \text{(Tuple)} \\
 & | & \{\texttt{id}_0 = k_0; \ldots ; \texttt{id}_n = k_n\} & \text{(Record)} \\
 & | & [k_0; \ldots ; k_n] & \text{(Array)} \\
 & & & \\
p & ::= & \texttt{id} & \text{(Name pattern)} \\
 & | & (p_0, \ldots, p_n) & \text{(Tuple pattern)}
\end{array}
$$

Figure 1.1: The FShark syntax

$$
\begin{array}{lcll}
e & ::= & k & \text{Constant} \\
 & | & v & \text{Variable} \\
 & | & (k_0, \ldots, k_n) & \text{(Tuple expression)} \\
 & | & \{\texttt{id}_0 = k_0; \ldots ; \texttt{id}_n = k_n\} & \text{(Record expression)} \\
 & | & [k_0; \ldots ; k_n] & \text{(Array expression)} \\
 & | & e_1 \odot e_2 & \text{(Binary operator)} \\
 & | & -e & \text{(Prefix minus)} \\
 & | & \texttt{not } e & \text{(Logical negation)} \\
 & | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{(Branching)} \\
 & | & v.[e_0] \ldots .[e_n] & \text{(Array indexing)} \\
 & | & v.\texttt{id} & \text{(Record indexing)} \\
 & | & v_0.v_1 & \text{(Module indexing)} \\
 & | & \texttt{let } p = e_1 \texttt{ in } e_2 & \text{(Pattern binding)} \\
 & | & v\ e_0\ \ldots\ e_n & \text{(Function call)} \\
 & & & \\
\mathit{entry} & ::= & \texttt{[<FSharkEntry>]} & \\
 & | & \epsilon & \\
 & & & \\
\mathit{fun} & ::= & \mathit{entry}\ \texttt{let}\ v\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t = e & \\
\mathit{typealias} & ::= & \texttt{type}\ v\ = t & \\
\mathit{module} & ::= & \texttt{module}\ v = \mathit{prog}'\ \mathit{progs}' & \\
 & & & \\
\mathit{prog} & ::= & \mathit{module}\ \mathit{prog} & \\
 & | & \mathit{prog}'\ \mathit{prog} & \\
 & | & \epsilon & \\
\mathit{prog}' & ::= & \mathit{typealias} & \\
 & | & \mathit{fun} & \\
\mathit{progs}' & ::= & \mathit{prog}'\ \mathit{progs}' & \\
 & | & \epsilon &
\end{array}
$$

Figure 1.2: The FShark syntax, expressions

$$
\begin{array}{lcll}
e & ::= & k & \text{Constant} \\
 & | & v & \text{Variable} \\
 & | & (k_0, \ldots, k_n) & \text{(Tuple expression)} \\
 & | & \{\mathtt{id}_0 = k_0; \ldots; \mathtt{id}_n = k_n\} & \text{(Record expression)} \\
 & | & [k_0; \ldots; k_n] & \text{(Array expression)} \\
 & | & e_1 \odot e_2 & \text{(Binary operator)} \\
 & | & -e & \text{(Prefix minus)} \\
 & | & \mathtt{not}\ e & \text{(Logical negation)}
\end{array}
$$

Figure 1.3: f binary operators

$$
\begin{array}{lcll}
e & ::= & k & \text{Constant} \\
 & | & v & \text{Variable} \\
 & | & (k_0, \ldots, k_n) & \text{(Tuple expression)} \\
 & | & \{\mathtt{id}_0 = k_0; \ldots; \mathtt{id}_n = k_n\} & \text{(Record expression)} \\
 & | & [k_0; \ldots; k_n] & \text{(Array expression)} \\
 & | & e_1 \odot e_2 & \text{(Binary operator)} \\
 & | & -e & \text{(Prefix minus)} \\
 & | & \mathtt{not}\ e & \text{(Logical negation)}
\end{array}
$$

Figure 1.4: FShark SOACs

1.1 The supported F# subset for FShark

A standard F# program automatically includes the `Microsoft.FSharp.Core` namespace, which contains the `Core.Operators` module. As `Core.Operators` contains all the basic operators and standard functions for, this is where the F# subset suitable for FShark compilation has been picked out.

