# Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs

Gergely Dévai*, Máté Tejfel*, Zoltán Gera*, Gábor Páli*, Gyula Nagy*, Zoltán Horváth*,
Emil Axelsson†, Mary Sheeran†, András Vajda‡, Bo Lyckegård§ and Anders Persson§
*Eötvös Loránd University, Budapest
Email: {deva,matej,gerazo,pgj,n_g_a,hz}@inf.elte.hu
†CSE Dept., Chalmers University of Technology
Email: {emax,ms}@chalmers.se
‡Ericsson Software Research
Email: andras.vajda@ericsson.com
§Ericsson
Email: {bo.lyckegard,anders.c-j.persson}@ericsson.com

***Abstract*—Software for digital signal processors (DSPs) is traditionally highly hardware-dependent and hence porting it to new processors usually requires significant design effort. In this paper we present Feldspar (Functional Embedded Language for DSP and Parallelism), an embedded, high-level, domain-specific language for DSP algorithm design and the compilation techniques we developed for generating C code from specifications written in Feldspar.**

**While Feldspar allows description of algorithms on specification level, we show that with the right set of abstractions and transformations this high level, functional specification can be transformed into C code that is comparable or better than reference, hand-crafted C language implementations. The Feldspar compiler is highly modular and plugin-based, hence future hardware-specific plugins will enable automatic generation of efficient, hardware-specific code. This approach enables the encapsulation of knowledge of hardware completely in the compiler and thus allows description of algorithms in completely hardware-independent, portable manner.**

## I. Introduction

### A. Motivation

In an industrial setting, Digital Signal Processing (DSP) algorithms are implemented as highly optimized C or assembly code on special purpose hardware platforms. The production of this software involves painstaking manual hand-crafting of code, and is expensive. In addition, the resulting code is typically difficult to maintain and, in particular, difficult to port to new platforms.

We propose a new domain-specific programming language called *Feldspar* to tackle this problem. Feldspar expresses DSP algorithms in a clean, abstract and hardware-independent way. Initial experiments show that it is easier to design and implement algorithms on this level of abstraction compared to that of C or assembly.

Feldspar is a high-level domain-specific language. This means that it was designed specifically for DSP algorithms, taking into account the specific constructs of this field. It expresses algorithms in an abstract declarative way instead of using hardware-dependent low level constructs.

The interpreter of Feldspar is a lightweight tool to immediately try out and test the code during the implementation process. However, using this interpreter to run DSP algorithms or even writing them in a general purpose functional programming language (like Haskell [11]) and using its compiler leads to unacceptable performance loss.

The compiler of Feldspar is designed to bridge the gap between an abstract, easy to understand and hardware-independent source program and a highly optimized target code which makes use of the special features of the DSP hardware.

### B. An Embedded Domain-specific Language

Domain-specific languages are designed to capture the common entities and constructs of a given problem domain. Many DSLs has been developed for a variety of domains from digital hardware design [3] to database queries [4]. Recently, work at the Pervasive Parallelism Lab at Stanford has placed domain-specific languages at the core of the search for solutions to the problem of enabling ordinary programmers to program multicore machines. The key element of this approach is the restriction to well delimited, specific domains [1].

DSLs are both extended and restricted compared to general purpose programming languages: They have extra features specific to the given problem domain making the development easier, while their sets of language constructs are limited, which enables more optimized compilation.

Designing new programming languages from scratch is difficult and costly. Embedding [12] is a technology that implements a new language in terms of an existing one, called the *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 language. In case of

deep embedding, the language is defined as a library consisting of functions not performing actual computation but resulting in a data structure. The compiler of the embedded language takes this data structure as input, manipulates it and generates code in the target language.

Currently, Feldspar is an embedded language using Haskell [13] as the host language. It provides two libraries built on top of each other: the *core language* and the *vector library*. The core language includes imperative-like constructs (loops for example) with pure functional semantics. The vector library reimplements some of the standard list functions to be able to transform them to a core language program. During this transformation, the optimization technique called *fusion* is performed (see Section II).

