User's Guide to the Feldspar Compiler

July 26, 2011

Contents

Chapter 1

Introduction

Feldspar (Functional Embedded Language for DSP and PARallelism) is developed in the framework of the joint research project of *Ericsson*, *Chalmers University of Technology* (Göteborg, Sweden) and *Eötvös Loránd University* (Budapest, Hungary).

The final goal of the project is to define a high-level language that allows description of signal processing algorithms and has the following characteristics:

- Allows compact and easy-to-understand definition of algorithms.
- Allows execution and functional verification.
- Hardware platform independent.
- Allows efficient code generation for various targets (at least DSPs and FPGAs).
- Enables future extensions for code generation for multi-core targets in case of computing-intensive algorithms.
- Supports code generation for multiple streams.
- Supports specification of external interfaces as C functions, and supports inclusion of existing components.
- Supports merging, re-grouping, splitting of algorithms and groups of algorithms.
- Supports expression of control information handling.

This document is the user's guide for the prototype compiler and related tools. For the documentation of the Feldspar language itself, see the *User's Guide to the Feldspar language*. For installation instructions the reader is refered to the *Installation Guide*.

This version of the compiler supports ISO C99 code generation from Feldspar functions. There are many optimization techniques, including loop unrolling, implemented by the compiler using a dedicated plugin framework. For developers, tracing support is also provided.

1.0.1 Log of Changes

Current release includes the following improvements compared to version 0.3:

- Support for version 0.4 of the Feldspar language.
- Compilation of Feldspar-level pairs.
- Compilation of complex numbers.

- Handling arbitrary number of function parameters.
- Improved TI platform support.
- API for standalone compilation.

Changes in version 0.3:

- Support for version 0.3 of the Feldspar language.
- Limited support for the TMS320C64x chip family.
- Automatic generation of fixed-point and floating-point C code from generic Feldspar functions.
- Support for trace functions.

Chapter 2

Invoking the Compiler

2.1 Standalone Compilation from Command Line

Synopsis:

```
feldspar [options] FeldsparSource.hs
```

When the compiler invoked it will give some feedback on the progress of compilation (see the example outputs below).

Notes:

- When no output file name is specified, the input file name with `.c` extension is used.
- The input file parameter is mandatory, even in single-function mode (see below).

2.1.1 General Structure of Feldspar Programs

A typical Feldspar program can contain numerous functions: Functions using the types provided by the Feldspar library and ordinary Haskell functions as helper functions. The Feldspar compiler will compile functions with signature consisting of Feldspar types only.

For the sake of convenience, multi-function compilation mode is the default in the standalone compiler. The compiler will try to compile all functions, telling the user if the compilation of a function failed but nevertheless continuing the code generation process. See the examples section for an example output of the compiler.

2.1.2 Available Options

The following options can be used to tweak the settings of the standalone compiler.

```
-f <function>
--singlefunction=<function>
```

Enables single-function compilation. In order to make it work the name of the function has to be specified.

```
-o <output_file.c>
--output=<output_file.c>
```

Overrides the file name for the generated output code.

```
-p <platform>
--platform=<platform>
```

Overrides the target platform. Valid options are: `C99`, `Tic64x`. Please refer to Section 2.3 for more information.

```
-u <unroll count>
--unroll=<unroll count>
```

Enables loop unrolling with the specified unroll count.

```
-D <debug level>
--debuglevel=<debug level>
```

Specifies debug level. Currently the only possible option is `NoPrimitiveInstructionHandling`.

```
-h
--help
```

Shows the help message.

2.1.3 Examples

Here are some examples on using the compiler in standalone mode. Here `$` denotes the normal command-line prompt without any special privileges.

