# FSCL

F# to OpenCL Compiler and Runtime

Gabriele Cocco
Ph.D. Student
Computer Science Dept.
University of Pisa

# Contents

# Glossary

**F# kernel** : a module function, lambda function, static or instance method that implements and OpenCL computation in F#. This is the source of compilation that produces valid OpenCL C99 source code.

**AST** : Abstract Syntax Tree, a data-structure that stores an high-level representation of the code.

**Compiler component** : either a step, a processor or a type handler of the compiler.

**Configuration object** : an instance of the CompilerConfiguration class. It contains all the information needed to create a specific compiler pipeline.

**Configuration source** : the input of the configuration process, which leads to the creation of a configuration object.

**Components source** : a container (DLL library) of compiler components.

**Native component** : a step or a processor belonging to the default (native) compiler pipeline.

**Reflected function/method** : a function of method marked with [ReflectedDefinition] attribute, which informs the .NET runtime to produce and maintain the AST the function/method body at runtime.

**Target AST** : the AST resulting from the transformation step. This AST is the closer representation of the target OpenCL code.

**Target types** : the set of .NET types allowed to appear in the target AST. This is also the set of types for which it exists a type handler capable of producing a valid target code (string) representation of the type.

**Type handler** : an entity responsible to declare a subset of valid target types and to produce their target code (OpenCL C99) representation.

# Chapter 1

# Introduction

Nowadays, OpenCL is one of the most popular programming frameworks for high-performance computing over multicore and many-core devices. Thanks to OpenCL, different parallel architectures coming from an heterogeneous set of brands and companies can be programmed in an homogeneous, standard way. Nevertheless, OpenCL lacks of some aspects and facilities characterizing high-level parallel programming frameworks and, more generally, high-level languages, which makes parallel programming still difficult, error-prone and hardly portable. In the last few years there have been many activities toward raising abstraction over OpenCL, introducing facilities to write parallel code and enhancing portability of features and performances. In this document we present *FSCL*, an F# to OpenCL open compiler aimed at raising abstraction over OpenCL programming allowing programmers to express OpenCL kernels inside F#.Net with all the benefits of integrated development, code completion and error reporting. Moreover, working over a Virtual Machine (VM) allows to automatize most of the effort OpenCL programmers must spend in coding the host-side part of parallel computations, which means creating context, devices, buffers and coordinating the execution of the computation on the device.

# Chapter 2

# FSCL Compiler

At a glance, the FSCL compiler infrastructure provides two main features:

1. It exposes an object model to write OpenCL kernels in F#;
2. It compiles F# kernels into valid OpenCL C99 kernel sources.

Instead of producing device-dependent code (x86, AMD) or intermediate representation (AMD IL, PTX), the compiler generates OpenCL C99 source code. Since the OpenCL specification is a standard, the FSCL compiler is completely independent from the device brand and model. Once a valid source OpenCL kernel source has been generated, the programmer can rely on the OpenCL compiler released by a specific device manufacturer (e.g. the Intel OpenCL compiler) to produce executable code. Alternatively, programmers can exploit the *KernelRunner* project, built on top of the FSCL compiler, to abstract from this task and make OpenCL-to-device code compilation and kernel execution completely transparent.

## 2.1 Compiler infrastructure

The FSCL compiler infrastructure is made of four main components: *steps*, *processors*, *type handlers* and the *kernel module*.

### 2.1.1 Steps and processors

The compiler pipeline is composed by a set of *steps*, each of which is made of a set of *processors*. As the name suggests a step is a sequential, independent processing action of the compiler pipeline. For example, parsing, preprocessing, Abstract Syntax Tree (AST) transformation and pretty printing are steps. To do something meaningful, a step requires a set of processors each of which performs a subset of the computation provided by the step. For example, the code generation step is responsible or producing the target representation of the code. Each processor of this steps handles the code generation of a particular AST node (if-then-else, while-loop, variable declaration, etc).

The transformation step is instead reponsible of manipulating the AST to produce an abstract representation that is closer to the OpenCL language. Each processor of this step performs the transformation of a particular construct.

The steps of the compiler pipeline are executed sequentially, from the first to the last one. In other terms, the steps orchestration is sequential. While steps are orchestrated in a fixed way, the processors orchestration depends on the particular step considered. Some steps test their processors against the input and select the first processor that claim to be able to handle the input itself. Some other steps execute the set of processors sequentially in the same way the pipeline executes steps. For example, the transformation step organizes its processors sequentially and executes them from the first to the last one. Unlike transformation, the code generation step orchestrates its processors in a recursive descent way: to produce the target code of an AST node, a processor recursively invokes the step to obtain the code relative to the children of the node and finally composes the results.

The orchestration establishes the model used to run the set of processors inside a step or a set of steps inside a pipeline. As already told, while the steps orchestration is fixed, the processors orchestration depends on the particular step the processors belong to. However, in both the cases the orchestration doesn't specify the particular order in which the set of processors or steps are executed but only the shape of execution (all the processors, the first suitable, recursive-descent, etc.). Even the pipeline sequential orchestration doesn't specify in which order steps are executed, but only that the entire set of steps is executed sequentially. The order of execution of steps and processors is determined from the *dependencies* that each processor or step declares to have.

To express inter-processor and inter-step dependencies FSCL employs .NET custom attributes. Each step and processor of the compiler pipeline is characterized by the following set of information needed to determine the execution order.

**ID** : a globally unique step/processor identifier;

**After** : the step/processor dependencies, represented by a list of IDs of steps/processors that must be executed before the current one;

**Before** : a list of IDs of steps/processors that must be executed after the current one.

In most of the cases the dependency list is enough to order processors/steps and *Before* is left unspecified. Nevertheless, this information is needed to guarantee the highest flexibility to the compiler plugin system (section 2.5).

Given the set of meta-information about steps and processors, the compiler plugin builder is capable of building a pipeline where the execution order of steps and the one of the processors inside each step respect the inter-step and inter-processor dependencies. For each step, processors are partially ordered so that for each processor A whose Before is C and whose After is B is executed after B and before C. The same applies to the steps of the pipeline. Whereas dependencies are too strict or wrong to be respected by the processors/steps[1] the pipeline fails in determining a valid partial order and reports

[1]For example, there is a circular dependency

an configuration error.

### 2.1.2 Types and type handlers

The set of .NET types is much wider than the set of valid OpenCL C types. In addition, OpenCL allows only to extend the set of types involved in a kernel compuation defining structs and respecting a set of particular constraints, while .NET gives a complete freedom declaring new types. For these reason, we can identify two set of types involved in the F#-to-OpenCL compilation process: *source types* and *target types*. The source types are all the types programmers can employ inside F# kernels. During the compilation, many of these types are transformed into different types to obtain a type-system that is closer to the OpenCL one. For example, since OpenCL doesn't allow to use multi-dimensional arrays (pointers-to-pointers), during compilation every multi-dimensional array is replaced with a proper unidimensional array, manipulating indexing expressions to guarantee the computation correctness. F# records and ref variables are another example of source types that are transformed during compilation. Records are replace with structs and ref variables with singleton arrays.

At the end of this kind of transformation, the set of types employed in the representation of the code is restricted to a subset of the original one, which is what we called *target types*. Type handlers comes into play to produce the target OpenCL code for this set of types. In other terms, a type handler is an entity responsible to declare a set of valid target types and to produce the proper OpenCL string representation for each of them. For this reason, type handlers are mainly used in the latest compilation steps (codegen), where the target code representation is produced.

In the current version of FSCL type handlers are one of the most important part of the framework not only because they contribute to the code generation but also because they catch most of the errors resulting from mistakes of programmers writing kernels. In fact, when F# kernel developers declare variables or define expressions of some type for which it doesn't exist any associated type handler, a type error is reported and the compilation fails. The FSCL type-type handler association and the exception mechanism allows to detect the 90% of the F# kernels errors.

### 2.1.3 Kernel module

The whole pipeline works on a inter-step data-structure, called *kernel module*, that is created during parsing and progressively filled by the rest of the pipeline. To create a new kernel module, the parser must provide what is called a *kernel source*, constituted by a signature and an AST. Different parser processors are capable of inspecting (parsing) different objects, such as method references and lambda functions, but all of them must output a Kernel Source, that is, provide a signature and a AST representing the body of the kernel function. The kernel source is only one of many other information contained in the kernel module and it is the only one needed to instantiate the module. The other information are listed below.

**Source** : this is the kernel source, constituted by a pair (signature, body);

**Kernels** : the set of kernel instances that will participate to the compilation. A kernel instance is a kernel source where all the generic parameters are *instantiated* using a particular subset of the supported types (i.e. int, float, double, etc.). In case of a non-generic source, one only kernel instance participates to the compilation[2];

**Functions** : the set of "utility" functions used by one or more kernel instances;

**Directives** : the list of directives that need to be considered while producing the target code. An example is *#pragma OPENCL EXTENSION cl_khr_fp64: enable*, which is a directive used to enable double precision computations;

**Global types** : the set of non-primitive types used inside a kernel or an utility function, such as structs or records;

**Custom info** : a (String, Object) dictionary used to inject additional information inside the kernel module.

### 2.1.4 Native components

The FSCL compiler is shipped with a set of default steps, processors and type handler to compile a wide variety of kernels. Programmers are free to extend this set to handle additional F# features to be used inside kernels or to replace the functionality of a particular step or processor. The compiler extension mechanism is discussed in section 2.5.3. The following figure illustrates the organization of the default pipeline steps.

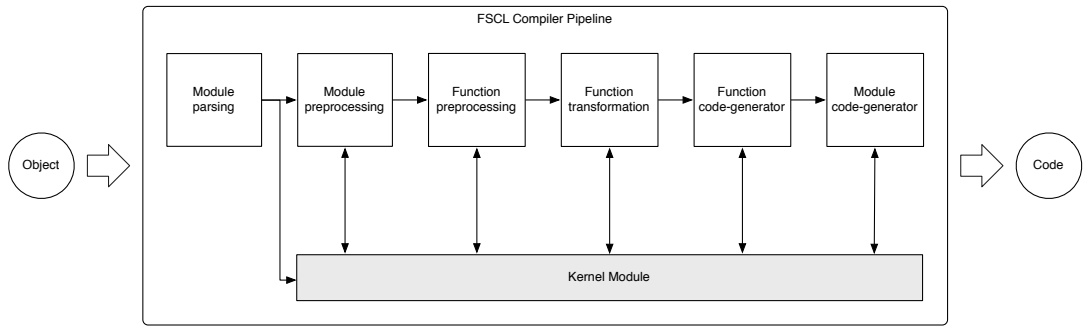


Figure 2.1: Native FSCL compiler pipeline

[2]In this case, the kernel instance matches the kernel source.