The Feldspar compiler's input is a core language program represented as a graph. This graph is first transformed to an abstract imperative program that is no longer purely functional. Optimization transformations are performed on this representation, which is finally pretty-printed with C syntax.

### C. History of the Feldspar Language

Development of the Feldspar language and compiler was started in 2009 as a joint effort of Eötvös Loránd University (Budapest, Hungary), Chalmers University of Technology (Gothenburg, Sweden) and Ericsson. The first public prototypes of the language and the compiler were released in November, 2009 [9]. Simple DSP algorithms, like autocorrelation with normalization, high- and lowpass filters etc. were successfully implemented in Feldspar. The compiler currently supports ANSI C code generation and the results are comparable with hand-crafted reference implementations.

### D. Paper Structure

This paper is organized as follows. Section II presents the Feldspar language frontend, the fusion technique and the Haskell datastructure representing core language programs. In Section III we describe the compilation of core language programs to abstract imperative code as well as the optimization transformations performed on that representation. Hardware-dependent optimization techniques that we plan to implement are also presented. Evaluation of results and comparison with hand-crafted C code is documented in Section IV; then Section V compares our achievements with related work and a brief summary is given in Section VI.

## II. Language Frontend and High-level Optimizations

The high-level library exported to the user is based around the `Vector` data structure. Vectors can be manipulated using functions similar to Haskell's list operations. They can also be treated as subscripted sequences, allowing programs to be written in a style similar to mathematical formulas.

```
*Main> printCore sumSq
program v0 = v11_1
  where
    v2 = v0 - 1
    v3 = v2 + 1
    v4 = v3 - 1
    (v11_0,v11_1) = while cont body (0,0)
      where
        cont (v1_0,v1_1) = v5
          where
            v5 = v1_0 <= v4
        body (v6_0,v6_1) = (v7,v10)
          where
            v7 = v6_0 + 1
            v8 = v6_0 + 1
            v9 = v8 * v8
            v10 = v6_1 + v9
```

Figure 1. Core program corresponding to `sumSq`

Here follows an example of a "list-like" program for computing the sum of the squares of the numbers between 1 and `n`:

```
sumSq :: Data Int -> Data Int
sumSq n = sum (map square (1 ... n))
  where
    square x = x*x
```

The first line is a type signature stating that `sumSq` accepts an integer argument and gives an integer result. The right-hand side of the definition should be read from right to left. It starts by forming a vector containing the elements from 1 to `n`. Next, each element is squared using `map`, which is a general function for applying an operation – in this case `square` – to each element in a vector. Finally, the elements of the squared vector are summed. Note that `sumSq` is expressed as a *composition* of subfunctions, each one focusing on a single task. This style is typical for functional programs.

While `sumSq` looks like a program that operates on integer data, it is in fact a *program generator* (or macro) that, when run, produces another program that computes the given function. This second program is expressed in Feldspar's *core language*, which servers as the interface to the C code back-end described in this paper. The `printCore` command displays the core program resulting from a given generator. The core program from `sumSq` is shown in Figure 1.

The core language is a pure functional language, but it only consists of low-level constructs that are fairly easy to translate to a machine-oriented language. For example, the program in figure 1 contains a *while-loop* which repeatedly transforms its input using the `body` function until the condition computed by the `cont` function becomes false.

The use of vectors in `sumSq` raises the worry that the program might require excessive memory for the intermediate results. However, looking at the core output, we can

see that this is not the case. All variables (`v0`, `v2`, etc.) are defined by simple scalar expressions. We can also see that all vector operations have been "fused" into a single loop. This efficient core program was obtained by a technique known as *deforestation* or *fusion* [5].

Fusion techniques in functional languages are normally implemented as source code transformations. In Feldspar, we have taken a more direct approach to fusion, taking advantage of the availability of the host language, Haskell, as a macro language. The `Vector` type is defined as follows:

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

This definition says that a vector is essentially a pair (in Haskell) of a length and a function mapping each index to its element. This means that a vector is not a reference to a block of elements in memory; it merely contains the necessary information to compute such a memory block.