- Standard code generation (compiling all functions).

```
$ feldspar FeldsparSource.hs
```

- Standard code generation (compiling only one function).

```
$ feldspar -f functionName FeldsparSource.hs
```

- Code generation to `code.c` with loop unrolling enabled.

```
$ feldspar -o code.c -u 8 FeldsparSource.hs
```

- Getting help.

```
$ feldspar --help
```

Example Outputs

- Multi-function compilation.

```
$ feldspar Examples/Simple/Basics.hs
==== [ Compilation target: module Examples.Simple.Basics                    ] ====
==== [ Output file: Basics.c                                                ] ====
==== [ Number of functions to compile: 10                                   ] ====
[WARNING @ PluginArch/Precompilation]: not enough named parameters in function example1
numArgs: 1, parameter list: []

================ [ Summary of compilation results ] ================
```

```
Function example1 ..... [OK]
Function example2 ..... [OK]
Function example3 ..... [OK]
Function example4 ..... [OK]
Function example5 ..... [OK]
Function example6 ..... [OK]
Function example7 ..... [OK]
Function example8 ..... [OK]
Function example9 ..... [OK]
Function example10 ..... [OK]
```

- Single-function compilation.

```
$ feldspar -f example1 Examples/Simple/Basics.hs
=== [ Compilation target: module Examples.Simple.Basics ] ===
=== [ Output file: Basics.c ] ===
=== [ Compiling function example1... ] ===
[WARNING @ PluginArch/Precompilation]: not enough named parameters in function example1
numArgs: 1, parameter list: []
```

2.2 Interactive Compilation Using GHCi

2.2.1 Loading the Compiler's Module

Load a source file into the Haskell interpreter first.

```
$ ghci source_file
>
```

To call the Feldspar compiler from *GHCi*, `Feldspar.Compiler` is required. It is possible to import it in the source file.

```
import Feldspar.Compiler
```

The other solution is loading it in the interpreter using the following command.

```
> :module +Feldspar.Compiler
>
```

2.2.2 Generating C Code into Files

In the interpreter, Feldspar functions can be compiled using the following command.

```
> compile <function> <filename> <function name> <options>
```

The `compile` function has four arguments.

- A Feldspar function to compile.
- Name of the output file to save the generated code in (String).
- Name of the C function to generate (String).
- Compilation options.

For example, take the following function from `Examples/Simple/Basics.hs`.

```
example1 :: Data Int32 -> Data Int32
example1 = id
```

Compile this function and write the C output into a file named `simple.c`.

```
> compile example1 "simple.c" "simple" defaultOptions
```

The generated file will have the following contents.

```
#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <complex.h>

/*
 * Memory information
 *
 * Local: none
 * Input: signed 32-bit integer
 * Output: signed 32-bit integer
 *
 */
void simple(struct array mem, int32_t in0, int32_t * out1)
{
    (* out1) = in0;
}
```

2.2.3 Using the Compiler Interactively

The `icompile` (*interactive compile*) function returns the generated C code to the Haskell interpreter instead of writing it into file.

```
> icompile <function>
```

For the sake of convenience, the function requires only one argument, a Feldspar function to be compiled. The generated C function's name defaults to `"test"` while the compilation is driven by the contents of `defaultOptions`.

However, by using the `icompile'` function, it is possible to specify all the compilation arguments.

```
> icompile <function> <function name> <options>
```

Note that arguments are the same as `compile`'s, but there is no need for a file name.

As an example, compile the `example3` function (also from `Examples/Simple/Basics.hs`).

```
> icompile example3
```

The resulting output can be seen directly in the interpreter.

```
#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
```

```
#include <string.h>
#include <math.h>
#include <complex.h>


/*
 * Memory information
 *
 * Local: none
 * Input: none
 * Output: signed 32-bit integer array(4)
 *
 */
void test(struct array mem, struct array * out0)
{
    setLength(out0, 4);
    at(int32_t,(* out0),0) = 42;
    at(int32_t,(* out0),1) = 1;
    at(int32_t,(* out0),2) = 2;
    at(int32_t,(* out0),3) = 3;
}
```

2.3 Compiler Options

`Options` is a Haskell data type which holds compilation options to be passed to the compiler. Modules `Feldspar.Compiler.Backend.C.Options` and `Feldspar.Compiler.Backend.C.Platforms` have to be imported first to define these options.

```
> :module +Feldspar.Compiler.Backend.C.Options Feldspar.Compiler.Backend.C.Platforms
```

Then such a value can be defined as follows.

```
> let customOpts = Options <platform> <loop unrolling> <debugging> <memoryInfoVisible>
```

Currently supported values for the different options are as follows.