As shown, some steps are *module-wise*, the others are *function-wise*. A module-wise step (fig. 2.2) orchestrates its processors once and each processor can access the whole kernel module. A function-wise step (fig. 2.3) executes the set of its processors independently on each function and kernel instance[3]. Each function-wise processor can only access only the function or kernel instance the step is currently processing. After having executed the set of processors on each function and kernel stored into the module, the function-wise step ends.

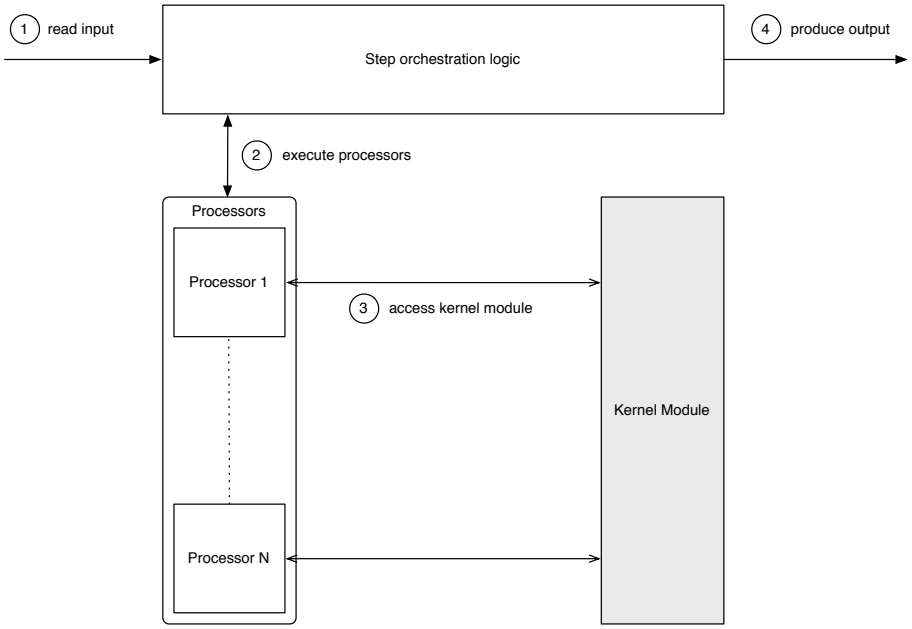


Figure 2.2: Module-wise step

### Module parsing and preprocessing

Module parsing is the step responsible to recognize a kernel and to extract it from an input object. The input is an object, the output a kernel module instantiated with the proper kernel source. The module parsing step organizes its processors following a *first-*

[3]From the step point of view, kernels and utility functions are represented using the same object type. Nevertheless, a function-wise step/step processor is always capable to determine if the processed element is a kernel or an utility function

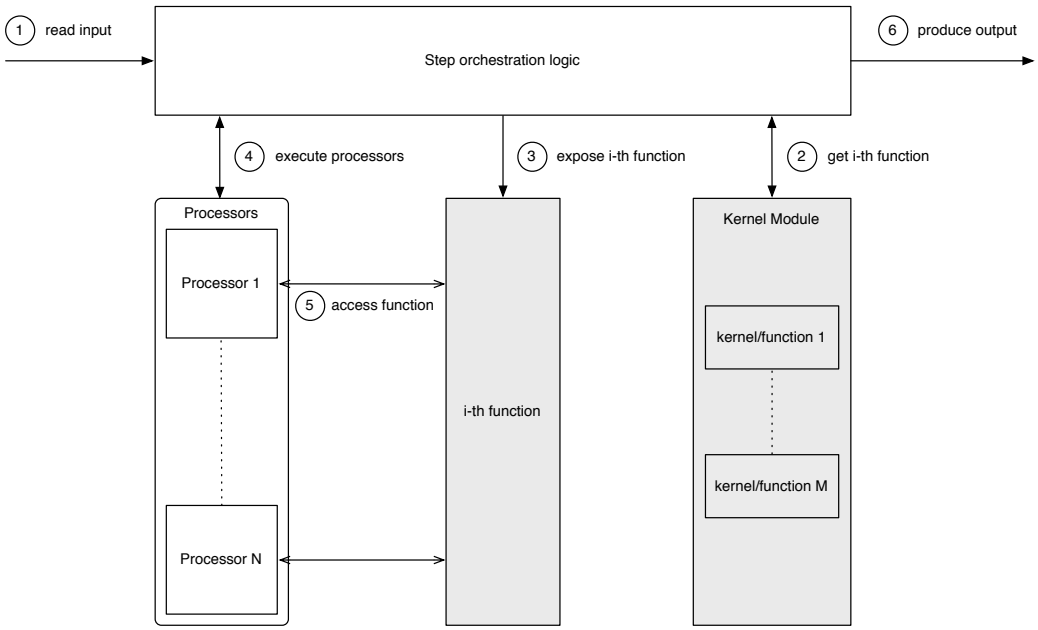


Figure 2.3: Function-wise step

*applicable* rule. One by one, the processors are tested against the input object as soon as the step finds a processors that claim to be capable of parsing the object. The default pipeline assigns three processors to this step, one capable of parsing method/function references, one responsible of parsing MethodInfo objects[4] and the last associated to lambda functions. The set of these three processors allows to express F# kernels in a wide variety of ways: as static methods, instance methods, module functions or "on-the-fly" lambda functions.

The module-wise preprocessing is a step executed right after the parsing. The purpose of this step is to analyze the "raw" module resulting from parsing and to set up the environment to make it possible for the successive steps to work independently on each kernel or function in the module. The set of processors are executed sequentially from the first to the last one. The most relevant processors belonging to this step are respectively called *function discover* and *struct discover*. The first detects references to reflected functions inside the kernel body, the second looks for usage of structured types. The collected information are stored inside the kernel module for the successive codegen step.

#### Function preprocessing and transformation

The function preprocessing step executes all the necessary actions aimed at preparing the evironment for the function transformation step. The most important action taken in this step is to analyze the F# kernel signature and to produce all the additional parameters nedeed to index the elements of the vector parameters in a OpenCL C kernel. The processors of this step are executed using the same strategy of the module preprocessing step, that is, they are sequentially executed from the first to the last one. Since this is a function-wise step, the execution strategy is applied independently to each function and kernel stored into the module.

After a kernel/function has been preprocessed, the function transformation step is executed to transform the AST represeting the F# kernel into an AST that is closer the avstract representation of the target OpenCL C code. For example, since OpenCL C doesn't support multi-dimensional arrays, the transformation step replaces all the instances of multi-dimensional arrays to proper instances of linear arrays, mapping the respective index expressions. Other kind of transformations affect the F# records, replaced with .NET structs. After the function transformation step has completed, the AST is suitable to be processed by the codegen step to produce OpenCL C code.

#### Function and module codegen

The function codegen step analyzes the signature and the body of a kernel/function to generate the related OpenCL C code. Once all the kernels and functions has been processed by the function codegen step, the module codegene step is executed to assemble these pieces of code into a single source. In addition, the module codegen is responsible

[4]MethodInfo is a .NET class which contains a wide set of meta-information regarding a mathod or function, such as the name and the type of each parameter.

of producing C pragmas[5] and the declaration of global types (e.g. structs deriving from F# records).

## 2.2 Supported features

The FSCL compiler native pipeline currently supports all the features of OpenCL specification v1.2 except for the image subset. The part of OpenCL v1.2 supported also includes vector types, such as *float4*, *int2* and so on. These types are part of the FSCL object model and can be used in the same way OpenCL C99 programmers are used to, accessing a subset of the vector components (e.g. var.xyzw) and using algebra operators (e.g. sum to *float4* variables).

In addition, the native pipeline supports higher-level language features to enhance programming abstraction and productivity, such as generic parameters, F# ref variables, structures and records. The list of enhanced features currently supported is listed below:

**Automatic array length** : when coding OpenCL C kernels working on arrays, the length of each array often must be explicitely passed as an additional parameter. Thanks to the power of reflection and dynamic method construction of .NET, programmers can code F# kernels without the need of passing the length of each input/output array. Whenever the length of an array is required in the kernel body, programmers can use the well-known *Length* property exposed by the .NET array type. The FSCL compiler is capable of genenrating additional parameters to host the length of each input/output arrays and to replace each usage of the *Length* property with a reference to the appropriate additional parameter;

**Ref variables** : ref variables can be used as kernel parameters to used to pass information from the kernel to the host without using arrays. Referenced values can be either primitive values, records or structs. The compiler lifts ref variables replacing them with arrays of unary length;

**Records and structs** : records and structs containing primitive fields can be passed to F# kernels. Struct/record parameters are processed to generate C99 struct declaration in the OpenCL kernel source;

**Return type** : OpenCL kernels are constrained to return no value (void). The FSCL compiler removes this contraint and allows F# kernels to return array values. When a kernel returns a value, the compiler analyzes the whole kernel body and produces a valid OpenCL C99 code with an additional buffer representing the returned data. The ability to return a value allows to define kernels as overloaded operators (e.g. a multiply operator for two matrixes, which returns a matrix);

**Generic kernels** : the FSCL compiler allows to declare generic F# kernels. For example, programmers can write a parallel matrix multiplication once, using generic

[5]For example, a pragma to enable double-precision if somewhere in a kernel or function double precision variables are declared

arrays, and then run it on two integer, float, double matrices. The only constraint set by the compiler infrastructure is that generic types can be instantiated only with primitive types. When an F# kernel containing generic parameters is compiled, FSCL produces an instance of kernel for each possible combination of primitive types assigned to each generic parameter.

