Feldspar: A Domain Specific Language for Digital Signal Processing algorithms

Emil Axelsson*, Koen Claessen*, Gergely Dévai†, Zoltán Horváth†, Karin Keijzer*,
Bo Lyckegård‡, Anders Persson‡*, Mary Sheeran*, Josef Svenningsson* and András Vajda§
*Chalmers University of Technology, †Eötvös Loránd University,
‡Ericsson, §Ericsson Software Research
Email for corresponding author: emax@chalmers.se

***Abstract*—A new language, Feldspar, is presented, enabling high-level and platform-independent description of digital signal processing (DSP) algorithms. Feldspar is a pure functional language embedded in Haskell. It offers a high-level dataflow style of programming, as well as a more mathematical style based on vector indices. The key to generating efficient code from such descriptions is a high-level optimization technique called vector fusion. Feldspar is based on a low-level, functional core language which has a relatively small semantic gap to machine-oriented languages like C. The core language serves as the interface to the back-end code generator, which produces C. For very small examples, the generated code performs comparably to hand-written C code when run on a DSP target. While initial results are promising, to achieve good performance on larger examples, issues related to memory access patterns and array copying will have to be addressed.**

I. Introduction

Telecommunications, and especially mobile communications, have seen a rapid development over the last decade, with well over half of the world's population already connected. For telecommunications infrastructure, the consequence is a dramatic increase in bandwidth and computational needs, coupled with an increasing need to deliver new services faster. At the same time, we are witnessing a dramatic shift in the architecture of available computational platforms: the emergence of manycore, heterogeneous chips as well as the diversification of available solutions are adding to the complexity of delivering telecommunications solutions. Signal processing is central to managing content delivery, especially over the air, between mobile base-stations and mobile terminals. Currently most high performance signal processing code is developed in C, often in a very low level form of C with hardware specific intrinsics; essentially, this is assembly language programming in C. Consequently, development is error-prone, has a significant lead-time and hence is costly. For much the same reasons, code portability is limited, even when code is ported between processors from the same vendor. The move to new parallel architectures will exacerbate these problems, as well as calling into question the suitability of C as the high level programming language. The Feldspar project aims to tackle these problems head on, by developing a Domain Specific Language (DSL) for Digital Signal Processing (DSP) algorithm design, with particular application to baseband signal processing. The language aims at raising the abstraction level at which the programmer works with algorithms, reducing development time. The restricted domain permits the use of a rather specialised domain specific language, and it is this restrictiveness that gives us hope of achieving the necessary performance. For a recent snapshot of practical developments in DSLs, see the slides and videos from DSL DevCon 2009 [18]. Our approach is to embed a DSL in Haskell, building upon a wealth of earlier work in the functional programming community on embedded DSLs [14], [7]. In the longer term, we may choose to make a standalone language, but working with an embedded language initially has enabled a more rapid exploration of the design space since we can rely on the considerable infrastructure now available for DSL construction in Haskell. This paper reports the current status of the Feldspar project, in which sequential ISO C99 code is generated from Feldspar descriptions of algorithmic kernels. A major long term goal is to support parallelism, easing the problem of exploiting future manycore architectures. We will, in future, also enable the generation of target-specific C code from high level specifications that are platform independent to the extent possible. Although not yet implemented, these goals have strongly influenced the design of Feldspar and the architecture of its backend compiler.

II. Introduction to Feldspar

Domain experts in DSP tend to explain algorithms using boxes and arrows, and tend to be comfortable with the idea of composing sub-components. Therefore, we have chosen a dataflow style of algorithm description. We expect that this will give us greater ability to capture and exploit potential parallelism.

The domain specific language is called Feldspar and it is, together with its associated compiler, available as open source software [9]. The current version of Feldspar deals only with pure data processing; although, we have initiated work to extend the language to encompass control.