The trick here is to avoid computing the vector elements until they are really needed. Many vector operations can be defined by just manipulating the length and/or index function. For example, here is the definition of `map`:

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

Thus, an operation of the form `map f vec` results in a new vector, of the same length as `vec`, but with a different index function. The new index function is simply the composition of `f` and the previous index function `ixf`. Note that `map` does not actually do any work. It just alters the program at the macro level. Many functions from Haskell's standard list library can be defined in a similar way.

The `sum` function can be defined using a *for-loop*:

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

In each iteration (indexed by `i`), the vector element at index `i` is added to the running sum. The for-loop is also a macro, in the sense that it translates into a while-loop in the core language. This is the loop that we see in figure 1.

In the `sumSq` function, the vector `(1...n)` can be defined as

```
(1 ... n) = Indexed n (+1)
```

Substituting `square` and `(1...n)` into the definition of `map`, we get

```
map square (1 ... n) =
  Indexed n (square . (+1))
```

This vector can now be substituted into the definition of `sum` to yield

```
for 0 (n-1) 0 $ \i s -> s + (square . (+1)) i
```

Thus, in the final program, the vectors have completely disappeared, and the index functions have been fused into the body of the for-loop. Note that all of this happens instantaneously while running the generator in Haskell.

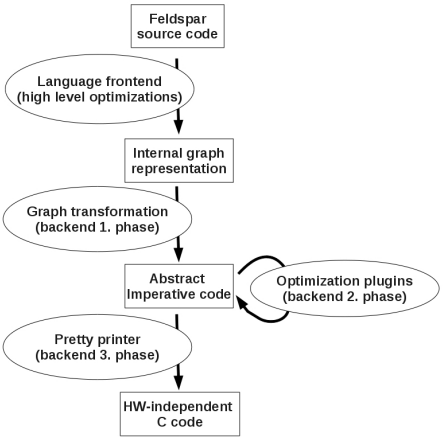


Figure 2. Compiler structure.

Representing vectors by their index functions also has the benefit of allowing a mathematical style of programming. For example, a mathematical definition of `map square a` might be

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

This translates directly to the Feldspar code

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

## III. Compilation and Lower Level Optimizations

### A. Compiler Structure

The previous section describes the first phase of the compilation process, which results in a *core language program* represented by a graph. The target language backend produces C code from this intermediate representation. This transformation consists of several distinct phases (see Figure 2).

The graph is first transformed to *abstract imperative code* described in Section III-B. This transformation is not complicated: local variable declarations can be generated according to the nodes of the graph. The nodes representing primitive operations, branches or loops are transformed to corresponding programming structures in the abstract imperative code. The transformation also takes care of initializing and updating the loop states. This transformation yields an unoptimized program which can already be pretty-printed in C. The result is shown in Figure 3.

The unoptimized program is subject to further optimization steps described in detail in Sections III-C and III-D. These steps are implemented in a highly modular way being plugins in a general architecture. Each plugin consists of an analysis and a transformation routine.

Analysis determines whether the corresponding transformation step is applicable to specific parts of the program, and collects semantic information. Each data structure has

polymorphic type. This enables attaching different semantic information to the nodes of the program in case of different optimization plugins. After having computed the necessary informatiom, the actual modification of the code is performed by the transformation step.

Both analysis and transformation steps need to walk through the program tree and apply some kind of computation at each node of the tree. The plugin architecture offers default functions for both of these activities. This means that only the plugin specific functions have to be imlemented in each plugin. This setup makes the development process faster and code maintenance easier.

Finally, after the optimization steps, a pretty printer is applied producing hardware-independent ANSI C code. Hardware dependent optimization plugins will also be implemented in the near future.

### *B. Representation of Imperative Programs*

Core language Feldspar programs are first transformed to an intermediate representation. This representation encodes an imperative program in an abstract model, which allows easy implementation of optimization transformations, while it can be pretty printed in C in a straightforward way. This representation will be referred to as *abstract imperative code* in the rest of the paper.

The top level datastructure represents a function with its input and output parameters and the program being the body of the function definition. The body is represented by another data structure consisting of local variable declarations and instructions. Instructions are standard imperative constructs like assignment, procedure call, sequence, if-then-else, while and for loops. Figure 4 shows the Haskell representation of the mentioned programming constructs.