- Platform:
  - `c99`: Use the ISO/IEC C99 standard. (The support is currently limited to modified function signatures to help C99-enabled compilers to produce faster target code.)
  - `tic64x`: Use C extensions of Texas Instruments for the TMS320C64x chip family.

  For more information on how to define platforms see Section 2.4.
- Loop unrolling:
  - `NoUnroll`: Avoid loop unrolling.
  - `Unroll` n: Unroll `for` loops n times (where n is an integer).
- Debugging:
  - `NoPrimitiveInstructionHandling`: Stop after generating imperative program constructions, do not translate primitive instructions.
    Using this option may yield incorrect C code, it is supposed to be used for debugging purposes only.
  - `NoDebug`: Disable debugging, apply all compilation stages.

- AllocationInfo:
  - `True`: Generate a comment before the function, containing the memory allocation information. (as in previous examples)
  - `False`: Disable this option.

Note that there are predefined `Options` constants are exported by the `Feldspar.Compiler` module.

- `defaultOptions`, `c99PlatformOptions`: Support for the ISO C99 standard. All compilation steps are performed and no loop unrolling happens.
- `tic64xPlatformOptions`: Support for the Texas Instruments TMS320C64 chip family. All compilation steps are performed and no loop unrolling happens.
- `unrollOptions`: All compilation steps are performed and innermost for loops are unrolled 8 times.
- `noPrimitiveInstructionHandling`: Turn on the `NoPrimitiveInstructionHandling` debugging option.
- `noMemoryInformation`: Turn off the `memoryInfoVisible` option.

2.4 Platform Definitions

User-defined platforms can be specified by filling in a value of type `Platform` with the following fields. (All described data types are defined in `Feldspar.Compiler.Options`.)

```
data Platform = Platform {
    name          :: String,
    types         :: [(Type, String, String)],
    values        :: [(Type, ShowValue)],
    primitives    :: [(FeldPrimDesc, Either CPrimDesc TransformPrim)],
    includes      :: [String],
    isRestrict    :: IsRestrict
}
```

- `name`: Name of the platform.
- `types`: Mapping between Feldspar types and platform types. This is a tuple with three fields. The first field is a Feldspar type using the compiler's representation, the second is the name of the corresponding type on the given platform, the third one is a logical name of the type witch may appear in function names. (See the description of the `primitives` field for the details of function naming conventions). The Haskell definition of the types that can be used is the following.

  ```
  data Type =
        VoidType
      | BoolType
      | BitType
      | FloatType
      | NumType Signedness Size
      | ComplexType Type
      | UserType String
      | ArrayType Length Type
      | StructType [(String, Type)]

  data Size = S8 | S16 | S32 | S40 | S64
  ```

```
data Signedness = Signed | Unsigned
```

- `values`: Functions to define the format of the constants in generated code. This is a pair where first is the type of the constant, the second is a function which creates a string from the constant. There are defaults for `Bool`, `Float`, and `Integer` types but it can be overridden.
- `primitives`: Mapping between Feldspar's primitives and operators, functions available on the platform. Actually this is a list of pairs which contains descriptions of the Feldspar and the corresponding platform primitives or an explicit transformation function.
  - `FeldsparPrimDesc`: Feldspar primitive

```
data FeldPrimDesc = FeldPrimDesc
    { fName :: String
    , inputs :: [TypeDesc]
    }

data TypeDesc
    = AllT
    | BoolT
    | RealT
    | FloatT
    | IntT | IntTS | IntTU | IntTS_ Size | IntTU_ Size | IntT_ Size
    | ComplexT TypeDesc
    | UserT String
```

    It contains the name of the primitive, and a pattern to the type of its parameters. `AllT` matches all type, `IntTS` matches signed integers `IntTU` matches unsigned and so on.