## 2.3 Programming interface object model

In this section we decribe the set of types and functions that the FSCL compiler exposes to the programmer. This set is actually made of three parts: the *compiler user interface*, the *compiler extension interface* and the *kernel language*. The compiler user interface is the set of types and functions that programmers rely on to create an instance of the compiler and to run the compiler pipeline on a given input and to create custom configurations of the compiler. The compiler extension interface is the part of the object model that is exposed to allow programmers to extend the compiler developing custom steps and step processors. Finally, the kernel language is a set language constructs that programmers employ to write F# kernels, such as vector types and address-space annotations for the kernel parameters.

### 2.3.1 Compiler user interface

The main information exposed by the compiler user interface is the *Compiler* type, which is characterized by a set of constructors and a single method *Compile*. The various constructors provided differ only on the configuration input. The compiler can be configured as default (no parameters) or through a configuration file or a configuration object. Section 2.5 provides and in-deph decription of the compiler configuration options. Programmer can create one or multiple instances of the FSCL compiler. Each instance exposes the *Compile* function, which takes an expression (*Expr* object) an returns an object. Returning a raw objects gives the highest freedom in extending the compiler, especially the last step (module codegen). The runtime type returned by the native compiler pipeline is a pair *(String, KernelModule)*, where the first element is the OpenCL C99 source code and the kernel module is the core structure containing all the information produced during the compilation.

### 2.3.2 Compiler extension interface

The extension interface is mainly composed by the core components libraries of the FSCL compiler project. Each library exposes a particular step type and the types of its processors. The main target of employing these libraries is to develop additional processors for pre-existing compiler steps. The appendix A provides an in-deph view of how to start developing compiler extensions.

Once developed, custom steps and processors can be employed in the pipeline properly configuring the compiler2.5.

### 2.3.3 Kernel language

## 2.4 Usage

To use the FSCL compiler infrastructure to write F# kernels and to compile them programmers must:

1. Link the appropriate libraries: *FSCL.Compiler.dll* and *FSCL.Compiler.KernelLanguage.dll*;
2. Open the appropriate namespaces: *FSCL.Compiler* and *FSCL.Compiler.KernelLanguage*;
3. Write the F# kernel and mark it with the *ReflectedDefinition* attribute;
4. Instantiate the *Compiler* type and call the *Compile* method.

The following code sample represents a template for F# kernel definition and compilation.

**Listing 2.1: Template for kernel definition and compilation using the FSCL compiler**

```
// Compiler user interface
open FSCL.Compiler
// Kernel language library
open FSCL.Compiler.KernelLanguage

// Kernel properly marked with ReflectedDefinition attribute
[<ReflectedDefinition>]
let VectorAdd(a: float32[], b: float32[], c: float32[]) =
    let gid = get_global_id(0)
    c.[gid] <- a.[gid] + b.[gid]

[<EntryPoint>]
let main argv =
    // Instantiate the compiler
    let compiler = new Compiler()
    // Compile the kernel
    compiler.Compile(<@@ VectorAdd @@>)
```

The decision of developing two separated libraries for the kernel language (FSCL.Compiler.KernelLanguage) and the compiler interface (FSCL.Compiler) has been driven by the will to allow kernel definition and kernel compilation to be completely independent. Programmers can create libraries of F# kernels, in which case only the kernel language library is needed. Similarly, to compile kernels already defined in a dll, programmers have only to refer the compiler interface library. Since developing, storing and sharing F# kernels inside libraries is one of the most important benefits of the FSCL and could become a widely popular approach, we decided to separate the user interface and the kernel language, allowing kernels developers and kernels users to be provided only of the relevant information to accomplish their particular task.

## 2.5 Compiler configuration and extension

The FSCL compiler is thought to be completely customized and easily extended thanks to a flexible plugin system. The customization and extension mechanisms are based on the compiler configuration, which describes the set of compiler components and the way they have to be orchestrated. The core of the compiler configuration is the *CompilerConfiguration* class, whose instances represent specific configurations of the pipeline components. In the rest of this document we will refer to an instance of this class as a *configuration object*.

### 2.5.1 Configuration object

A configuration object is a collection of *component sources*, where a components source can be either the path of a DLL file or the full name of a global assembly[6]. A components source can be either *implicit* or *explicit*. In the first case, the set of compiler components contained in the assembly is not explicitely specified but it is instead determined inspecting the types exposed by the assembly, looking for those annotated with one of the following custom attributes

**StepAttribute(id, dependencies = null, before = null)** : the attribute used to annotate a step definition. The global *id* must be specified, while *dependencies* and *before*, which are arrays of step ids (string), can be omitted;

**StepProcessorAttribute(id, stepId, dependencies = null, before = null)** : the attribute used to annotate a processor definition. The processor *id* and the id of the step the processor belongs to (*stepID*) are the only required parameters;

**TypeHandlerAttribute(id)** : the attribute used to expose type handlers defined inside a source. The handler *id* is required.

In case of explicit components sources, the set of components contained in the assembly and their dependencies are instead explicitely specified and the custom attribute system is ignored. The reason behind explicit sources is the flexibily of building valid compiler configurations for generic platforms without an a-priori knowledge of the extensions deployed on that platform. For example, consider two programmers respectively developing fresh extensions on two different platforms, each of which defines a step (respectively, *StepA* and *StepB*) that must be executed after the native transformation step. Each developer annotates the step with the appropriate attribute and specifies it depends on the transformation step to guarantee that the last one if executed before the custom step. One done, the extension can be compiled into a DLL and deployed. Now, consider what happens if a compiler user wants to use both the extensions. If StepA and StepB have no inter-dependencies the compiler behaves correctly regardless the relative ordering. If StepB depends on StepA, the dependencies of StepB must be changed to

[6]In case of an assembly name, the assembly itself must be available in the GAC (Global Assembly Cache) to be discovered and loaded

take into account the precence of StepA. Relying only on implicit sources would force to change the source code of StepB and, in particular, to modify the dependencies parameter of the relative StepAttribute. Thanks to explicit sources, adapting an extension to a platform containing other "conflicting" extension can instead be handled handled without the need of source code. The compiler user has only to make the components source containg StepB explicit (to bypass the custom attribute analysis) and to declare that it contains a step whose id is StepB and whose dependency is StepA. We will provide an example of implicit and explicit sources at the end of this section.

In addition to a collection of components sources, a configuration object is also characterized by its relation with the native configuration. In most of the cases, the configuration object contains steps, processors and type handlers that represent an extension of the native compiler pipeline. Native components are therefore loaded and the content of the ones defined in the configuration object are merged with them. A configuration object can also specify that it doesn't require the compiler native components. In this case, the content of the configuration object fully replaces the native pipeline.

### 2.5.2 Configuration process

The configuration process is the process that leads to the construction of a pipeline to be used to instantiate the compiler. While the target of this process is fixed (a structured and organized set of steps, processes and type handlers), the source can be one of the following:

- Configuration file;
- Configuration object;
- Implicit (default storage)

In figure 2.4 we demonstrate the control-flow executed by the configuration process to obtain a configuration object starting from a particular configuration source. As shown, if a configuration object is specified, this object is used as it is. The usage of configuration objects is useful to test/debug pipeline configurations before their final deployment. If the user specifies a configuration file (i.e. a file path) the configuration file is parsed to obtain a configuration object. Finally, if the programmer doesn't pass any file/configuration object to the compiler constructor, the configuration process looks for an xml file called *FSCL.Config.xml* in a fixed location (*ProgramData/FSCL*). If this file is found it is parsed, otherwise the process firstly builds a configuration object containing the native components and then inspects the *Components* subfolder. If this folder exists and contains one or mode components sources (DLLs), the process loads them and adds the relative components to the native configuration.

Once a configuration object has been obtained starting from one of the possible configuration sources, the configuration is *normalized*. At first, normalization transforms all the implicit sources in the configuration object to explicit ones. This action requires to load the source assembly and to analyze it to discover all the components it contains.

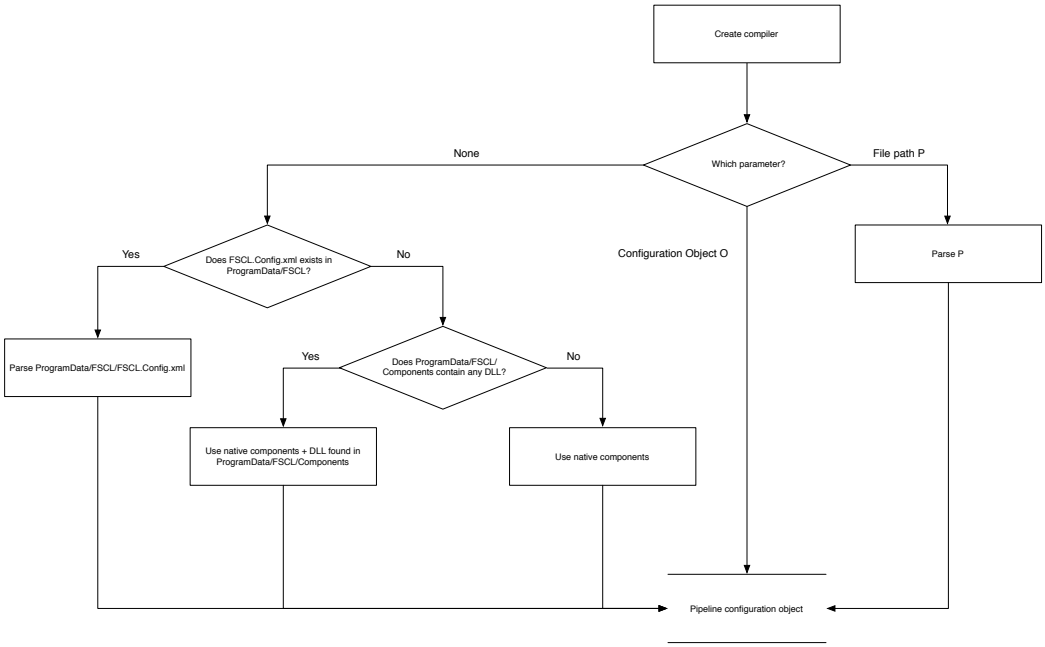


Figure 2.4: Control-flow executed to obtain the compiler configuration

After that, if the set of native components is required, these components are loaded and merged with the ones in the configuration object. Finally, the configuration is validated to guarantee that

1. Each step, processor and type-handler has a globally unique ID (there are no two steps/processors/type handlers in the configuration with the same ID);

2. Each reference to a step or processor declared by another step or processors can be resolved without escaping the configuration.