The type system of this abstract imperative code is different from that of C. A range of fixed size signed and unsigned integer types with 8, 16, 32 and 64 bits width are used, as well as fixed length arrays, floating point and boolean types. Transformation of these types to C types depends on the given platform.

In C, pointers and pointer operations are used to handle parameter passing and arrays. This makes optimization transformations more difficult to implement, therefore the abstract imperative code abstracts away from the pointer operations. For each parameter and local variable its *logical type* and *role* (input parameter, output parameter or local variable) are stored. The logical type and role define how the parameter or variable will appear in certain positions of the C code. These rules are implemented by the pretty printer that converts the abstract imperative code to C.

### *C. Copy Propagation*

Figure 3 makes it clear that core language Feldspar programs and their translations to abstract imperative code contain many variables to be eliminated. Copy propagation

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

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

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

Figure 3. Unoptimized, genarated C code

is a well-known technique to replace occurrences of a variable with the expression that was assigned to it. This transformation may increase runtime performance, but it can be disadvantageous, if multiple occurrences of a variable get replaced with a large expression. To avoid these cases, we used the following heuristics: a variable is eliminated if either the expression assigned to it is cheap to compute (e.g. a variable or a constant), or there is only one occurrence to replace.

Besides this well-known technique, the Feldspar compiler supports backwards propagation too. The following structure is often present in programs compiled from Feldspar core language:

```
data CompleteProgram
    = CompPrg
    {
        locals  :: [Declaration],
        body    :: Program
    }

data Program =
        Empty
    |   Primitive Instruction
    |   Seq [Program]
    |   IfThenElse
            Variable
              -- condition variable
            CompleteProgram
              -- then part
            CompleteProgram
              -- else part
    |   SeqLoop
            Variable
              -- condition variable
            CompleteProgram
              -- condition calculation
            CompleteProgram
              -- loop body
    |   ParLoop
            Variable
              -- counter
            ImpLangExpr
              -- number of iterations
            Int
              -- step
            CompleteProgram
              -- loop body
```

Figure 4. Haskell representation of programming constructs of the abstract imperative code

```
{
  // ...
  var = expr;
  // ...
}
out = var;
```

After checking that the transformation yields equivalent code, the compiler propagates the `out` variable backwards by replacing occurrences of `var` with it. The result is the following code:

```
{
  // ...
  out = expr;
  // ...
}
```

This technique becomes important when the eliminated operation copies an array.