  - `CPrimDesc`: C primitive.

```
data CPrimDesc
    = Op1 {cOp :: String}
    | Op2 {cOp :: String}
    | Fun {cName :: String, funPf :: FunPostfixDescr}
    | Proc {cName :: String, funPf :: FunPostfixDescr}
    | Assig
    | Cas
    | InvalidDesc

data FunPostfixDescr = FunPostfixDescr
    { useInputs :: Int
    , useOutputs :: Int
    }

noneFP      = FunPostfixDescr 0 0
firstInFP   = FunPostfixDescr 1 0
firstOutFP  = FunPostfixDescr 0 1
```

    * `Op1` and `Op2`: C operator with one or two parameters. In this case the name of the operator (e.g. `"=="`) is specified.
    * `Fun` and `Proc`: C function or procedure. The name of the function/procedure is specified in the following way.

      `cName` is a base name which is extended by the types of some of the parameters. The parameters to use for this purpose is defined by `funPf`. (This is important as C does not support function overloading. In most cases, using the type of the first input or output parameter (`firstInFP`, `firstOutFP`) is enough.)

    * `Assig`: C assignment.
    * `Cas`: C casting.
  - `TransformPrim`: Transformation function.

```
type TransformPrim
    = FeldPrimDesc
    -> [Expression ()]
    -> Type
    -> PrgDesc

data PrgDesc
    = PrgDesc [Crt] [Line] Rgt

data Crt
    = Crt Type Var (Maybe Rgt)

data Line
    = Asg Var Rgt
    | Prc CPrimDesc [Rgt] [Var]

data Rgt
    = Exp (Expression ())
    | Fnc CPrimDesc [Rgt] Type
    | VarR Var

data Var
    = Var String
```

    Parameters of this function are the description of the Feldspar primitive and the lists of input and output parameters. The result of the function is a list of C primitives, each with its list of input and output parameters.

  In order to compile a Feldspar primitive to C, the compiler chooses the first record from the `primitives` list with matching `FeldsparPrimDesc`. The record contains either a `CPrimDesc` or a transformation function to generate the resulting C primitive.

- `includes`: A list of the header files to include.
- `isRestrict`: Specifies if the `restrict` keyword is supported in the platform. Possible values are `Restrict` and `NoRestrict`.

Built-in platform descriptions can be found in `Feldspar/Compiler/Platform.hs` that may be re-used when implementing a new platform desription.

```
...
primitives = new_primitives ++ primitives c99
...
```

Note that passing user-defined platform descriptions to the compiler is currently supported only in interactive compilation mode. (see Section 2.2).

Chapter 3

Further Features

3.1 Loop Unrolling

Loop unrolling is an optimization technique traditionally used to decrease the number of jumps during execution. The Feldspar Compiler provides a limited and experimental support for unrolling `for` loops. As C compilers can also unroll loops, this feature of the Feldspar Compiler alone may not increase efficiency. The goal of this experimental feature is to make further optimization transformations possible: In later releases, the instructions of an unrolled loop may be reordered such that more of them can be replaced by one, more efficient instruction specific to the target platform.

The Feldspar Compiler unrolls a `for` loop only if it consists of a sequence of primitive instructions. If the number of its iterations is known at compile time, the loop is unrolled only if the number of iterations is a multiple of the value given in the `Unroll` option. If the number of iterations is not known at compile time, unrolling is done *without this check*. Note that, as a consequence, it is not guaranteed that loop unrolling performed by the Feldspar Compiler will maintain behavioral equivalence.