The result of this three-step process is a valid, self-contained configuration, which means that all the information about steps, processors and type-handlers have been extracted from the relative assemblies, each component can be identified using its identifier and and every reference to a step or processor is a reference to a step or processor that is part of the configuration object.

Starting from a self-contained configuration, the FSCL pipeline building system analyzes the dependencies between steps and tries to produce a partially ordered graph of steps that respects all these dependencies. In case of circular dependencies, this graph can't be obtained and the building system reports a configuration error.

The definitions 2.1 and 2.2 formally describe the normalization process.

$$
\begin{aligned}
IsValid \quad & (X : CompilerConfiguration) \iff \\
& \forall S1, S2 : S1, S2 \in Steps(X) \Rightarrow ID(S1) \neq ID(S2) \; and \\
& \forall P1, P2 \in Processors(X), ID(P1) \neq ID(P2) \; and \\
& \forall TH1, TH2 \in TypeHandlers(X), ID(TH1) \neq ID(TH2) \; and \\
& \forall S \in Steps(X), I \in After(S) \cup Before(S) \Rightarrow \exists S2 \in Steps(X) : ID(S) = I \; and \\
& \forall P \in Procs(X), \exists S \in Steps(X) : ID(S) = Step(P) \; and \\
& \forall P \in Procs(X), I \in After(P) \cup Before(P) \Rightarrow \exists P2 \in Procs(X) : ID(P2) = I
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
Normalization : \quad & X \rightarrow Y \; where \\
& X, Y \in CompilerConfiguration \; and \\
& \forall S \in ComponensSources(X) : IsExplicit(S) \; and \\
& IsValid(Y)
\end{aligned}
\tag{2.2}
$$

Starting from a self-contained configuration, the FSCL pipeline building system analyzes the dependencies between steps and tries to produce a partially ordered graph of steps that respects all these dependencies. In case of circular dependencies, this graph can't be obtained and the building system reports a configuration error. Once built, the graph is flattened to obtain a totally ordered list. No assumptions can be made on the way the building system orders the steps/processors childrens of the same node. Unordered graph building and flattening is performed also on the set of processors of each step. Once done, the compiler pipeline is fully specified and can be used to compile F# kernels. Figure 2.5 summarizes the steps involved in building a compiler pipeline strarting from a configuration object.

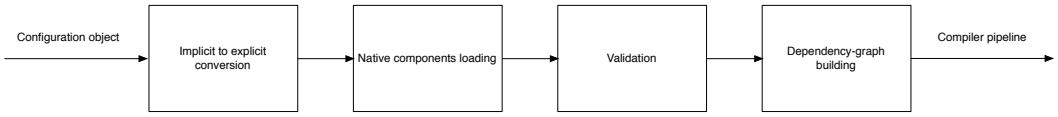


Figure 2.5: Compiler pipeline building process

### 2.5.3 Configuration samples

In this section we provide some examples of the compiler configuration capabilities. In all the examples, we consider the definition of an extension that contains:

- A step whose id is *MyStep* that must be executed after the parsing step;

- A processor *MyProcessor* belonging to the *MyStep*;
- A processor *OtherProcessor* that belongs to the native transformation step and that must be executed after the reference variables lifting.

The most straightforward option to integrate this extension in the compiler pipeline is to manually instantiate a configuration object and to pass it to the compiler constructor, as illustrated in the following snippet.

**Listing 2.2: Configuration object with implicit source**

```
// Instantiate configuration object
let configuration = CompilerConfiguration(true, // Load native components
                [
                    ComponentsSource(
                        // Implicit source
                        AssemblySource(typeof<MyStep>.Assembly)
                        // Programmer can also specify a file source,
                            FileSource("MyStep.dll")
                )])
let compiler = Compiler(configuration)
```

To same configuration can be achieved turning the assembly source explicit and manually declaring each component. To make a components source explicit programmer has only to declare at least one step, processor or type handler contained in the source. As soon as a component is explicitely declared by the programmer, the source is considered explicit and the attrubute analysis is bypassed.

**Listing 2.3: Configuration object with explicit source**

```
// Instantiate configuration object
let configuration = CompilerConfiguration(true, // Load native components
                [
                    ComponentsSource(
                        AssemblySource(typeof<MyStep>.Assembly),
                        // No type handlers
                        [],
                        // Steps
                        [ StepConfiguration("MyStep",
                            typeof<MyStep>,
                            // The id of the native parsing step
                            [ "FSCL_MODULE_PARSING_STEP"]) ],
                        // Processors
                        [ StepProcessorConfiguration("MyProcessor",
                             // The step the processor belongs to
                             "MyStep",
                             typeof<MyProcessor>);
                          StepProcessorConfiguration("OtherProcessor",
                            // The id of the transformation step
                            "FSCL_FUNCTION_TRANSFORMATION_STEP",
                            typeof<OtherProcessor>,
                            // The id of the ref variables processor
```

```
                                ["FSCL_REF_VAR_TRANSFORMATION_PROCESSOR"]);
                            ])
                    ])
let compiler = Compiler(configuration)
```

Instead of creating a new instance of the configuration object, the programmer can store the configuration in an XML file and pass it to the compiler constructor. The structure of a generic xml configuration file is deeply resembles the structure of a configuration object, as illustrated in the following samples.

**Listing 2.4: Configuration file with implicit source**

```
<?xml version="1.0" encoding="utf-8" ?>
<CompilerConfiguration LoadDefaultSteps="true">
    <ComponentsSources>
        <!-- Implicit source -->
        <ComponentsSource FileSource="MyAssembly.dll"/>
    </ComponentsSources>
</CompilerConfiguration>
```

**Listing 2.5: Configuration file with explicit source**

```
<?xml version="1.0" encoding="utf-8"?>
<CompilerConfiguration LoadDefaultSteps="true">
    <ComponentsSources>
        <!-- Explicit source -->
        <ComponentsSource FileSource="MyAssembly.dll">
            <StepsConfiguration>
                <StepConfiguration ID="MyStep" Type="MyStep">
                    <Dependencies>
                        <Item ID="FSCL_MODULE_PARSING_STEP" />
                    </Dependencies>
                </StepConfiguration>
            </StepsConfiguration>
            <StepProcessorsConfiguration>
                <StepProcessorConfiguration ID="MyProcessor"
                    Step="MyStep"
                    Type="MyStep">
                </StepProcessorConfiguration>
                <StepProcessorConfiguration ID="OtherProcessor"
                        Step="FSCL_FUNCTION_TRANSFORMATION_PROCESSOR"
                        Type="OtherProcessor">
                    <Dependencies>
                        <Item ID="FSCL_REF_VAR_TRANSFORMATION_PROCESSOR"/>
                    </Dependencies>
                </StepProcessorConfiguration>
            </StepProcessorsConfiguration>
        </ComponentsSource>
    </ComponentsSources>
</CompilerConfiguration>
```

To configure the compiler using an XML file, it is sufficient to pass the path of the file to the compiler constructor.

**Listing 2.6: Instantiate the compiler using a configuration file**

```
let compiler = Compiler("conf.xml")
```

Most of the compiler users don't want to cope with configuration objects or files. For this reason, FSCL provides a third option to configure the compiler, which is totally transparent to the users and therefore the preferred choice for long-term deployment of stable configurations. This option is based on two folders: *ProgramData/FSCL* and *ProgramData/FSCL/Components*. To deploy the extension proposed in this section, users can either copy the XML configuration file in the first of these two folders and rename it to *FSCL.Config.xml* or copy the DLL in the second folder. If a configuration file already exists in *ProgramData/FSCL* the new components source must be appended to the existg ones, as illustrated in the following sample.

**Listing 2.7: Default configuration file FSCL.Config.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<CompilerConfiguration ...>
    <ComponentsSources>
        <ComponentsSource FileSource="...">
        </ComponentsSource>
        <ComponentsSource FileSource="...">
        </ComponentsSource>
        ...
        <ComponentsSource FileSource="PathToAssembly/MyAssembly.dll"/>
    </ComponentsSources>
</CompilerConfiguration>
```

## 2.6 F# kernel samples

**Listing 2.8: Simple vector addition**

```
[<ReflectedDefinition>]
let VectorAdd(a: float32[], b: float32[], c: float32[]) =
    let gid = get_global_id(0)
    c.[gid] <- a.[gid] + b.[gid]

[<EntryPoint>]
let main argv =
    let compiler = new Compiler()
    let result = compiler.Compile(<@@ VectorAdd @@>)
    ...
```

**Listing 2.9: Matrix convolution**

```
[<ReflectedDefinition>]
let filterWidth = 3

[<ReflectedDefinition>]
let Convolution(input:float32[,], [<Constant>]filter:float32[,], output:
    float32[,], [<Local>]block:float32[,]) =
    let output_width = get_global_size(0)
    let input_width = output_width + filterWidth - 1
    let xOut = get_global_id(0)
    let yOut = get_global_id(1)

    let local_x = get_local_id(0)
    let local_y = get_local_id(1)
    let group_width = get_local_size(0)
    let group_height = get_local_size(1)
    let block_width = group_width + filterWidth - 1
    let block_height = group_height + filterWidth - 1

    //Set required rows into the LDS
    let mutable global_start_x = xOut
    let mutable global_start_y = yOut
    for local_start_x in local_x .. group_width .. block_width do
        for local_start_y in local_y .. group_height .. block_height do
            block.[local_start_y, local_start_x] <- input.[global_start_y,
                 global_start_x]
            global_start_x <- global_start_x + group_width
            global_start_y <- global_start_y + group_height

    let mutable sum = 0.0f
    for r = 0 to filterWidth - 1 do
        for c = 0 to filterWidth - 1 do
            sum <- (filter.[r,c]) * (block.[local_y + r, local_x + c])
    output.[yOut,xOut] <- sum