Applying both propagation transformations to the program presented in Figure 3 results in the much more readable and efficient program in Figure 5.

```
void sumSq(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 5. Result of copy propagation

### D. Hardware-dependent Optimization Techniques

Before converting the program from imperative representation into final form ANSI C code, a number of different hardware-dependent optimizations can be done. These algorithmic transformations are part of the hardware-specific backend of the compiler, so these techniques can vary between different hardware types. However, there are a number of common techniques which can be utilized generally.

Note that whenever programmers port an algorithm to a new platform, the following techniques are done by hand. Feldspar is a tool which is designed to automatically deliver the same optimizations letting the programmer focus only on abstract algorithmic problems.

Partial loop unrolling is a key optimization implemented by the Feldspar compiler. The well known trade-offs between speed and code size have different optima between various hardware flavors. Most C compilers support this optimization step to spare time-consuming jumps at runtime. Feldspar uses this method in a more advanced way to gain extra performance. Unrolling more iterations in one loop body enables the exploit of on-chip parallelization offered by most DSPs, if the number of internal execution units of the processor are taken into account when unrolling a loop.

The transformed code can be improved further by adding the `restrict` keyword to the input parameters of functions. This ensures the compiler that the pointers will not alias each other, something which is automatically guaranteed by the Feldspar compiler.

We have future plans about giving guarantees of minimal, maximal, multiplicative features of iteration numbers reducing the number of condition evaluations. Feldspar could also generate pragmas from this information to enable the

C compiler to produce more efficient code.

The instructions of an unrolled loop may be reordered and grouped together to be replaced by hardware-specific intrinsics. Although these ruin the compatibility with ANSI C standard, intrinsics are really important performance boost factors of hand-optimization of DSP algorithms, so Feldspar is already designed to support such hardware-dependent transformations, and it will definitely implement them in the future.

## IV. Evaluation and Results

### A. Method of Comparison

The fastest DSP programs are always highly optimized to a specific platform which they run on. Optimizations often change the constructs and structure of the actual implementation, that brings the source code away from the original idea. The more optimizations were implemented on the code, the less the programmer will catch the main features of the algorithm. Without knowing these important features, any further improvements or ports to a different platform are risky both taking performance and functionality into account.

Feldspar chooses a different approach. By using a compact and intuitive functional language, the source code serves the programmer's understanding. The Feldspar compiler tries to make all the usual optimizations automatically without drawing the programmer's attention away from abstract algorithmic problems.

To test the performance of code compiled by Feldspar, we will choose commonly used, small but easily measurable DSP algorithms as bases of our comparison. We will take the abstract, high-level Feldspar code of the algorithms and let the Feldspar compiler do all implemented optimizations when creating ANSI C code. On the other side of the comparison, we will choose ANSI C reference implementations which will be identical in function to the Feldspar versions. They are partially optimized hand-written sources lacking hardware-specific optimizations to keep the hardware-independence of examples on both sides. This is necessary because Feldspar is currently fully hardware-independent. Hardware-specific optimization is part of our future work.

The Feldspar versions of the algorithms are far more compact. First sight complexity comparison is impossible because the Feldspar code only contains the core idea without the usual implementation details. We should compare the implementations via compiling both into machine-runnable code and profiling them under normal circumstances. It is important to use a C compiler which is capable of compiling platform-optimized code to simulate behavior also used in the industry. We call the C compiler each time with the same parameters yielding the best optimizations to make sure that ceteris paribus assumption is in place.

### B. Comparison Details

Our chosen algorithms for the test are

- The *convolution* function almost identically taken from the technical specification of 3GPP TS 26.073 V8.0.0 alias the Release 8 of the ANSI C code for the Adaptive Multi-Rate (AMR) speech codec called by using 1000 samples.
- Standard *matrix multiplication* called by using 40x40 matrices respectively.
- Guitar *overdrive* sound effect made on a buffer consisting of 1000 samples.
- Special pitch-shifter transform called *octave up*, done on 1000 samples, derived from PSOLA method without synchronization.

DSP algorithms almost never use dynamic arrays. Our examples use fixed array sizes, and the experiments showed that different buffer sizes yielded the same results.

Our target platform will be TMS320C64xx, a widely used DSP architecture from Texas Instruments (the underlying hardware architecture is discussed in [16]). The compiler and profiler used are included in Code Composer Studio Integrated Development Environment (IDE) Free Evaluation Tools (FETs). In the comparison, this tool permits the study of memory timing, cache size and other low-level features of the target platform, which are all crucial in a real-world simulation.