Figure 1 shows how a high-level Feldspar program is transformed into efficient C code. The initial code appears to be a program operating on actual data; however, this is

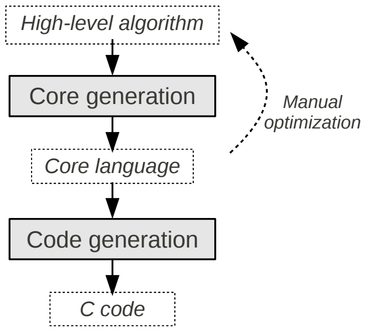

Figure 1. Feldspar compilation flow

just an illusion. The initial program is in fact a *generator* that, when run, results in a data structure representing a program in Feldspar's low-level *core language*. Since Feldspar is embedded in Haskell, the generator is an ordinary Haskell program, and it is the evaluation of this program that we refer to as the *core generation* phase. It is in this generation that many high-level optimizations take place, see section III for more details. The user can look at the generated program (see `printCore` below), and, if necessary, go back and optimize the original code. Next, the code generator produces C for further processing by the DSP chip vendor compilers. The first implementation produces generic ISO C99 and by leveraging the vendor C compiler we can quickly get representative results even for targets with special DSP facilities. For the future, we are working on adding platform descriptions as compiler plugins in order to enable generation of platform-specific C including the use of intrinsics and compiler pragmas.

The rest of the section introduces Feldspar using examples and explains the translations from general Feldspar to the core language. In this paper, we only show code that is accepted by the current Feldspar version (0.3), and all displayed output is the actual output from our tools. In some cases (clearly marked) the generated code has been elided for readability. The examples are included in the file `Memocode2010.hs` included in the 0.3 release[1].

Since Feldspar programs are Haskell programs, we start by introducing a simple example of plain Haskell – a function for computing the sum of the squares of the numbers between 1 and `n`:

```
square :: Int -> Int
square x = x*x

sumSq :: Int -> Int
sumSq n = sum (map square [1..n])
```

This code introduces two functions, `sumSq` and its helper function `square`. A general pattern for introducing functions in Haskell is given by `square`: The definition starts with a type signature (often optional) stating that the function takes an integer argument and returns an integer result. The next line is an equation whose left-hand side is function name and its argument (multiple arguments can be separated by spaces). The right-hand side gives the function body, which may refer to the listed arguments. The definition of `sumSq` follows the same pattern. Its contains two function applications. Inside the parentheses, `map square` is applied to the list $[1, 2, \ldots, n]$ resulting in the list $[1^2, 2^2, \ldots, n^2]$. The *higher-order function* `map` applies its first argument (in this case the function `square`) to each element of its second argument. The resulting squared list is summed by the `sum` function, which gives the final result.

The body of `sumSq` can alternatively be expressed using *function composition*. An equivalent definition is

```
sumSq n = (sum . map square) [1..n]
```

The *compound* function `(sum . map square)` is applied to the list.[2] We tend to use this function composition style whenever possible.

Feldspar descriptions are intended to look as much as possible like Haskell, but the language is severely restricted in order to enable optimization and code generation for the array-processing functions typical of the DSP domain. Here is the Feldspar version of `sumSq`:

```
square :: Data Int -> Data Int
square x = x*x

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

The types use `Data Int` instead of `Int` (in general, `Data a` is the type of Feldspar program computing a value of type `a`). Also, instead of ordinary Haskell lists, this definition uses Feldspar's *symbolic vectors*. The initial vector is now created by the binary operator `...`, and it is processed by the functions `sum` and `map square`, which have been redefined for symbolic vectors. Function composition is unchanged.

The user can *evaluate* a Feldspar program at the prompt of the Glasgow Haskell Compiler's interactive environment (GHCi):

```
*Main> eval (sumSq 10)
385
```

More interestingly, the function `printCore` can be used to show the core language code generated from `sumSq`:

```
*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
```

[1]http://hackage.haskell.org/package/feldspar-language-0.3

[2]The `.` operator is a higher-order function for composing two functions. `(f . g) a` is equivalent to `f (g a)`