[<EntryPoint>]
let main argv =
    let compiler = new Compiler()
    let result = compiler.Compile(<@@ Convolution @@>)
    ...
```

# Chapter 3

# FSCL Runtime

The FSCL compiler provides a syntax, a set of data-types and a compilation pipeline to express OpenCL kernels inside F# and to compile them into OpenCL C99 kernel sources. The choice of targeting OpenCL source code is driven by the aim of keeping the compiler independent from the instruction set of particular device models and manufacturers, relying on specific source-to-device binary compilers released by all the major CPU-GPU manufacturers supporting OpenCL to produce executable device code.

Nowadays, there are many OpenCL projects for .NET that allow to build and execute OpenCL C99 kernels without escaping the virtual machine environment, such as Cloo and OpenCL.NET. Those projects are nothing more than a one-to-one mapping of the C/C++ routines exposed by the OpenCL library, which means they expose a set of functions and data-types that strongly resemble the ones they wrap. In particular, as like as the OpenCL library, they require stringified OpenCL C99 kernels as input of the kernel build function. FSCL users can produce this input starting from F# kernels, exploiting all the features of the FSCL compiler. Once done, they can rely on these .NET OpenCL wrappers to fill the gap from OpenCL kernel source code to device executable and to run the computation on the device itself.

Nevertheless, there are various disadvantages of using the combination of FSCL and OpenCL .NET to accomplish the whole workflow from the definition of a kernel to its execution on a target device. At first OpenCL .NET wrappers don't offer any support in executing the computation on the target device, such as device discovery, automatic buffer creation and disposal and host-device synchronization. Given the rich set of meta informations together with a power of virtual machine reflection infrastructure, most of these actions could be automatize. Another disadvantage is the break of the abstraction provided by kernels expressed in F#. Programmers use two independent tools for the first (F# to OpenCL) and the second (OpenCL to executable and execution) parts of the general OpenCL development and execution workflow, using F# source code as the interface between them. Being exposed to programmers, source code might be accidentally or intentionally modified, leading to kernels that can't execution or whose execution behaviour is unpredictable.

The FSCL *Runtime* is a framework built on top of the FSCL compiler whose target

is to higher the level of abstraction over OpenCL host-code, providing a set of services to make device-code generation transparent to the programmer and to simplify the management of kernel executions. Thanks to this framework, programmers do not have to rely on an OpenCL .NET wrapper to produce executable code and to coordinate its execution on the target device. Moreover, the FSCL runtime allows to automatize about all the host-code that is required to setup, schedule e coordinate the kernel execution, allowing programmers to focus exclusively on the algorithm to be executed.

## 3.1 Services and features

The basic feature of KernelRunner is making the generation of device executable starting from OpenCL C99 source code transparent to the programmer. Programmers no longer use FSCL compiler directly but they exploit the Runtime user interface to generate device executable.

In addition, the framework allows to simplify the host-code part of OpenCL computations and to save the programmer from all the actions required to setup and handle the execution of the kernel on the device, such as buffer allocation, initialization, and transfer, device selection and host-device synchronizations. Regarding buffers, FSCL runtime is capable to recognize the size and the of each kernel parameter and the way the F# kernel accesses it (read-only, write-only, read-write). Thank to this ability, buffers can be allocated, inizialized and data can be transferred trasparently to the programmer.

Synchronization with the device is also performed behind the scenes. When the kernel execution ends, all the buffers potentially changed by the kernel[1] are read and the corresponding .NET object are appropriately set. In this way, programmers obtain the data produced by the kernel execution without escaping the managed environment, without coping with explicit buffer reads and with no effort in handling managed-unmanaged data transformation.

In addition, KernelRunner integrates a *metric-based* device selection service for platforms exposing multiple OpenCL devices whose target is to analyze F# kernels and to transparently select the best[2] device to execute it.

Generating target device code, analyzing the source of F# kernels to handle buffer allocation and data-transfer and to determine the best OpenCL-enabled device on the platform for execution might introduce an high overhead. For this reason, the KernelRunner adopts multiple optimizations on both the metric system and on the code generation and access analysis of kernel parameters. The first set of optimizations is fully described in section 3.5. For what regards device code generation and kernel parameters analysis, the framework applies a strategy called *kernel caching*. The first time the programmer executes a particular kernel, the kernel body is analyzed to determine:

- The OpenCL device where to execute the computation, if the programmer doesn't force the execution on a particular device;

[1] On the basis of the access mode evaluation performed analyzing the F# kernel

[2] For *best* device we mean the device that the KernelRunner estimates to be the best choice (e.g. the one with the lowest completion-time) on the basis of the chosen metric

- For each parameter that requires a buffer (an array or a variable that the kernel can modify with side-effects), the parameter type, the size (if constant) and the kernel access mode

These information, together with a pointer to the dump of the executable, are stored inside an internal data-structure indexed with the kernel signature. Successive invocations to the same kernel requires only to access this structure, bypassing both the device-code generation and the entire analysis process, which strongly reduces the global overhead.

## 3.2 Framework structure

The FSCL runtime framework is made of a set of interrelated components built on top of the FSCL compiler, as shown in figure 3.1. The runtime interacts with the

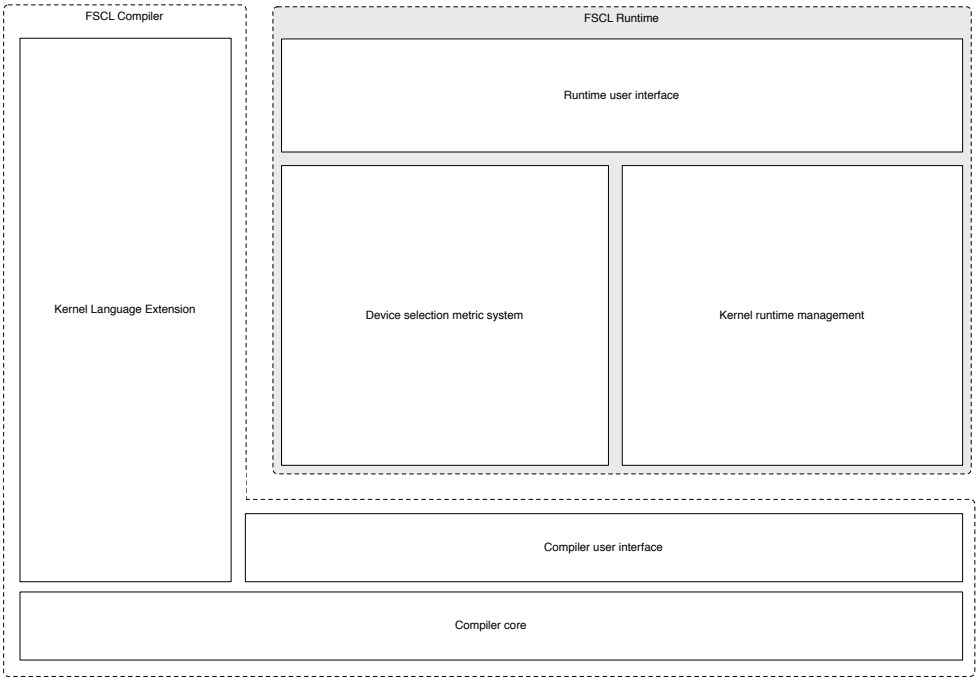


Figure 3.1: Structure of the FSCL runtime framework

compiler exclusively through the compiler user interface, without any assumption on the implementation. In other terms the compiler could be entirely replaced without affecting the runtime, as long as the interface continues to expose a *Compile* function returning the approriate data-structure that contains the result of the compilation.

The main two components of the runtime are the *device selection metric system* and the *kernel runtime management*. The first one is plugin-based subsystem responsible to select the "most suitable" Opencl-enabled device for each computation to execute (section 3.5). The second component manages the compilation and the execution of kernels

during the lifetime of the host application with the target of reducing the framework and the kernel execution overheads (section 3.3).

The runtime user interface represents the object model exposed by the runtime, which is a set of functions and datatypes to customize and trigger the execution of F# kernels. The customization consists in a set of programming constructs to force the scheduling on a particular device, to switch to multithread execution (section 3.4) or to change the dafault management strategy of kernel buffers.

## 3.3 Kernel runtime management

The kernel runtime manager is a component responsible of managing F# kernels compilation and execution. To accomplish this target, the runtime manager uses two main data-structures: the *kernel storage* and the *device storage*. The first one contains many information regarding the F# kernels that the FSCL runtime is asked to execute. The device storage instead contains information describing the OpenCL devices used to run the kernels, together with all the data allocated per-device and potentially reused, such as command queues and buffers.

The basic feature provided is the F#-to-OpenCL and the OpenCL-to-Device binary compilation. The first one is performed using the FSCL compiler, while the second is obtained properly invoking the functions exposed by the OpenCL library provided by the specific device manufacturer. Once compiled for a specific device, the code produced by the FSCL compiler (i.e. OpenCL source code) is stored in the kernel storage for future usage and the executable is dumped on disk. If the same kernel is successively asked to run on a different device, the stored OpenCL source code is used to save the FSCL compiler overhead. In addition, if the kernel is required to run on the same device previously used, the runtime manager retrieves the dumped executable to cut out the entire compilation process.

The first time an F# kernel is required to run on a particular device, the kernel runtime manager allocates all the OpenCL data-structures needed to communicate with that device and to schedule computations on it, such as device contexts, command queues. As like as the OpenCL source code produced by the FSCL compiler, these data are not released after the end of the computation but only at the end of the host application. Therefore executing multiple kernels on the same device is affected by a single context/queue allocation overhead.

The kernel runtime manager is also responsible of allocation and initialization of OpenCL buffers. Thanks to the power of .NET reflection, this task can be performed in a way that is completely transparent to the programmer, analyzing the F# kernel parameters and inspecting the kernel body to determine the way array parameters[3] are used. If an array parameter is only read/written in the kernel body, a read-only/write-only buffer is allocated. Otherwise a read-write buffer is created. This analysis assumes that the kernel body doesn't contain any array parameter alias. Aliasing is verified

[3]With array parameters we mean non-scalar parameters that require buffers, i.e. arrays, lists and reference cells

checking if there are some array accesses $A[x]$ where $A$ is not a reference to a kernel parameter. If an access of this type is found and exception is reported. More complex aliasing cases are not covered. Array parameters marked with *Constant* or *Local* are threated differently to match the OpenCL specification. In particular, no buffers are allocated for array parameters marked as local, while constant array parameters are verified to be properly accessed (read-only) inside the kernel body.

Unless differently specified, buffers are created on the device (the) and reading/writing from/to buffers is performed calling *clEnqueueReadBuffer*/*clEnqueueWriteBuffer*. The FSCL runtime exposes a set of functions to customize the buffer allocation and read-/write strategy (e.g. mapping, buffer flags such as $CL\_MEM\_ALLOC\_HOST\_POINTER$).

By default, buffers are not release at the end of the kernel execution. This allows to reuse them for other kernels scheduled on the same device[4]. If an already allocated buffer isn't big enough to store a parameter of the new kernel or is incompatible (e.g. the buffer was created read-only but the parameter is written in the body of the new kernel), the allocation of a new buffer is performed together with the release of the old one.

Synchronization and reading of kernel results are also performed automatically. By default, every buffer correponding to an array parameter modified in the kernel body is read at the end of the computation. In some cases programmers might want to use arrays to exchange data between successive kernels. For example, in a map-reduce algorithms, programmers might use an array to contain the result of the map, which is the first kernel executed, and pass it to the reduce. If an array is exclusively used inside F# kernels[5] the overhead of transferring its content from the device to the CPU (host) should be avoided. Therefore, the FSCL runtime exposes some constructs to annotate a kernel parameter as "not-used", which prevents the kernel runtime manager from transferring back the content of the corresponding buffer at the end of the computation.

## 3.4 Multithreading support and fallback

The FSCL runtime is capable of switching from OpenCL-based execution and multi-thread/sequential execution on the CPU. There are two cases where FSCL runtime selects multithread or sequential execution:

- The programmer explicitely ask the runtime to execute an F# kernel in multithread or sequential mode;
- The system doesn't contain any OpenCL-enabled device.

Whereas the system doesn't expose any OpenCL-enable device, the FSCL runtime fallbacks to multithread execution in a way that is completely transparent to the programmer. No changes to the kernel or to the way it is invoked are required. CPU threads

[4]Strictly speaking, the OpenCL specification declares that buffers are allocated per-context and not per-device, but the runtime manager creates a different context for each OpenCL deivce in the system

[5]From an OpenCL C perspective, the array is not read in what is called the "host code".

are created and spawn using the same ids organization adopted by OpenCL. Given a three-dimensional threads workspace, ids are 3D values [x, y, z] assigned using an higher-dimension major order. For example, given a 3D space with size [2, 2, 3], the ids are generated in the following order:

> [0, 0 ,0], [0, 0, 1], [0, 0, 2], [0, 1, 0], [0, 1, 1], [0, 1, 2], [1, 0, 0], [1, 0, 1], [1, 0, 2], [1, 1, 0], [1, 1, 1], [1, 1, 2]

One-dimensional and two-dimensional threads spaces are threated extending them to three-dimensional spaces whose higher sizes are set to 1 (e.g. A 1D space of size 10 is threated as a 3D space of sizes [10, 1, 1]).

Given the matching of OpenCL ids management, programmers can use multithreading execution without the need to change kernel code nor to change the threads space size. In addition, this makes much more meaningful using multithread or sequential execution to test or debug OpenCL kernels before the execution on an OpenCL-enabled device.

The FSCL runtime and compiler are thought to enable development and execution of OpenCL kernels through F# functions, instance and static methods. Moreover, these frameworks try to keep kernel syntax as close as possible to the standard OpenCL C99 language, while introducing additional features, such as generic types and collections, that programmers can employ to high the coding abstraction level. To preserve the F# kernel sintax close to OpenCL C99 threads-space related functions, such as *get_global_id*, *get_local_size* and *barrier*[6] are implemented as module functions. Once the module has been opened, no objects or class names are involved to call them. In the F# to OpenCL compilation, those functions are simply used to generate the appropriate function call. That is, only the function signature is used to produce OpenCL c99 source, while the implementation (the semantic) is provided by the OpenCL compiler of the specific device manufacturer.

While OpenCL execution requires F# kernels to be compiled into OpenCL C99 sources and consequently into target device executables, when multithread execution is selected the F# kernel should be directly executed. Therefore, the FSCL compiler/runtime must provide a meaningful implementation of thread-space related functions. Unfortunately, since these functions are defined as global module functions, it is difficult to provide an implementation that returns the correct thread id to each thread that invokes them and regardless the thread-space size.

We took into account various options to cope with this problem. The most straightforward solution consists in replacing module functions with instance methods. While developing kernels, instead of invoking *get_global_id* programmers invoke $x$.*get_global_id*, where $x$ is an instance of a utility class exposing thread-space related functions and containing appropriate values for global/local ids and objects to be used for inter-thread synchronization. The drawback of this solution is the need to take into account this

[6] barrier is an OpenCL C function used to synchronize thread in a group or all the threads globally spawn

utility class/type in writing kernels and to explicitely add a parameter of this type to the kernel signature.

Another possibility consists in transforming the AST of kernel body. Since FSCL can both retrieve and build ASTs through F# quotations feature, it is possible to process the kernel body to substitute every occurrence of a specific module function call (e.g. *get_global_id*, *barrier*) with the correspondent invocation of an instance method. The main problem of this approach is that there are no builtin ways to compile/evaluate ASTs. Moreover, even if we empoyed libraries to evaluate ASTs, such as the one provided by Linq.Expression, we would introduce an high overhead due to the additional evaluation layer[7].

To prevent the programmer from introducing additional kernel parameters to retrieve global and local thread ids while maintaining high performances, the FSCL runtime emploies a solution based on dynamic assembly/methods and intermediate language injection. In particular, whenever an F# kernel has to be executed using multithreading, a dynamic method is generated behind the scenes. The signature of the new method matches the one of the F# kernel, except for an additional parameter that represents a container for the thread-space information, like global and local ids and objects to be used to implement group-wise and global synchronization. The body of the dynamic method is generated directly using MSIL instructions. The set of instructions is taken from the body of the original method and modified, injecting instructions to load the additional parameter and replacing each call to thread-space module functions with a call to the proper instance method exposed by the additional parameter itself. The following code snippet exemplifies the dynamic method generation when multithread execution is required.

**Listing 3.1: Dynamic method generation sample for multithread execution**