Loop unrolling can be turned on in the `unroll` field of the options by setting an n that is to be passed to the compiler with `Unroll` where n is a non-negative integer. Using the predefined `unrollOptions` is one way to achieve this. For details of using this method, see Section 2.3.

For illustrating the differences, take a simple Feldspar program.

```
vector1 :: DVector Index
vector1 = map (*10) $ reverse $ map (+3) $ enumFromTo 1 16
```

When it is compiled with `defaultOptions` (no loop unrolling) the corresponding generated C code will be as follows.

```
#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <complex.h>



/*
 * Memory information
 *
```

```
 * Local: none
 * Input: none
 * Output: unsigned 32-bit integer array(16)
 *
 */
void test(struct array mem, struct array * out0)
{
    setLength(out0, 16);
    {
        uint32_t i1;
        for(i1 = 0; i1 < 16; i1 += 1)
        {
            at(uint32_t,(* out0),i1) = ((((15 - i1) + 1) + 3) * 10);
        }
    }
}
```

But when it is compiled with `unrollOptions` the resulted C code will be as follows.

```
#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <complex.h>


/*
 * Memory information
 *
 * Local: none
 * Input: none
 * Output: unsigned 32-bit integer array(16)
 *
 */
void test(struct array mem, struct array * out0)
{
    setLength(out0, 16);
    {
        uint32_t i1;
        for(i1 = 0; i1 < 16; i1 += 8)
        {
            at(uint32_t,(* out0),i1) = ((((15 - i1) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 1)) = ((((15 - (i1 + 1)) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 2)) = ((((15 - (i1 + 2)) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 3)) = ((((15 - (i1 + 3)) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 4)) = ((((15 - (i1 + 4)) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 5)) = ((((15 - (i1 + 5)) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 6)) = ((((15 - (i1 + 6)) + 1) + 3) * 10);
            at(uint32_t,(* out0),(i1 + 7)) = ((((15 - (i1 + 7)) + 1) + 3) * 10);
        }
    }
}
```

3.2 Tracing

The `trace` function provides support for tracing Feldspar functions. From the functional point of view, this function is the identity, thus it can be inserted in the Feldspar program without affecting the computation. When C code is generated, the compiler will insert logging function calls in order to log the actual value of the traced variable together with some other information in the `trace-`*`date-time`*`.log` file in the current directory. Each row of this file contains an identifier of the trace call, the value of the traced variable and time elapsed since startup.

```
trace :: Int -> Data a -> Data a
```

The first parameter is the identifier of the trace call, the second is the variable to log the actual value of. For vectors you can use `trace` combined with `map`.

Let us see an example on how to use this function in Feldspar programs.

```
traceExample :: Data [Int32] -> Data Int32
traceExample xs
    = fold (\x y -> trace 3 $ trace 1 x + trace 2 y) 0 $ unfreezeVector' 255 xs
```

Here is the resulting C code with trace calls.

```
#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <complex.h>


/*
 * Memory information
 *
 * Local: none
 * Input: signed 32-bit integer array(255)
 * Output: signed 32-bit integer
 *
 */
void test(struct array mem, struct array in0, int32_t * out1)
{
    int32_t temp2;

    traceStart();
    (* out1) = 0;
    {
        uint32_t i3;
        for(i3 = 0; i3 < 255; i3 += 1)
        {
            int32_t trc0;
            int32_t trc1;
            int32_t trc2;

            trc0 = (* out1);
            trace_int32(trc0, 1);
            trc1 = at(int32_t,in0,i3);
            trace_int32(trc1, 2);
            trc2 = (trc0 + trc1);
            trace_int32(trc2, 3);
```

```
            temp2 = trc2;
            (* out1) = temp2;
        }
    }
    traceEnd();
}
```

