Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 149–156.

Feldspar – A Functional Embedded Language for Digital Signal Processing*

Gergely Dévai, Zoltán Gera, Zoltán Horváth, Gábor Páli, Máté Tejfel

Eötvös Loránd University, Faculty of Informatics
{deva, gerazo, hz, pgj, matej}@inf.elte.hu

Abstract

Digital Signal Processing (DSP) algorithms are usually designed and described on an abstract level and than transformed to a DSP chip specific C code by expert programmers. The problem is that the gap between the abstract description and the platform dependent code is huge and even the C code optimized for two different chips differ a lot. This makes it expensive to rewrite the algorithms each time a new chip is targeted.

We expect that designing and implementing a high-level domain-specific language (DSL) will make the implementation of algorithms easier and a compiler together with platform-specific code generator and optimizer modules will take the burden of target-dependent low-level programming off the programmers.

To address these problems, we propose a new programming language called *Feldspar* (Functional Embedded Language for Digital Signal Processing and Parallelism). We started the design and implementation of this language a year ago, and now we have a compiler prototype which translates programs written in Feldspar into hardware-indepdendent ISO C99-compilant code.

Keywords: Digital Signal Processing, Haskell, Language Embeddeding, Deforestation, Compile-Time Optimization

MSC: 68N20 (Compilers and Interpreters)

1. Introduction

Digital signal processing (DSP) algorithms are usually implemented as highly optimized C or assembly programs on special purpose hardware platforms. Production of such software involves many local manual optimizations in the code with lot

*Supported by KMOP-1.1.2-08/1-2008-0002, Ericsson and Ericsson Software Research, POSDRU/6/1.5/S/3/2008

of knowledge on the given platform in mind, which is rather expensive most of the time. As a result, it is typically hard to maintain the source code, and, in particular, hard to port to new platforms.

Feldspar is being developed as a domain-specific language specifically for describing DSP algorithms, taking into account the specific constructions appearing in the problem domain. Feldspar expresses algorithms in an abstract declarative style instead of using any hardware-dependent low-level programming constructs. In that sense, Feldspar is both extended and restricted compared to general-purpose programming languages at the same time: it has extra features specific to the captured problem domain, making the development easier with only a limited set of language elements, which also enables more optimized compilation. Our initial experiments have shown that it is easier to design and implement algorithms on this level of abstraction compared to that of C or assembly.

Our research project is being carried out as a joint effort of Ericsson and Ericsson Research, Eötvös Loránd University (Budapest, Hungary), and Chalmers University of Technology (Gothenburg, Sweden) [1]. Our goal is to develop a high-level domain-specific language designed for digital signal processing algorithms and to implement a prototype compiler and related tools (e.g. debugger) for the language, together with some support for deployment in a real-world environment in the future.

The structure of this paper is organized as follows. In Section 2, the high-level layer of the Feldspar language is introduced in terms of internal representation and techniques applied on higher level optimizations. Section 3 describes the translation of Feldspar core programs into abstract imperative programs and the optimization transformations performed on them. Section 4 compares our approach with some related work in the field, and gives a brief summary on the results we have achieved so far.

2. Frontend, Generic Optimizations

Designing and implementing a new programming language from scratch is challenging and takes time usually. Nowadays one might find fine lexer and parser generators but working with type checkers is still an experimental topic. It is important for Feldspar as it is a statically typed programming language, i.e. checked in compile-time.

Embedding [4] is a method where a new language is implemented by using an existing one, called *host language*. As a consequence, there is no need for a lexer, parser or type checker as these tasks are performed by the compiler of the host. In this case, a new language can be defined as a library consisting of functions not implementing actual computations but resulting in a data structure, which is technically an abstract syntax tree of the embedded program. The compiler of the embedded language takes this data structure as input, manipulates it and generates code for the target language. Embedding enables various kinds of compile-time simplifications in a very natural way and it offers quick language prototyping.

```
sumSquares :: Data Int -> Data Int
sumSquares n =
  sum $ map square $ 1 ... n
    where square x = x * x
```

Figure 1: Feldspar program `sumSquares`

Feldspar uses the Haskell functional programming language as a host for embedding. Haskell is a perfect choice for working with data structures since it abstracts away from pointers and includes many compiler optimizations connected to high-level description of data structures, called data types. Static typing and functional nature of the language are also inherited from there. Implementation of Feldspar provides two libraries: The *core language* and its high-level interface, the *vector library*. Programmers need to import these Haskell libraries and write Haskell programs which use their exported functions in order to write Feldspar programs. The core language introduces imperative-like constructs (for instance, loops) with pure functional semantics, while the vector library reimplements some of the standard Haskell list functions to be able to transform them to a core language program.