```
// Thread-space information container
type ThreadSpaceInformationContainer(global_id, local_id, ...,
    local_sync_object)
   // Information
   member val GlobalId = global_id with get
   ...
   member val LocalSyncObject = local_sinc_object with get

   // Functions matching the thread-space module ones
   member this.GetGlobalId(i) = ...
   member this.Barrier(mode) = ...
   ...

// Original F# kernel
[<ReflectedDefinition>]
let OriginalKernel(input:float32[,], output:float32[,]) =
    let output_width = get_global_size(0)
    ...
    barrier(CLK_LOCAL_MEM_FENCE)
    ...
```

[7]The code executed on the VM analyzes the AST and evaluates its nodes using reflection

```
// Generated dynamic method
let DynamicKernel(input:float32[,], output:float32[,], tsic:
    ThreadSpaceInformationContainer) =
    let output_width = tsic.GetGlobalId(0)
    ...
    tsic.Barrier(CLK_LOCAL_MEM_FENCE)
    ...
```

As shown, starting from an F# kernel a method containing an additional parameter of type *ThreadSpaceInformationContainer* is generated. In the body of this method every call to a thread-space module function is replaced with the corresponding method of the *ThreadSpaceInformationContainer* instance. The same dynamic function can be used for sequential execution, which simply consists in invoking the function iteratively, each time using a properly set a *ThreadSpaceInformationContainer* instance.

FSCL generates the dynamic method only the first time a particular kernel is required to run in multithreading. Once genrated it is stored in an internal data structure. Therefore sucessive multithread executions of the same kernels requires only a method invocation. Moreover, since the body of the dynamic method is built in memory using IL instructions, its execution has the same performances of statically defined methods.

The only drawback of dynamic method generation is the difficulties in debugging code. If debugging is required in multithread/sequential execution mode, programmers must explicitely pass the *ThreadSpaceInformationContainer* as a parameter of the F# kernel and use its instance methods in place of thread-space module functions. The FSCL runtime automatically recognizes a kernel taking this additional parameter and doesn't produce any dynamic method for multithread execution, simplifying debugging. In addition, the runtime is also able to produce valid OpenCL kernels from this kind of kernels, performing the opposite transformation [8].

For more information about multithreading support, fallback mechanisms and code samples please refer to section 3.6.

## 3.5 Metric-based scheduling system

TODO

## 3.6 Usage

In most of the cases, the entire runtime infrastructure is completely hidden to FSCL users. To compile and execute an F# kernel, programmers are only asked to put the F# kernel invocation inside a quotation and invoke the (extension) method *Run* on it. The following example shows how to execute a simple vector addition.

[8] the last parameter is removed and every invocation to one of its methods is replaced with the appropriate module function

**Listing 3.2: Running a simple vector addition**

```
[<ReflectedDefinition>]
let VectorAdd(a: float32[], b: float32[], c: float32[]) =
    let gid = get_global_id(0)
    c.[gid] <- a.[gid] + b.[gid]

[<EntryPoint>]
let main argv =
    // Create input and output arrays or 100 elements
    let a = Array.create(..., 100)
    let b = Array.create(..., 100)
    let c = Array.create(..., 100)

    // Execute the computation
    <@ VectorAdd(a, b, c) @>.Run(100, 10)

    // Now c contains the sum of a and b
    ...
```

The Run method takes the global and local sizes as parameters. In case of one-dimensional thread-space, programmers can use two integer values. Otherwise, the must pass two integer arrays of the appropariate rank, as shown in the following sample, where a two-dimensional thread-space is created to perform matrix multiplication where each thread multiplies a specific row of the first matrix by a specific column of the second.

**Listing 3.3: Matrix multiplication**

```
[<ReflectedDefinition>]
let MatMult(a: float32[,], b: float32[,], c: float32[,]) =
    let row = get_global_id(0)
    let col = get_global_id(1)
    ...

[<EntryPoint>]
let main argv =
    // Create input and output 2D arrays of 100x100 elements
    let a = Array2D.create(..., 100, 100)
    let b = Array2D.create(..., 100, 100)
    let c = Array2D.create(..., 100, 100)

    // Execute the computation
    <@ MatMult(a, b, c) @>.Run([| 100; 100 |], [| 10; 10 |])

    ...
```

When using the Run method, multithread fallbacking is performed automatically if the FSCL runtime doesn't find any OpenCL-enabled device in the system. Alternatively, programmers can force the multithread and sequential execution respectively using *RunMultithread* and *RunSequential*, as shown in the following code sample.

**Listing 3.4: Multithread and sequential execution**

```
[<ReflectedDefinition>]
let VectorAdd(a: float32[], b: float32[], c: float32[]) =
   ...