Finally here is an excerpt of a sample log file generated by the C program above.

```
Logging started at 08-Dec-2010 17:11:42.
id=1, time=0.000007, value=0
id=2, time=0.000058, value=0
id=3, time=0.000105, value=0
id=1, time=0.000143, value=0
id=2, time=0.000178, value=1
id=3, time=0.000213, value=1
id=1, time=0.000248, value=1
id=2, time=0.000283, value=2
id=3, time=0.000318, value=3
id=1, time=0.000353, value=3
id=2, time=0.000387, value=3
...
id=1, time=0.028212, value=31626
id=2, time=0.028248, value=252
id=3, time=0.028283, value=31878
id=1, time=0.028318, value=31878
id=2, time=0.028353, value=253
id=3, time=0.028388, value=32131
id=1, time=0.028425, value=32131
id=2, time=0.028471, value=254
id=3, time=0.028509, value=32385
Logging finished.
```

3.3 Array support

3.3.1 The structure of arrays

Arrays in Feldspar-generated C programs use a struct called `array`. This resides in `feldspar_array.c` and `feldspar_array.h`. Its structure is as follows:

```
struct array
{
    void* buffer;
    unsigned int length;
    int elemSize;
};
```

The first field, `buffer` is a pointer to the buffer of elements, i.e. to the data itself. The second field, `length` stores the number of elements in the array. The third field, `elemSize` stores the size of elements in bytes. In the case of nested arrays, its value is -1.

3.3.2 Using arrays

For all Feldspar-generated C functions, the first parameter of the function is a parameter called `mem`, of type `struct array`. This parameter is used by the C function for temporary data storage. For example:

```
/*
 * Memory information
 *
 * Local: complex float array(256), complex float array(256)
 * Input: complex float array(256)
 * Output: complex float array(256)
 *
 */
void test(struct array mem, struct array in0, struct array * out1)
{
    ...
}
```

In order to use the function, the `mem` parameter has to be initialized. The memory information located in a comment block above the C function helps with this task. In the above example, the memory information says that the `mem` parameter has to contain two elements, both of type `complex float array` (see "Local"), i.e. two array structures containing complex floats. If the compiler can determine the sizes of the arrays needed, the memory information will also include this information. Otherwise, the sizes need to be determined by the developer based on the algorithm implemented in the function. In the above example, both arrays in the temporary storage are of size 256, and the size of the input and output arrays is also 256.

The initialization of the `mem` parameter can be done as follows:

```
#include <stdio.h>
#include <complex.h>
#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <inttypes.h>
#include <stdlib.h>

void test(struct array mem, struct array in0, struct array * out1)
{
    // ...
}

int main(int argc, char *argv[])
{
    // Initialization of the temporary storage (mem)
    struct array mem_row0;
    float complex mem_row0_Storage[256];
    mem_row0.buffer = &mem_row0_Storage;
    mem_row0.length = 256;
    mem_row0.elemSize = sizeof(float complex);

    struct array mem_row1;
    float complex mem_row1_Storage[256];
    mem_row1.buffer = &mem_row1_Storage;
    mem_row1.length = 256;
    mem_row1.elemSize = sizeof(float complex);

    struct array mem;
```

```
    struct array mem_Storage[2];
    mem.buffer = &mem_Storage;
    mem.length = 2;
    mem.elemSize = -1; // because it contains arrays

    at(struct array,mem,0) = mem_row0;
    at(struct array,mem,1) = mem_row1;

    // Initialization of input and output parameters
    struct array inp_0;
    float complex inp_0Storage[256];
    inp_0.buffer = &inp_0Storage;
    inp_0.length = 256;
    inp_0.elemSize = sizeof(float complex);

    struct array out_1;
    float complex out_1Storage[256];
    out_1.buffer = &out_1Storage;
    out_1.length = 256;
    out_1.elemSize = sizeof(float complex);

    // Call the function and exit
    test(mem, inp_0, &out_1);
    return 0;
}
```