The measurements are done by the Code Composer Studio default profiler tool. The programs are compiled without debug information on the highest possible optimization level (-O3) with maximum optimization (level 5) for speed. The *whole program optimization* is switched on.

We have used the optimization engine of the Feldspar compiler version expected to be released in the end of February, 2010. Most of these optimizations were already implemented in the first release of Feldspar launched in November, 2009. Up-to-date Feldspar releases and all detailed information about this comparison including source code of test cases can be found at [9].

### C. Results

The results in Table I and in Figure 6 show that the codes generated by the Feldspar compiler in case of convolution and matrix multiplication have similar performance to the reference implementations. This means that Feldspar could successfully take implementation details away from the programmer.

The overdrive effect was optimized far better by Feldspar. This is a huge success and an interesting case where the programmer wanted to create a clean looking C implementation by keeping the formulas together in all branches. The C compiler was unable to find and move the invariant operation parts out of the branch scope. Feldspar has made the unification of invariants easily because the high-level

| | Feldspar | Reference |
|---|---|---|
| Convolution | 1,530,511 | 1,528,516 |
| Matrix Multiply | 114,608 | 114,606 |
| Overdrive | 4,012 | 15,096 |
| Octave Up | 3,019 | 1,528 |

Table I
PERFORMANCE OF FELDSPAR GENERATED CODE AND REFERENCE CODE IN CPU CYCLES

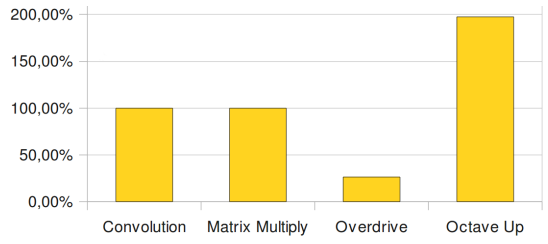


Figure 6. Ratio of Feldspar cycles per reference cycles

language allows thinking in a different way for both the programmer and the compiler.

Octave up is the opposite case. The main difference between the two versions is that they fill the output vector in different styles. Feldspar follows the functional order where the output buffer is written sequentially, while the reference implementation parses the input buffer only once, writing the output buffer in an interleaving fashion. It turns out that the latter case is more cache friendly so the Feldspar generated code gets a huge performance loss. Cache related optimizations are quite hardware-specific, so there are no such optimization constructs currently in Feldspar. However, the plugin architecture makes it possible to implement such optimizations easily.

We have seen a few other examples related to filtering techniques which have some odd dataflow constructs. One example was the lowpass filter in which Feldspar was unable to eliminate some redundant data copies causing similar performance losses like in the previous example. It indicates that there are also hardware-independent optimizations which can be implemented to further improve our code performance.

One of the biggest challenges of our future work is to learn from such examples and implement many optimization methods.

The Feldspar compiler does not yet deploy all the possible optimization transformations and it is currently platform independent, thus hardware-specific optimizations are not yet implemented. We estimate that with the plugin architecture of the compiler, all these optimizations can be automatically generated. However, our compiler already performs well against platform-independent reference implementations. We would like to emphasize that the structure of the resulting code is very much similar to the handwritten code, even though it originates from a very different, compact and high-level language. This is a strong indication that the compiler is able to substitute most of the programmer's optimization efforts, as long as the high level-language has well defined, restricted, domain-specific semantics.

## V. RELATED WORK

The idea of translating high-level models into efficient C programs and using architecture-specific, retargetable backend optimizations is not new. It has been extensively investigated previously in several researches [7], as well as in a dedicated book on optimizing source code for data flow dominated embedded software [8].

Bhattacharyya et al [2] describes the potential pitfalls and problems of ANSI C source code generation for DSP platforms, and they propose a sophisticated architecture for such compilers. However, they implemented all the components of a classical compiler (e.g. a lexical and syntax analysis) but we could safely omit many of them. Their graph-based internal representation can be considered similar to the Feldspar core language, which is also optimized before the code generation phase. Their design supports mapping the internals to concrete machine instructions and the paper addresses all the related problems (question of parallelization, how to generate code that takes memory access costs into account etc.) in details. In our case this is future work.

Regarding the language design, there are other domain-specific languages designed for composing programs for special hardware and implemented in a functional language, like Lava [3] and Obsidian [17] (both developed at Chalmers University of Technology). Another relevant DSL in this topic is SPL [19], the modeling language of SPIRAL [10]. Though the implementation of SPL is not connected to functional programming, it describes digital signal processing computations in a rather elegant way.

It is suggested [6] to augment a "host" programming language (such as Haskell) with a domain-specific library, and embed an optimizing compiler for it. As a consequence, the host language acts as a powerful macro language for writing program in the embedded language. We have followed the same approach in the construction of Feldspar, but the code generation is actually divided into two distinct phases, hence there are a language frontend and a target language backend.