1.1.1 F# operators available in FShark

Infix operators:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulo operation
`**`	Exponentiation
`&&`	Boolean `and`
`||`	Boolean `or`
`<`	Less
`<=`	Less-or-equal
`>`	Greater
`>=`	Greater-or-equal
`=`	Is-Equal
`<>`	Is-Not-Equal
`<|`	Left-apply: e_1 `<|` $e_2 \equiv e_1(e_2)$
`|>`	Right-apply: e_1 `|>` $e_2 \equiv e_2(e_1)$

Special operators:

`[`e_0`...`e_1`]`	Generate an array of numbers in the interval $[e_0, e_1]$.

1.1.2 F# standard functions available in FShark

`id`

The identity function

Chapter 2

The FShark Compiler and Wrapper

Introduction

Parsing and building a regular F# program is trivial when using official build tools like `msbuild` or `fsharpc`. But in the case of FShark, we are not interested in the final result from the F# compiler, but merely its half-finished product.

As the F# Software Foundation offers the official F# Compiler as a freely available NuGet package for F# projects, we can use this package `FSharp.Compiler.Services` to parse the entire input FShark program and give us a Typed Abstract Syntax Tree of the FSharp expressions therein.

FIGURE HERE OF THE USUAL FSHARP COMPILATION

As the F# Software Foundation actively encourages developers to create projects using the F# compiler library, they have published the collected F# compiler as a NuGet package, alongside a tutorial**??**on the usage of the various compiler parts.

For FShark, the Compiler Services package is used to compile a Typed Abstract Syntax Tree from a valid FShark source code file, which we then convert into- and print as a valid Futhark program. The Typed Abstract Syntax Tree is merely an AST that already has tagged all the contained expressions with their respective types.

We'll start with a detailed explanation of the FShark Compiler Pipeline.

2.0.1 The FShark Compiler Pipeline in practice

To examine the compiler pipeline in action, we'll go through the motions with the small example program displayed in figure 2.3. We begin by constructing an instance of the FSharkWrapper. It has the following mandatory arguments:

`libName`
: This is the library name for the FShark program. In the final Futhark `.cs` and `.dll` files, the main class will have the same name as the `libName`. This doesn't really matter if FShark is just used as a JIT compiler, but

it's good to have a proper name if the user only wants to use the compiler parts of FShark.

`tmpRoot`
: The FShark compiler works in its own temporary directory. This argument must point to a directory where F# can write files and execute subprocesses (Futhark- and C# compilers) which also has to write files.

`clooPath` and `monoOptionsPath`
: The C# compiler needs the `Cloo`- and the `Mono Options` libraries available for the compilation, and the finished FShark .dll file also needs these two libraries available. To ensure their availability, the FSharkWrapper requires these paths at the beginning of the process, so it can pass them on later in the process.

`preludePath`
: The FShark compiler needs the FShark prelude available to compile FShark programs.

`openCL`
: Although Futhark (and therefore FShark) is most effective on OpenCL-enabled computers, the benchmarks in **??** still show a significant speed increase for non-OpenCL Futhark over native F# code. Therefore, FShark is also available for non-OpenCL users. Use this flag to tell FShark whether Futhark should compile C# with or without OpenCL.

`unsafe`
: For some Futhark programs, the Futhark compiler itself is unable to tell whether certain array operations or SOAC usages are safe, and will stop the compilation, even though the code should (and does) indeed work. To enable these unsafe operations, pass a `true` flag to the compiler.

Now, we can pass a source file to the FShark wrapper, compile[1] it and load it into the FShark wrapper object.

To use the compiled FShark function, we must first wrap our designated input in an `obj array`. In this case, our chosen FShark function takes one argument, an `int array`. We define this array, and construct an argument array containing this single element. If the FShark function takes two arguments, we define an input `obj array` with two elements, and so forth. It is important to declare the input array as an `obj array`. Otherwise, F#s own type checker might very well faultily infer the input array as something else. In this particular case, `input` would've been inferred as being an `int array array`, until we declared its type specifically.