[<EntryPoint>]
let main argv =
    ...
    // OpenCL (if possible) with multithread fallback
    <@ VectorAdd(a, b, c) @>.Run(100, 10)

    // OpenCL (if possible) otherwise an exception is thrown
    <@ VectorAdd(a, b, c) @>.RunOpenCL(100, 10)

    // Multithreading
    <@ VectorAdd(a, b, c) @>.RunMultithread(100, 10)

    // Sequential
    <@ VectorAdd(a, b, c) @>.RunSequential(100, 10)
    ...
```

As shown, multithreading fallback can be explicitely disable (RunOpenCL). This is particularly useful the computation is required to run exclusively in OpenCL and programmers know one or more OpenCL-enabled devices are installed in the system. In this case, fallbacking is the symptom of a device/operative system/driver problem and an exception should be immediately reported by FSCL instead of silently executing the kernel using CPU threads.

Since multithread execution involves the generation of a dynamic method starting from the F# kernel, programmers cannot exploit integrated, visual debugging when executing kernels in multithread/sequential mode. To make debugging easier, programmers must add a specific parameter that represents a wrapper for all the information regarding the thread space and use the functions exposed by this wrapper. A comparison between the standard kernel definition and the one that simplifies debugging is shown in the following code sample.

**Listing 3.5: Explicit thread space wrapper for visual debugging**

```
// Standard kernel defintiion
[<ReflectedDefinition>]
let StandardKernel(a: float32[], b: float32[], c: float32[]) =
   let id = get_global_id(0)
   let local_id = get_local_id(0)
   ...
   barrier(CLK_LOCAL_MEM_FENCE)

// Debugging-enabled kernel definition
[<ReflectedDefinition>]
let DebuggableKernel(a: float32[], b: float32[], c: float32[]) (space:
    ThreadSpaceInformationContainer) =
   let id = space.get_global_id(0)
   let local_id = space.get_local_id(0)
   ...
```

```
    space.barrier(CLK_LOCAL_MEM_FENCE)

// Entry-point
[<EntryPoint>]
let main argv =
    ...
    // Execute standard kernel
    <@ StandardKernel(a, b, c) @>.RunMultithread(100, 10)

    // Execute debuggable kernel (no istance of
        ThreadSpaceInformationContainer is required)
    <@ DebuggableKernel(a, b, c) @>.RunMultithread(100, 10)
    ...
```

As shown, to enable visual debugging programmers must add a curried parameter of type *ThreadSpaceInformationContainer* and replace every call to a function concerning the thread space (e.g. *get_global_id*(), *barrier*()) with a call to the matching function exposed by the additional parameter. Note that it is not required to provide a value for the additional parameter to execute this kind of kernels. The approriate value is determined by the FSCL runtime when the CPU threads are created. It is also important to note that a kernel definition with an explicit ThreadSpaceInformationContainer parameter is meaningful only for multithread and sequential execution. When OpenCL execution is selected, F# kernels cannot be exploit visual debugging since the actual computation escapes the virtual machine environment.

When a computation is executed on a system that contains many OpenCL-enables devices, device section is generally performed transparently to the programmer, thanks to the FSCL runtime scheduling system. Programmers can also force a computation to be executed on a specific device through the *Device* attribute. The device is globally identified by a pair of indexes The first is the index of the OpenCL platform in the ordered list of available platforms, while the second is the index of the device in the ordered list of devices belonging to that platform. To retrieve the list of available platforms and devices programmers can invoke the static method *AvailableDevices* provided by the *FSCLRuntime* class. The following sample shows how to use the device attribute to force a computation to run on the second device of the first available platform.

**Listing 3.6: Static computation scheduling**

```
[<Device(0, 1)>]
[<ReflectedDefinition>]
let VectorAdd(a: float32[], b: float32[], c: float32[]) =
    ...
```

# Appendix A

# Guidelines for FSCL Compiler extension development

The FSCL compiler can be extended in two ways: creating a new step or creating additional processors for pre-existing steps.

## A.1 Developing custom steps

A custom step must inherit from $CompilerStep < T', U' >$, where $T$ is the input type of the step and $U$ is the output type.

**Listing A.1: CompilerStep class**

```
[<AbstractClass>]
type CompilerStep<'T,'U>(tm:TypeManager, procs:ICompilerStepProcessor[]) =
    ...
    member val Processors = procs with get
    abstract member Run: 'T -> 'U
```

To define a new step, programmers must provide an implementation of the *Run* method, respecting the input/output type contraints. Since the entire infrastructure design is based on steps executing processors, the base type requires to specify also the set of processors of the custom step to be instantiated. Nevertheless, programmers are free to emply and empty array, which means the step doesn't contain an processor and all the behaviour of the step is performed by the step itself inside in *Run* function. Even if a "processorless" step might looks like the best choice for very simple steps (e.g. the module codegen, whose target is to concatenate the pieces of code of each function/kernel in the module), for a design, modularity and extensibility purpose, this approach is discouraged.

To be correctly deployed in the compiler pipeline, a step must identified by a globally unique id and must declare its dependencies. In case of explicit compiler configurations, these informations can be provided dinamically (section 2.5). Anyway, it is a better

approach to statically provide all the known information regarding the id and the dependencies, since this allows to integrate the custom step into the automatic compiler configuration process. Assuming that the id of the custom step is $NEW\_STEP\_ID$ and that it must be executed after the steps $STEP\_A$ and $STEP\_B$, the custom step declaration will look as follows:

**Listing A.2: Custom step definition**

```
[Step(ID="NEW_STEP_ID", After=[| "STEP_A"; "STEP_B" |])]
type MyCustomStep(tm:TypeManager, procs:ICompilerStepProcessor[]) =
    inherit CompilerStep<MyInputType, MyOuputType>(tm, procs)

    override this.Run(i:MyInputType) =
      ...
```

To integrate the custom step please refer to section section 2.5.

## A.2 Developing custom step processors

To develop a step processor for a pre-existing step, the starting point is the definition of the step itself, since different steps accept processors of different types and execute them using different approaches.

Once determined the step execution strategy and processors type, the programmer can define a new processor declaring a class extending $CompilerStepProcessor < I, O$, which is the base class of every step processor.

**Listing A.3: CompilerStepProcessor class**

```
type CompilerStepProcessor<'T,'U> =
   ...

   abstract member Process: 'T * ICompilerStep -> 'U
```

A custom processor must implement the method *Process*, which takes an instance of the processor input type, a reference to the owner step and produces an instance of the processor output type.

As a concrete example, consider the native function codegen step. This step accepts processors of type *FunctionBodyCodegenProcessor*, which is an alias of $CompilerStepProcessor < Expr, Stringoption >$. Therefore, a custom processors for the function codegen step must implement a *Process* method that takes an *Expr* and an *IcompilerStep* and produces a *String option*.

Like steps, processors must be globally identifiable and must declares inter-processors dependencies. These information can be associated to the processor using custom attributes, in the same way programmers associate ids and dependencies to compiler steps. In case of processor, also the identifier of the owenr step must be declared.

**Listing A.4: Custom processor definition**

```
[<StepProcessor("MY_CUSTOM_PROCESSOR_ID", "OWNER_STEP_ID", After=[|...|])
    >]
type MyCustomProcessor() =
   inherit CompilerStepProcessor<MyProcessorIputType,
       MyProcessorOutputType>()
   override this.Process(i:MyProcessorInputType, s:ICompilerStep) =
      ...
```

Before starting writing custom processors and steps, we suggest to take a look to the projects in the *Core* folder of the compiler sources, which contain the definition of the native steps and processors of the FSCL compiler.

## A.3 Extension sample

We provide a concrete example of extending the FSCL compiler showing how to integrate a new type into the compiler system. In particular, we extend the compiler to enable programmers to use pairs (tuples of two elements) in writing F# kernels. For what regards the representation of the new type in the target language, pairs can be easily shaped with C structs. In addition, the FSCL compiler native components handle both F# records and .NET struct types, which are both mapped to C structs. Therefore, a prossible approach to integrate pairs into the compiler system is to map them to .NET structs in the early compiler phases and to exploit the native .NET structs support to do the rest (generate custom struct declaration in the target code, generate struct access costructs, etc.).

The definitions of struct types that must appear in the target code are contained in the *Global types* field of the kernel module. Therefore, the step involved in generating struct type definitions must be module-wise (i.e. must have access to the whole module). This is the reason why the native processor responsible of detecting structs usage and of generating the corresponding definitions is owned by the *ModuleProceprocessing* step, which is the module-wise step executed right after parsing. This processor analyzes the signature and the body of both kernels and utility functions. Wherever a struct or record types is encountered, the corresponding definition is enqueued in the global types field of the kernel module to guarantee to be inserted in the target code.

For what regards the extension for pairs, this means that to exploit the FSCL structs support we must transform pairs into structs before structs discovery processor is executed. We can therefore define a new processor for the *ModulePreprocessing* step and declared that is must be executed before the structs discovery. A little documentation allows to determine that the id of the step is $FSCL\_MODULE\_PREPROCESSING\_STEP$ and the id of the structs discovery processors is $FSCL\_STRUCT\_DISCOVERY\_PROCESSOR$. To guarantee that our custom processor is executed before the struct discovery, we could either insert the id of our processor in the list of depedencies of the struct discovery (to force discovery to run *after* our processor) or insert "$FSCL\_STRUCT\_DISCOVERY\_PROCESSOR$" is the *Before* list of out processor (to force the discovery to run *before* our processor). The second approach is the best choice, since it saves us from modifying a source of

the FSCL compiler native components and from recompiling the entire FSCL compiler project.

Before starting developing our custom processor, we check that the module preprocessing step accepts processors of type *CompilerStepProcessor¡KernelModule, unit¿*. We have now all the information to write the a draft of our custom processor, that we will call *PairDiscoveryProcessor*.

**Listing A.5: Pair discovery processor definition draft**