```
    body (v6_0,v6_1) = (v7,v10)
      where
        v7 = v6_0 + 1
        v8 = v6_0 + 1
        v9 = v8 * v8
        v10 = v6_1 + v9
```

The code it produces is actually runnable Haskell code (given suitable helper definitions). It contains a number of simple variable definitions and a single use of the `while` function, corresponding to a C-style while loop. The core language is purely functional, which means that there is no hidden state. Instead, the `while` function operates on an explicit state (which is initially `(0,0)` and finally bound to `(v11_0,v11_1)`). (See section III-A for further details.) Looking at the first three assignments, it is clear that some small local optimizations are needed. However, the important point to note is that there is only *one while loop* and not two or three, even though the function was composed by three distinct vector operations (filling the vector, squaring the elements and summing the result). If compiled separately, these three operations would result in one loop each.

The resulting C code is closely related to the core program:

```
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 var6_0;
            signed int var8;
            var6_0 = var11_0;
            var11_0 = (var6_0 + 1);
            var8 = (var6_0 + 1);
            (* out) = ((* out) + (var8 * var8));
        }
    }
}
```

As another example, consider one-dimensional convolution of a kernel vector `a` of length k with an input vector `b`. A mathematical definition is $y[n] = \Sigma_{i=0}^{k-1} a[i] \cdot b[n-i]$. That is, for each n, the output is the dot product of the kernel with the sequence $[b[n], b[n-1], \ldots b[n-(k-1)]]$, which is a (reversed) window into b. To express this windowing, we borrow an idiom directly from Haskell. The function `inits1` returns all the non-empty initial subsequences of an vector. To form the convolution, we must, for each subsequence of the input vector, reverse it and then take the dot product of the kernel with the result. The "for each" results in a `map`, and the mapped function is `scalarProd kernel . reverse`.

```
conv1D :: DVector Float -> DVector Float -> DVector Float
conv1D kernel  = map (scalarProd kernel . reverse) . inits1
```

The resulting core program is of the form

```
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1)) = (v6,v29)
  where
    [code elided for clarity]
```

```
    v29 = parallel v6 ixf
      where
        ixf v7 = v28_1
          where
            [code elided]
            (v28_0,v28_1) = while cont body (0,0.0)
              where
                [code elided]
```

The `parallel` construct (see section III-A) expresses that each element of the output array can be computed independently, using a while loop to perform the dot product. The `reverse` function has been fused into the dot product where it simply alters the loop index so that it traverses the vector in the reversed order. Although the Feldspar backend compiler does not currently take advantage of the opportunity for parallelism offered by `parallel`, it has been essential to design the core language with parallelism in mind.

III. Feldspar Implementation

Feldspar is implemented as an embedded language in Haskell, which means that the language constructs are given as a library of ordinary Haskell functions. These functions do not perform any actual computation (related to the domain). Instead, they simply result in a data structure representing the corresponding core language program. Other functions can then apply different interpretations to the core program; for example, evaluate it or compile it to a low-level language. An implementation based on such an intermediate data structure is usually referred to as a *deep embedding*.

Not all language constructs need to be part of the core language. In fact, one of the great benefits of an embedded language is the ability to use the host language to *generate* programs. This allows us to define complex language constructs as generators that translate into more primitive constructs. For example, the for loop in Feldspar is internally expressed as a while loop. Thus, the user has access to both kinds of loops, but the back-ends only need to support while loops. This is very similar to the way that most high-level languages use syntactic sugar to translate certain syntactic formulations into more primitive ones.

We have taken this idea quite far in Feldspar. While the user has full access to the core language, a typical high-level Feldspar program is mostly expressed using smart generators that even perform high-level optimizations on the fly. An example of this is provided by the kind of vector used in the earlier "sum of squares" example to hold the numbers from 1 to n. The vector `1...n`, which has type `Vector (Data Int)`, appeared in the original Feldspar program, but has completely disappeared in the corresponding core program. The result is that there is no array allocation in the final C code. These *symbolic* vectors will be considered in section III-B; they exemplify the approach of building a DSL around a simple core language plus a series of domain-specific APIs that give the user the feel of working in a much more sophisticated language.

A. Core language

The core language can be seen as a simplified variant of C, except that it has a purely functional semantics and the special parallel array construct introduced in section II. The core language is based around the following interface:

```
value :: Storable a => a -> Data a

ifThenElse :: (Computable a, Computable b) =>
    Data Bool -> (a -> b) -> (a -> b) -> (a -> b)

while :: Computable a =>
    (a -> Data Bool) -> (a -> a) -> (a -> a)

parallel :: Storable a =>
    Data Length -> (Data Int -> Data a) -> Data [a]
```

Core programs have the type `Data a`, where `a` is the type of the program's result. Moreover, core programs are generalized by the `Computable` class of types that can be turned into core programs. First of all, `Data a` is a member of `Computable` (for certain types `a`). In addition, nested tuples are also members of `Computable`. This means that ordinary Haskell tuples, such as `(Data Int, Data Bool)`, can be used with the core interface.

The first construct in the interface, `value`, is used to turn a Haskell value into a core language literal (a program that computes a constant value). For numeric literals, this constructor is inserted automatically, so they can be used directly in Feldspar programs (see, for example, the literal `1` in the earlier `sumSq` example).

The conditional construct, `ifThenElse`, selects between two functions based on a Boolean condition. The reason for operating on functions rather than values is that this allows a lazy semantics: only the selected function needs to be called.

The while loop, `while`, operates on a state of type `a`. The first argument is a function that computes the "continue condition" based on the current state. The second argument is the "body", which computes the next state from the current state. The result is a function from initial state to final state. Note that, unlike loops in imperative languages, this while loop has no side-effects, which is why it uses explicit state passing. Here is a simple example of how to use `while`:

```
modulus :: Data Int -> Data Int -> Data Int
modulus a b = while (>=b) (subtract b) a
```

This function computes the modulus division by repeatedly subtracting `b` from `a` until the result is less than `b`.

The Core language construct `parallel`, is used to express a vector where the computation of any element can be done independently from the others. This is intended to enable a future compiler to generate code exploiting the parallelism using for example SIMD operations. The first argument to `parallel` is the number of elements to compute, and the second is a function mapping each index (starting from `0`) to its value. For example, the first eight powers of 2 can be computed as follows:

```
powersOfTwo :: Data [Int]
powersOfTwo = parallel 8 (\i -> 2^i)

*Main> eval powersOfTwo
[1,2,4,8,16,32,64,128]
```

In addition to the above constructs, the core language also contains quite a large number of primitive functions; for example, arithmetic functions (`(+)`, `(*)`, etc.) and relational operators (`(==)`, `(<)`, etc.).

Similar to Pan [7], we use a data type to represent the abstract syntax of the core language. Each of the given core constructs has a corresponding constructor in the data type. In our case, we use a *generalized algebraic data type* GADT [20], which allows the constructors to be completely type safe with regards to the values flowing in the program graph.

B. Symbolic vectors

An important part of Feldspar is the vector library. This library both supports high level vector operations similar to Haskell's list operations (see, for example, `sumSq`) and indexing which allows for a style close to what is often seen in mathematical specifications (see, for example, `dct2` in section IV).

Vectors in Feldspar are not part of the core language. Instead, the vector library is implemented as a Haskell library and will be translated into core primitives. This is why we refer to the vectors as *symbolic*. The `Vector` type is defined as follows:

```
data Vector a  = Indexed (Data Length) (Data Ix -> a)
type DVector a = Vector (Data a)
indexed l ixf  = Indexed l ixf
```

Symbolic vectors are represented as a pair 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. Since singly-nested vectors are quite common, we often use the alias `DVector a`. In order to somewhat decouple the vector API from its implementation, we define the function `indexed` as an alias for `Indexed`. Symbolic vectors are instances of the `Computable` class, which means that they can be viewed as an *extension* of the core language.

Many vector operations can be defined by just manipulating the length and/or index function. For example, here is the definition of `map` (used in earlier examples):

```
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 introduce any actual array traversal in the final code. It just rearranges the program at the generator level. Many functions from Haskell's standard list library can be defined in a similar way.

In order to avoid recomputation of vector elements that are accessed multiple times, Feldspar provides a special identity function, `memorize`, which writes all elements to memory. This is the only vector operation that allocates any memory.