We then invoke the desired function through the wrapper. As all reflection-invoked functions return a value of type `obj`, we need to downcast this object manually. In this example, we use F#s downcast operator `(:?>)` to declare the return value as an `int array`. The actual return type is always the same as the return type declared in the source FShark file.

[1]See subsection **??**

2.0.2 When FShark Wrapper Compiles

The general way to compile and load an FShark program into the FShark Wrapper, is by adding FShark source files to the wrapper object by calling the `AddSourceFile` method, and followingly calling the `CompileAndLoad` method. Although the FShark wrapper also offers other methods of loading and compilation, this is the primary one, as it initiates the entire FShark compilation pipeline.

When calling `CompileAndLoad`, the supplied FShark source files are concatenated into one long source file, and written to a temporary location. An FSharpChecker is then initialized, so we can parse and type check the concatenated source code. The FSharpChecker is a class exported by the FSharp Compiler Services, and is a class that lets developers use part of the F# compilation pipeline at runtime.

We supply the FSharpChecker with the path to our precompiled FSharkPrelude assembly, and then call its `ParseAndCheckProject` method on to receive an assembly value, which contains the complete Typed Abstract Syntax Tree of our FShark program, in the form of an `FSharpImplementationFileDeclaration`.

If the FShark developer followed the guidelines to write a well-formed FShark module, the main declaration of the program, the `FSharpImplementationFileDeclaration`, should contain a single `FSharpEntity`, which in turn contains all the remaining declarations in the program.

The declaration types within F#'s Typed AST

The `FSharpImplementationFileDeclaration` type has three union cases.

`InitAction of FSharpExpr`

`InitActions` are `FSharpExprs` that are executed at the initialization of the containing entity. These are not supported in FShark.

`Entity of FSharpEntity * FSharpImplementationFileDeclaration list`

An `Entity` is the declaration of a type or a module. In the case of FShark, three different kinds of entities are supported:

- **FSharpRecords** are standard record types, and can be translated to Futhark records with ease. This entity has an empty `FSharpImplementationFileDeclaration list`.
- **FSharpAbbreviations** are type abbreviations, and are easily translated into Futhark type aliases. This entity has an empty `FSharpImplementationFileDeclaration list`.
- **FSharpModules** are named modules which contains subdeclarations. In this case, we retrieve the subdeclarations from the `FSharpImplementationFileDeclaration list`. The FShark compiler supports building FShark modules, but current limitations demands that modules are flattened when compiled to Futhark. This also means that function name prefixes in function calls are stripped when compiled to Futhark.

$[\![Int8]\!]$	=	`Prim Int FInt8`	$[\![Int16]\!]$	=	`Prim Int FInt16`
$[\![Int32]\!]$	=	`Prim Int FInt32`	$[\![Int64]\!]$	=	`Prim Int FInt64`
$[\![UInt8]\!]$	=	`Prim UInt FUInt8`	$[\![UInt16]\!]$	=	`Prim UInt FUInt16`
$[\![UInt32]\!]$	=	`Prim UInt FUInt32`	$[\![UInt64]\!]$	=	`Prim UInt FUInt64`
$[\![Single]\!]$	=	`Prim Float FSingle`	$[\![Double]\!]$	=	`Prim Float FDouble`
$[\![\tau[]]\!]$	=	`FSharkArray` $[\![\tau]\!]$	$[\![(\tau_0 \times \ldots \times \tau_n)]\!]$	=	`FSharkTuple` $[\![\tau_0]\!], \ldots, [\![\tau_n]\!]$

INSERT NOTE ON RULE FOR TUPLE ('a [] * long [])

Figure 2.1: F# (.NET) types to FSharkIL types

$[\![Const(obj, \tau)]\!]$	=	`Const(`$obj, [\![\tau]\!]$`)`	$[\![i]\!]$	=	i	$[\![f]\!]$	=	f
$[\![c]\!]$	=	$ascii(c)$	$[\![$`tt`$]\!]$	=	`true`	$[\![$`ff`$]\!]$	=	`false`

Figure 2.2: FShark SOACs

`MemberOrFunctionOrValue ofFSharpMemberOrFunctionOrValue * FSharpMemberOrFunctic`

F# doesn't differ between functions and values, which means that a function is merely a value with arguments. A pattern matched `MemberOrFunctionOrValue` value has the form `MemberOrFunctionOrValue (v, args, exp)`, where `v` contains the name and the type of the variable. If the `args` list is empty, `v` is simply a variable. If not, `v` is a function. `exp` contains the `FSharpExpr` that `v` is bound to. An `FSharpExpr` can be anything from a numeric constant to a very long function body.