```
[<StepProcessor("PAIR_DISCOVERY_PROCESSOR", "
    FSCL_MODULE_PREPROCESSING_STEP", Before=[| "
    FSCL_STRUCT_DISCOVERY_PROCESSOR" |])>]
type PairDiscoveryProcessor() =
   inherit CompilerStepProcessor<KernelModule, unit>()
   override this.Process(km: KernelModule, s: ICompilerStep) =
      ...
```

Its time to provide an implementation to the *Process* method. We must iterate troughout the kernels and functions in the kernel module and analyze their signatures and bodies looking for tuple types.

**Listing A.6: Process method implementation**

```
override this.Process(km: KernelModule, s: ICompilerStep) =
   let step = s :?> ModulePreprocessingStep
   let pairDict = new Dictionary<Type, unit>()

   for functionInfo in km.Kernels do
      CollectPairs(functionInfo, pairDict)
   for functionInfo in km.Functions do
      CollectPairs(functionInfo, pairDict)

   // Store the struct types inside kernel module as a flat list
   let structList = PairToStruct(pairDict)
   km.GlobalTypes <- km.GlobalTypes @ structList
```

Out functions starts collecting the pairs found inside kernels and functions in a dictionary. The dictionary allows to avoid collecting the same pair type (e.g. int * float) multiple times. After having collected all the pairs in the module, we transform them into appropriate custom struct types to be enqueued in the module global types. Let's processed implementing the *CollectPairs* method:

**Listing A.7: CollectPairs method implementation**

```
let rec CollectPairsInBody(e: Expr, pairs: Dictionary<Type, unit>) =
   let t = e.Type
   if FSharpType.IsTuple(t) && (FSharpType.GetTupleElements(t).Length = 2)
        then
      if not (pairs.ContainsKey(t)) then
         pairs.Add(t, ())
   // Recursive analysis
```

```
    match e with
    | ExprShape.ShapeVar(v) ->
       ()
    | ExprShape.ShapeLambda(v, body) ->
       CollectPairsInBody(body, pairs)
    | ExprShape.ShapeCombination(o, l) ->
       let t = o.GetType()
       if FSharpType.IsTuple(t) && (FSharpType.GetTupleElements(t).Length =
            2) then
          if not (pairs.ContainsKey(t)) then
              pairs.Add(t, ())
       List.iter(fun (e:Expr) -> CollectPairs(e, pairs)) l

let CollectPairsInSignature(m: MethodInfo:, pairs: Dictionary<Type, unit>)
     =
    for pInfo in m.GetParameters() do
       let t = pInfo.ParameterType
       if FSharpType.IsTuple(t) && (FSharpType.GetTupleElements(t).Length =
            2) then
          if not (pairs.ContainsKey(t)) then
              pairs.Add(t, ())

let CollectPairs(f: functionInfo, structs: Dictionary<Type, unit>) =
    CollectPairsInSignature(f.Signature)
    CollectPairsInBody(f.Body)
```

The discovery is performed separately on the function/kernel signature and on the relative body. Discovery inside the body is implemented recursively analyzing the AST. Note that we collect only tuples made of two elements, while tuples with three or more elements are ignored. Since there is no type handler for tuples, this means that if the programmer will employ tuples types with 3 or more elements, the compiler system will report an error during codegen phases, due to the lack of support for such kind of types.

To complete our processor we only need to implement the *PairToStruct* method, which generates proper structs types starting from the collected tuple types.

**Listing A.8: PairToStruct method implementation**

```
let PairToStruct(pairs: Dictionary<Type, unit>) =
    let asmName = AssemblyName("PairToStructConversion")
    let asm = AppDomain.CurrentDomain.DefineDynamicAssembly(asmName,
        AssemblyBuilderAccess.RunAndCollect)
    let moduleBldr = asm.DefineDynamicModule("PairToStructConversionModule
        ")

    List.ofSeq(seq {
       for item in pairs.Keys do
          let pairTypes = FSharpType.GetTupleElements(item)
          let structTypeName = "StructPair_" + String.Concat(Array.map (fun
               (t:Type) -> t.Name))
          let typeBldr = moduleBldr.DefineType(structTypeName,
              TypeAttributes.Public)
```

```
        // Define fields
        typeBldr.DefineField("first", ...)
        typeBldr.DefineField("second", ...)

        let attrBldr = CustomAttributeBuilder(
               typeof<CompilationMappingAttribute>.GetConstructor([|
                   typeof<SourceConstructFlags>|]),
               [|box SourceConstructFlags.RecordType|])
        typeBldr.SetCustomAttribute(attrBldr)
        yield typeBldr.CreateType()
    })
```

The conversion process creates proper struct types exploiting the dynamic assembly/-types .NET feature. The struct type name is generated concatenating the types of the pair elements, while the fields are respectively called *first* and *second*.

With the processor we have defined we introduce the ability to discover pairs and replace them with matching struct types. Anyway, we also need to replace each kernel/function parameter and local variable of type "pair of two elements" with the corresponding struct type and to replace pair-related constructs inside the body (e.g. fst(p), which allows to obtain the first element of a tuple) with the matching struct-related construct (e.g. s.first, which is the construct to access the field called "first" of a properly generated struct).

We won't show the implementation of the signature transformation, since it requires quite a lot of coding[1]. We instead show the implementation of the processor whose target is to discover invocations of pair-related functions (*fst* and *snd*) and produce the appropriate construct to access the correct field of the matching struct. Since this is a transformation performed on the kernel/function body, we implement a processor belonging to the *FunctionTransformation* step. The id of this step is $FSCL\_FUNCTION\_TRANSFORMATION\_STEP$ and its processors are of type $CompilerStepProcessor < Expr, Expr >$. Looking at the set of processors of the native function transformation step, we discover that there are no processors that are required to run before/after the custom transformation we are going to implement. Therefore, we have no inter-processor dependencies.

**Listing A.9: Pair access transformation processor draft**

```
[<StepProcessor("PAIR_ACCESS_TRANSFORMATION_PROCESSOR", "
    FSCL_FUNCTION_TRANSFORMATION_STEP")>]
type PairAccessTransformationProcessor() =
   inherit CompilerStepProcessor<Expr, Expr>()
   override this.Process(i: Expr, s: ICompilerStep) =
      ...
```

The *Process* method takes an AST node as input and must return a (potentially transformed) new AST node. In our case, the transformation must be applied only to calls

[1]Programmers interested in implementing this processor should take a look to the signature preprocessing processor in the FSCL compiler sources

to the *fst* and *snd* functions. F# interactive helps to understand the AST structure of such kind of calls:

**Listing A.10: AST structure of pair-related function calls**

```
> <@ fst(a) @>;;
val it : Quotations.Expr<int> =
  Call (None, Fst, [PropertyGet (None, a, [])])
```

We therefore need to check if the input node of the *Process* method is a *Call* to a module function called "Fst" or "Snd". If this is the case, we produce a new AST node that represents the access to the appropriate struct field.

**Listing A.11: Process method definition**

```
\label{code:process_method_def}
override this.Process(i: Expr, s: ICompilerStep) =
   match i with
   | DerivedPatterns.SpecificCall (<@ fst @>) (o, parameters, args) ->
      ...
      // The matching struct object is the first argument
      let propertiesInfo = FSharpType.GetRecordFields(args.[0].Type)
      // The first property is "first", the second is "second"
      Expr.PropertyGet(args.[0], propertiesInfo.[0])

   | DerivedPatterns.SpecificCall (<@ snd @>) (o, parameters, args) ->
      ...
      // The matching struct object is the first argument
      let propertiesInfo = FSharpType.GetRecordFields(args.[0].Type)
      // The first property is "first", the second is "second"
      Expr.PropertyGet(args.[0], propertiesInfo.[1])
```

As illustrated, we obtain the *PropertyInfo* object of the proper struct field and we create a new AST node representing an access to that field. To complete the definition of our transformation processor there are two other aspects that must be defined. The first is what the Process functions should return when the input is an AST node that has not to be processed. For this purpose we can exploit a method exposed by the function transformation processor, which is called *Default*.

```
override this.Process(i: Expr, s: ICompilerStep) =
   match i with
   | DerivedPatterns.SpecificCall (<@ fst @>) (o, parameters, args) ->
      ...
   | DerivedPatterns.SpecificCall (<@ snd @>) (o, parameters, args) ->
      ...
   | _ ->
      (s :?> FunctionTransformationStep).Default(i)
```

Since the transformation step is not able to compare if the output node and the input node are the same, it is important to note that calling *Default* is different than returning the input expression. In the first case, we are telling the step that we don't have to

process the input node but "some other processors might want to process it". Therefore, the input is forwarded to the successive transformation steps. If we instead returned the input untransformed, no other processor wwould be applied to it.

The second aspect to take into account is how to guarantee that nested (child) AST nodes are processed as well as interesting parent nodes. In our case, the interesting nodes are calls to specific functions. Both the arguments and the eventual object instance (if the function is an instance method) are of type *Expr* (i.e. they are subnodes of the function call node). Therefore, we must guarantee that those subnodes are analyzed and eventually transformed before processing the parent node. For this purpose we can exploit another function exposed by the transformation step, which is called *Continue*. We therefore changes the code **??** to accomplish processing subnodes.

**Listing A.12: Process method definition revisited**

```
override this.Process(i: Expr, s: ICompilerStep) =
   let step = s :?> FunctionTransformationStep

   match i with
   | DerivedPatterns.SpecificCall (<@ fst @>) (o, parameters, args) ->
      // We don't need to process "o" because we know fst is a static
          method
      let processedArgs = List.map(fun(e: Expr) -> step.Continue(e)) args
      // The matching struct object is the first argument
      let propertiesInfo = FSharpType.GetRecordFields(processedArgs.[0].
          Type)
      // The first property is "first", the second is "second"
      Expr.PropertyGet(processedArgs.[0], propertiesInfo.[0])

   | DerivedPatterns.SpecificCall (<@ snd @>) (o, parameters, args) ->
      // We don't need to process "o" because we know fst is a static
          method
      let processedArgs = List.map(fun(e: Expr) -> step.Continue(e)) args
      // The matching struct object is the first argument
      let propertiesInfo = FSharpType.GetRecordFields(processedArgs.[0].
          Type)
      // The first property is "first", the second is "second"
      Expr.PropertyGet(processedArgs.[0], propertiesInfo.[1])
```

As illustrated, the first action performed by the processor consists in recursively invoking the transformation step (the sequence of transformation processors) on the subnodes of the input node. After that, we can generate and return the new property-get AST node on the basis of the transformed subnodes.