The high-level library interface visible for the user is mainly based on the `Vector` data type. Vectors can be manipulated by functions similar to list operations in Haskell. Vectors can also be treated as subscripted sequences, allowing programs to be written like they were presented in a pure mathematical manner. An example of using vectors for representing formulas would be the calculation of sum of squares between 1 and n.

Figure 1 shows a Feldspar program (`sumSquares`) implementing this with a list-like structure. The first line is a type signature declaring that the function accepts an integer argument and returns an integer. The right-hand side of the definition should be read from right to left: It starts by forming a vector containing elements from `1` to `n` then each element is mapped to its square using `map` [1] then finally the elements of the squared vector are summed. Note that `sumSquares` is written as *composition* of simple functions, which is typical for functional programs.

Though `sumSquares` seems to be a program that works with integer data, it is actually a *program generator* that produces another program upon its evaluation which will compute the specified function. Programs generated this way are formulated by the core language of Feldspar which serves as an interface towards to the backends. In this sense, the actual Feldspar programs refer to construction of abstract syntax trees rather then the computation itself. Then these trees are received, transformed, and translated further by the specific backend implementations.

Core language constructs are fairly easy to translate to a machine-oriented language due to their imperative nature. Use of vectors in `sumSquares` may raise worries on excessive memory consumption of the resulting program. However, this

[1] Function `map` is a higher order operation for applying a function to each element in a vector.

can be avoided, since all the vector operations can be "fused" together. This level of efficiency is obtained by removal of intermediate data structures, a technique known as *deforestation* or *fusion* [3].

Fusion techniques enable the implementation to compute vector elements on demand, and source programs are mostly simplified in compile-time. Most functions virtually rewrite the input program on a macro level. There are several stages where a Feldspar program gets rewritten.

Vector operations might be defined by manipulating the length and the index function only. According to this observation, we can give the definition of type `Vector` below.

```
data Vector a = Indexed (Data Length) (Data Index -> a)
```

A vector is essentially a pair of a length and an index function which maps each index to an element in the vector. Thus vectors do not appear in the memory in their full lengths. Let us take the definition `map` as an example to show how to work with such representations. [2] The second parameter is a vector, where we are matching a pattern, i.e. decompose the contents of the passed value. According to the result of the pattern matching, `l` will hold the length, and `ixf` will be the index function of the given vector.

```
map f (Indexed l ixf) = Indexed l (f . ixf)
```

Applying `map` on an arbitrary `f` function and `v` vector results in a new vector of the same length as `v` but with a different index function. The new index function is simply a function composition of `f` and the old index function.

Representing vector by their index functions has the benefit of allowing a mathematical style of programming. For instance, a mathematical definition of `map square a` might be given as it is shown below.

$$b_i = a_i * a_i, \qquad \text{where } i \in \{\ 0,\ N-1\ \}$$

This formula translates to Feldspar almost seamlessly.

```
b = Indexed (length a) $ \i -> (a ! i) * (a ! i)
```

The `sum` function can be written using a `for` loop in the core language.

```
sum (Index l ixf) = for 0 (l - 1) 0 $ \i s -> s + ixf i
```

Arguments of `for` are as follows: a start index, an end index, and a starting state respectively followed by a body as an anonymous function parameterized with the actual index and the current state accumulated over the loop. A vector element at index `i` is added to the running sum in each iteration. At the moment `for` loops are just macros and they are translated to `while` loops in the core language, but they will have their own translation in the future.

The `1 ... n` expression can be viewed as a function generating a `Vector` with `...` as an infix operator.

[2] The symbol `.` here refers to the mathematical composition function.