2.0.3 FSharp-to-FSharkIL rules

INTRODUCTION HERE For these translations, we will disregard that all `FSharpExprs` are union cases of the F# data type `BasicPatterns`.

In figure 2.4 we see a small but valid FShark program. It reads like a regular F# program, but contains the three vital parts that makes it usable as an FShark program.

- The module declaration on the first line declares that the following code is inside a module. In this case, we are declaring the module `ExampleModule`, although we could use any valid F# module name. As shown in figure 2.5, the top module declaration falls away during compilation, so only the top module contents are left.
- This `open` statement ensures that the F# Compiler Services has access to the FSharkPrelude during the compilation. It is possible to write an FShark program which doesn't use the FSharkPrelude, but this removes access to the SOACs that we use to write our data parallel programs.
- The `[<FSharkEntry>]` attributed function `TimesTwo` ensures that the resulting Futhark library from the FShark compiler has at least one entry point function. Without any entry point functions, we won't have any functions in the final compiled FShark program.

In figure 2.5 we see the resulting Futhark program. For now, we will ignore the transformations that have happened, except for two things: The `Map`

```
module FSharkExample
open FShark.Main

[<EntryPoint>]
let main argv =
  let wrapper =
    new FSharkWrapper(
      libName="ExampleModule",
      tmpRoot="/home/mikkel/FShark",
      clooPath="/home/mikkel/Cloo.clSharp.dll",
      monoOptionsPath="/home/mikkel/Mono.Options.dll",
      preludePath= "/home/mikkel/Documents/fshark/F-
       ↪  SharkPrelude/bin/Debug/FSharkPrelude.dll",
      openCL=true,
      unsafe=true
      )

  wrapper.AddSourceFile "../../srcs/ExampleModule.fs"
  wrapper.CompileAndLoad
  let xs = [|1;2;3;4|]
  let input = [|xs|] : obj array
  let xs' = wrapper.InvokeFunction "MapPlusTwo" input :?>
   ↪  int array
  printfn "Mapping (+2) over %A gives us %A" xs xs'
  0
```

Figure 2.3: An F# program using FShark

```
module ExampleModule
open FSharkPrelude

module SomeValues =
  let Four : int = 4

  let SomePlus (x : int) (y : int) : int = x + y

[<FSharkEntry>]
let TimesTwo (x : int) : int =
  SomeValues.SomePlus x x

[<FSharkEntry>]
let MapPlusTwo (xs : int array) : int array =
  Map ((+)2) xs

let PlusSeven (x : int) : int =
  SomeValues.SomePlus x 7
```

Figure 2.4: A valid FShark program

```
let Four : i32 = 4i32
let SomePlus (x : i32) (y : i32) : i32 =
  ((x i32.+ y))
entry TimesTwo (x : i32) : i32 =
  unsafe SomePlus(x) (x)
entry MapPlusTwo (xs : []i32) : []i32 =
  unsafe map (let x = 2i32 in
              (\(y : i32) -> ((x i32.+ y)))) (xs)
let PlusSeven (x : i32) : i32 =
  SomePlus(x) (7i32)
```

Figure 2.5: A valid FShark program, compiled to Futhark

function (called from FSharkPrelude) has been rewritten as the plain Futhark SOAC `map` in lowercase, and the module SomeValues has been flattened (see sec **??** for future plans.)

This Futhark program is then stored in a temporary location in the user's file system, and compiled into as a library, using Futhark's C# compiler, either with or without OpenCL support. Finally after this compilation, we can invoke the resulting .dll file from within the FShark-using F# program.

2.0.4 Building FShark from the Typed AST

FShark supports a subset of the F# language, which also means that only a subset of F#'s FSharpExpr

Only the supported FSharpExpr's has been listed here, but the full range of FSharpExpr's are available on [**?**].