The `sum` function can be defined using the *for loop* construct in Feldspar:

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

The first two arguments to `for` are the start and stop values for iteration, the third argument is the initial value of a state which is updated at each iteration by the fourth argument, the iteration function. The result value of the `for` construct is the final state produced by the last iteration. In this particular example each iteration (indexed by `i`, the current loop iteration), the vector element at index `i` is added to the running sum. The `for` generator translates into a while loop in the core language. This is the loop that ends up in the C code produced in section II.

Representing vectors as described above has a further benefit: it makes the implementation perform fusion on the vectors, meaning that intermediate vectors will be removed. We will explain fusion via an example by presenting how the function `sumSq` from section II is compiled using our representation of vectors. First of all, the vector `1...n` can be expressed as

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

The first argument to `Indexed` is the total length of the resulting vector. `(+1)` is the index function; it takes an index and sets the value at that position to the index plus one.

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.

C. Backend Implementation

The compiler backend transforms Feldspar core language programs to C [6]. Its input is a Haskell data structure that represents core language programs as dataflow graphs consisting of computation nodes and edges representing the flow of data between the nodes. Compound constructs, like loops and branches are represented by complex nodes containing sub-graphs for the loop body or the code inside the branches of an *ifThenElse* construct.

This dataflow graph has purely functional (side effect free) semantics, but it can be transformed into an imperative program relatively easily. For each node, one or more variables are defined to store the result of the computation described by the node. Passing the input for sub-graphs and writing their output to the right place has to be made explicit by additional data copying instructions. Computation of compound expressions is represented by a series of nodes in the dataflow graph, which translates to a sequence of primitive instructions each of which writes its result to a variable.

As a consequence, the result of the direct transformation of a graph to an imperative program can be optimized by eliminating variables needed only to hold intermediate values and by restructuring the code to reduce the need for data copying. These transformations cannot be performed on the dataflow graph, and it would be hard to implement them on the C level, so an intermediate representation is introduced. We refer to this as the *abstract imperative code*.

The intermediate level uses logical types (like arrays of integers of different sizes) and differentiates between *in* and *out* parameters and local variables. The abstraction level of this representation makes optimization steps easier to implement, yet pretty-printing to C syntax is still straightforward.

The following transformation steps are performed on the abstract imperative code (a more detailed description can be found in [6]):

- *Transformation of primitive instructions.* The instruction set of the Feldspar core language is mapped to that of C in this step. The compiler currently supports ANSI C99 code generation. Soon, it will generate target specific C code based on additional compiler options describing the target platform. Then, the primitive operations will be transformed to calls of optimized library functions or intrinsics of the given platform.
- *Copy propagation* eliminates many intermediate variables by replacing their occurrences by the expression they are bound to.
- *Backward copy propagation* eliminates another set of intermediate variables by propagating the LHS of moves.
- *Loop unrolling* is a classical optimization technique that C compilers can also perform. Our compiler has a different goal to achieve by unrolling a loop: the instructions inside a partially unrolled loop can be combined and replaced by intrinsic operations offered by many DSP platforms. Introducing intrinsics in the code is future work.

The transformation steps described above share common subtasks. Each of them has to walk through the program tree and collect semantic information about the code then perform the actual transformation. By generalizing this common structure, a *plugin architecture* is developed. Transformation steps like loop unrolling or copy propagation are implemented as plugins encapsulating the code that is specific to the given transformation step, while a general architecture takes care of walking through the program tree and passing semantic information.

IV. Further Feldspar Examples

The functional dataflow approach to algorithm specification used in Feldspar lends itself to a style of description similar to that used in the SPIRAL project and its associated DSL (called SPL) [21]. In SPL, transforms and their decompositions are defined as products of structured matrices, using a small number of constructs for building matrices. Then, a transform of input signal x to output signal y by the transform matrix M is expressed as a matrix multiplication $y = Mx$. For example, the Discrete Cosine Transform (type 2) matrix can be expressed as

$$DCT\text{–}2_n = \left[cos \frac{k(2l+1)\pi}{2n} \right]_{0 \le k,l < n}$$

and this transliterates directly into the following Feldspar code, in which `mat` is the $DCT\text{–}2_n$ matrix, which is multiplied by the input `xn` to produce the output vector of the same length.

```
dct2 :: DVector Float -> DVector Float
dct2 xn = mat ** xn
    where
      mat = indexedMat (length xn) (length xn)
              (\k l -> dct2nkl (length xn) k l)

dct2nkl :: Data Int -> Data Int -> Data Int -> Data Float
dct2nkl n k l = cos ( (k' *(2*l' +1)*3.14)/(2*n') )
  where
    (n',k',l') = (intToFloat n,intToFloat k,intToFloat l)
```