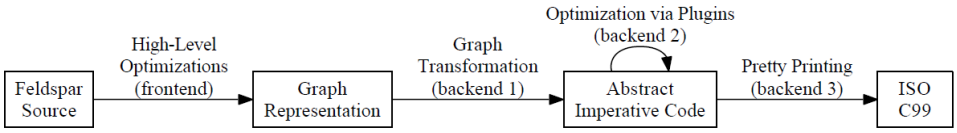

Figure 2: Internal Structure of the Feldspar Compiler

```
1 ... n = Indexed n $ \i -> i + 1
```

By combining `map` with definition of `square` and `1 ... n`, the high-level compiler builds a simplified program out of the `sumSquare` function. Vectors will completely disappear there and their index functions will be merged into the body of the `for` loop.

```
sumSquares n = for 1 n 1 $ \i s -> s + (i * i)
```

3. Backend, Target Optimizations

In general, target language backends produce platform-dependent code from the intermediate representation that core programs represent on a higher level. At the moment a backend for generating C programs is under development, however there is the possibility for developing others as well later on. In this paper, we will describe a C code generator developed at ELTE. A compiler structure including backend is shown in Figure 2, where Feldspar programs are compiled further to C with an input of a core language program which is represented as a graph.

Graph is transformed into an *abstract imperative code* first. Note that this is no longer purely functional. The transformation is straightforward because local variable declarations, primitive operations, branches, and loops are represented as nodes in the input graph and they can be mapped directly to the abstract imperative counterparts. Initialization and update of loop states are also taken care of there. Type system of abstract imperative code is different from that of C. A range of fixed size 8-, 16-, 32-, and 64-bit signed and unsigned integer types are used, arrays with fixed length, floating-point, and boolean types. Mapping these types to the corresponding C types heavily depends on the given platform. In C, pointers and operations on pointers are used to handle parameter passing and arrays. This makes implementation of optimizations harder, therefore the abstract imperative code hides pointer operations: For each parameter and local variable its *logical type* and *role* (input, output, or local) are stored that tells how a parameter or variable should appear in certain positions in the C code.

The result of the first pass is a naïve program, shown in Figure 3, which might be subject to several optimization steps. As it is directly translated from a core language program it includes many variables from the abstract imperative code. Some of them comes from rewriting the `for` loop as a `while` loop, therefore it

uses some additional subexpressions for describing the condition of termination and counting [3], while the C transcript uses few more other variables to implement update of these variables properly. Fortunately most of them can be eliminated: *Copy propagation* is a well-known solution for replacing occurrences of variables with expressions assigned to them. However, this may be disadvantageous when multiple occurrences of a variable is replaced with a larger expression. To avoid such problems, we decided to use the following heuristics: A variable is eliminated if either the expression assigned to it is relatively cheap to compute or there is only one occurrence to be replaced. The Feldspar compiler also supports backward propagation of copies, which is essential to eliminate copying of arrays.

A number of different hardware-dependent optimizations might be performed as well. These algorithmic transformations are part of the hardware-specific backend of the compiler, thus they may vary between different hardware types. There are a number of common tricks which can be utilized generally. The Feldspar compiler is designed to do the same optimizations automatically as a programmer would do when porting the given algorithm to a new platform. An example of such hardware-dependent manual tweaks is *loop unrolling* which is a trade-off between speed and code size with optima depending on hardware flavors. It is also important for the Feldspar compiler to gain extra performance since it is able to exploit the on-chip parallelization offered by most digital signal processors when number of internal execution units is taken into account for unrolling loops. The transformed code can be refined further by inserting `restrict` keywords at the right places. This ensures the compiler pointers will not alias each other [4] which is another potential speedup. Instructions in an unrolled loop body may be re-ordered and grouped in the future in order to be replaced for hardware-specific intrinsics, which would be a key in boosting performance to a level matching with hand-written C programs.

Optimizations are implemented in a highly modular way as plugins of a framework. Each plugin consists of an analysis and a transformation routine. Analysis determines whether the given transformation is valid to specific parts of the program, and collects semantical information. Different types of semantical information can be attached to the program nodes in a polymorphic fashion. After all the information required for the optimization is gathered, the transformation takes place. Both analysis and transformation need to traverse the program's abstract syntax tree and perform some kind of computation at each node. The implemented framework supports both of these activities, where only the plugin-specific functions have to be implemented. This approach delivers easier creation and maintenance of the plugins.

When all the optimization steps are run, hardware-independent valid C99 code is produced via a pretty printer. Detailed description of the language and compiler internals, including some performance measurements and comparisons can be found in our recent paper [8].