The generation of C programs is implemented in Haskell both in interactive and standalone manner, available as modules and as a binary executable. This compiler structure is similar to Silage's [18], a high-level language dedicated to signal processing. In the Silage compiler, an abstract syntax tree (in the form of a Lisp S-expression) in constructed by the frontend and different backends produce different types of output formats. The frontend is entirely created by using standard tools (Lex and YACC), and the backend is implemented in Common Lisp. Results for the C output are

similar to ours, although the Silage compiler is capable of annotating the variables coming from the source program, and simplifies the generated source better.

Since technically Feldspar is a higher-level programming language that has been created for expressing algorithms in an abstract, hardware-independent manner, one might lose on the efficiency on the other hand. To achieve the desired efficiency, the Feldspar compiler should include more knowledge about the target platforms, more general- and platform-specific optimizations.

Currently, the compiler framework introduced here does not support optimizations specific to compilers of various DSP chips. However, SPIRAL proposes a notable model for code generation for different target platforms [15]. The methodology employed in the model is based on exploiting the domain-specific mathematical structure of algorithms and implementing a feedback-driven optimizer. It offers ideas on how to include new optimization techniques, implementation platforms, and target performance metrics.

A different approach is taken in recent related work on Embedded MATLAB [14], which is a subset of the MATLAB language. MATLAB is a weakly dynamically typed but high-level programming language. Embedded MATLAB supports efficient C code generation for prototyping and deploying embedded systems, and accelerating of fixed-point algorithms. It also generates readable, efficient, and embeddable C code from MATLAB's M-code. Its C code generator is capable of preserving all the function names and comments (in the same position) of the original source code, and all the variable names are the same except that they are prefixed with `eml_`.

## VI. Summary

In this paper, we have presented compilation of Feldspar, a domain-specific language for describing digital signal processing algorithms. This language is embedded into the functional programming language Haskell and we created a compiler for it. At the moment, the compiler is capable of translating from the internal representation of Feldspar programs to ANSI C source code, while supporting several compile-time optimizations.

Our contribution is to design and implement an architecture for the backend that enables using optimization plugins. Some of these optimizations are hardware-independent, and some of them are applicable only on specific target platforms. In our opinion, this might improve both the efficiency and quality of the resulting code, so that the entire process becomes customizable. To our knowledge, no other compiler offers such functionality.

## VII. Acknowledgements

We would like to acknowledge the financial support of Ericsson Software Research, Ericsson Business Unit Networks and SSF (Sweden) as well as the Hungarian National Development Agency (KMOP-2008-1.1.2). We would also like to thank all those who contributed to this paper or to the development of Feldspar with their comments, suggestions and efforts in rewriting applications using Feldspar.

## References

[1] A. Aiken, B. Dally, and co–authors. Towards Pervasive Parallelism, 2009. Slides presenting the Pervasive Parallelism Lab. at Stanford, available at http://ppl.stanford.edu/wiki/images/9/93/PPL.pdf.

[2] Shuvra S. Bhattacharyya, Rainer Leupers, and Peter Marwedel. Software synthesis and code generation for signal processing systems. In *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, pages 849–875, 1999.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 174–184. ACM, 1998.

[4] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[5] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326. ACM, 2007.

[6] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In *Proc. Semantics, Applications and Implementation of Program Generation (SAIG 2000), LNCS*, pages 9–27. Springer-Verlag, 2000.

[7] Heiko Falk and Peter Marwedel. Control flow driven splitting of loop nests at the source code level. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10410, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Heiko Falk and Peter Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[9] Feldspar. (Functional Embedded Language for DSP and PARallelism), a domain specific language with associated code generator, mainly targetting digital signal processing algorithms. http://feldspar.sourceforge.net/.

[10] Daniel McFarlin Franz Franchetti, Frédéric de Mesmay and Markus Püschel. Operator Language: A Program Generation Framework for Fast Kernels. In *Proc. IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385– 410. Springer, 2009.

[11] Haskell. An advanced purely functional programming language. http://haskell.org/.

[12] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55. ACM, 2007.

[14] The MathWorks Inc. Embedded matlab$^{TM}$getting started guide, 2008–2009.

[15] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[16] Nat Seshan. High velociti processing [texas instruments vliw dsp architecture]. *IEEE Signal Process Magazine*, 15(2):86–101, 1998.

[17] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Implementation and Application of Functional Languages, 20th International Symposium, IFL 2008*, 2008.

[18] Edward Wang. A compiler for Silage. Technical report, Computer Science Division, University of California at Berkeley, 1994.

[19] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms, 2001.