[3] It can be eliminated by having a specialized implementation for `for` loops as future work.
[4] It is automatically guaranteed by the Feldspar compiler.

```
void sumSquares( signed int var0, signed int * out ) {
    signed int var2, var3, var4, var11_0, var11_1;

    var2 = (var0 - 1);
    var3 = (var2 + 1);
    var4 = (var3 - 1);
    var11_0 = 0;
    var11_1 = 0;
    {
        signed int var1_0, var1_1;
        int var5;

        var1_0 = var11_0;
        var1_1 = var11_1;
        var5 = (var1_0 <= var4);
        while ( var5 ) {
            signed int var6_0, var6_1, var7, var8;
            signed int var9, var10;

            var6_0 = var11_0;
            var6_1 = var11_1;
            var7 = (var6_0 + 1);
            var8 = (var6_0 + 1);
            var9 = (var8 * var8);
            var10 = (var6_1 + var9);
            var11_0 = var7;
            var11_1 = var10;
            var1_0 = var11_0;
            var1_1 = var11_1;
            var5 = (var1_0 <= var4);
        }
    }
    (*out) = var11_1;
}
```

```
void sumSquares( signed int var0, signed int * out ) {
    signed int var11_0;

    var11_0 = 0;
    *out = 0;
    {
        while ( (var11_0 <= (((var0 - 1) + 1) - 1)) ) {
            signed int var8;

            var8 = (var11_0 + 1);
            var11_0 = (var11_0 + 1);
            *out = (*out + (var8 * var8));
        }
    }
}
```

Figure 3: Naïve *(left)* and Optimized *(right)* Version of `sumSquares` in C

4. Related Work and Summary

We have presented a domain-specific language for digital signal processing embedded into the functional programming language Haskell. We created a code generator for it which is capable of translating Feldspar programs to ISO C source code with several compile-time optimizations.

Translation of high-level models into efficient C program has been extensively investigated previously in several researches as well as in a book on optimizing source code for data flow dominated embedded software [5]. Potential pitfalls and problems of ISO C source code generaion is already described by Bhattacharyya [2], who proposes a structure for such compilers. It implements all the components of an ordinary compiler that we omitted here. The employed graph-based representation is similar to our core language, and it is also optimized before code generation. The design supports mapping internals to specific machine instructions, in our case this is future work. Currently, the compiler framework introduced here does not support optimizations for compilers of various DSP chips. However, SPIRAL [6] proposed a notable model for code generation for different target platforms. It is built around a formula description language called SPL [7] and its compiler implementing a feedback-driven optimizer. It musters up many ideas on how to deploy new optimization techniques, platforms, and target performance metrics. Without providing the same, Feldspar might lose on the efficiency. To regain some

of these losses, the compiler should include more knowledge on the target platforms, together with more general- and platform-specific optimizations.

Our contribution is to design and implement a high level hardware-independent language with optimizations on both higher and lower levels. It makes easier to write and maintain signal processing applications, while the performance of the automatically generated code is comparable to the manually tweaked C reference implementations. The developed plugin framework allows hardware-dependent optimizations to be easily integrated as the development goes on.

5. Acknowledgments

We would like to acknowledge the financial support of Ericsson Software Research, Ericsson Business Unit Networks and SSF (Sweden), the Hungarian National Development Agency (KMOP-2008-1.1.2) as well as the Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007-2013 (POSDRU/6/1.5/S/3-2008, Romania). We would like to thank Emil Axelsson and all those who contributed to this paper or to the development of Feldspar with their comments.

References

[1] Feldspar Home Page: `http://dsl4dsp.inf.elte.hu/`

[2] S. S. Bhattacharyya, R. Leupers, P. Marwedel, Software Synthesis and Code Generation for Signal Processing Systems, *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, (1999), 849–875.

[3] D. Coutts, R. Leshchinskiy, D. Stewart, Stream Fusion: From Lists to Streams to Nothing at All, *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, (2007), 315–326,

[4] P. Hudak, Modular Domain Specific Languages and Tools, *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, (1998), 134.

[5] H. Falk, P. Marwedel, Source Code Optimization Techniques for Data Flow Dominated Embedded Software, *Kluwer Academic Publishers*, 2004.

[6] M. Püschelm, J. M. F. Moura, J. Johnson, et al, SPIRAL: Code Generation for DSP Transforms, *Proc. IEEE, Vol. 93, No. 2*, (2005), 232–275.

[7] J. Xiong, J. Johnson, R. Johnson, D. Padua, SPL: A Language and Compiler for DSP Algorithms, 2001.

[8] G. Dévai, M. Tejfel, Z. Gera, G. Páli, Gy. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård, A. Persson, Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs, *Workshop on Optimizations for DSP and Embedded Systems* 2010.