The `indexedMat` function is a generalization of the `indexed` function for symbolic vectors. It constructs a matrix from a function that maps row index (`k`) and column index (`l`) to the corresponding element.

The resulting core program shows the result of the vector fusion described in section III-B. The matrix has entirely disappeared.

```
*DCT> printCore (dct2 :: DVector Float -> DVector Float)
program (v0_0,v0_1) = (v0_0,v21)
  where
    v3 = v0_0 - 1
    v14 = intToFloat v0_0
    v15 = 2.0 * v14
    v21 = parallel v0_0 ixf
      where
        ixf v1 = v20_1
          where
            v8 = intToFloat v1
            (v20_0,v20_1) = while cont body (0,0.0)
              where
                cont (v2_0,v2_1) = v4
                  where
                    v4 = v2_0 <= v3
                body (v5_0,v5_1) = (v6,v19)
                  where
                    [code elided for clarity]
                    v16 = v13 / v15
                    v17 = cos v16
                    v18 = v7 * v17
                    v19 = v5_1 + v18
```

Matrices are currently represented as nested vectors. We are working on an improved representation based on a generalization of the `Vector` type.

Typically one would not use this naïve implementation of the DCT but rather use the FFT or a Fast DCT. The recursive descriptions from the Spiral project can also be written in Feldspar. However, the recursion must then be over a Haskell-level value, and must be unrolled at compile time. This produces more code than the naïve implementation, but uses fewer operations to compute the transform. The core language does not offer recursion.

As an alternative to recursion, Feldspar provides some standard combinators for composing programs, such as `map`, `fold` and `scan` [15]. The user is also able to implement new combinators for a specific sub-domain. For example, combinators similar to those used in Lava to describe sorting networks and related structures can be defined [3], although a more index-oriented style of description now seems appropriate. For example, the following is an iterative sorter based on the balanced periodic merger.

```
swapsT n = indexed (n+1) (\j -> (indexed (j+1)
              (\k -> swapcol n (n-j) (j-k))))

swapcol n i v
  = indexed (2^n) (\k -> g (setBitAndShift (i+v) k))
  where
    g k = (k, xor (onesZeros (v+1) i) k)

sort0 n as = fold (fold (fold cswap)) as (swapsT n)
```

The function `swapcol` uses bit level manipulations to calculate the array indices to be compared by each comparator (`cswap`) at a given phase of the sorter. This gives a triply nested loop in the core program, with a single comparator in the inner loop. Feldspar also provides the user with the ability to experiment with moving computation out of the inner loop, or even back to (frontend) compile time.

V. Related Work

A. Compilation methods

Our compilation methods differ from classical compiler technologies because of the embedded nature of the language. There is no need for a lexer, parser and type checker. However, the backend C code generator applies well-known optimization techniques described in [8] among others.

The way our compiler supports optimization by modular design and an optimization framework was described in section III-C. A similar framework is *Hoopl* [22]. Once the optimization is modular, one has to solve problems related to composition of optimization modules. In [16], an interesting solution to this problem is presented. As our plugin phases also perform analyses and transformation at the same time, these ideas may be applicable in our framework.

B. Language implementation

Our embedded language implementation is in the same vein as Pan [7], which has served as a great source of inspiration for Feldspar. Pan is an embedded language for image synthesis and manipulation. Arrays in Pan are created by a function mapping each index to its element, similar to Feldspar's `parallel` construct. However, Feldspar goes further by providing symbolic vectors, which are handled and optimized purely at the generator level.

Obsidian [23] is an embedded language for general-purpose programming of graphics processing units (GPUs). The aim of Obsidian is to enable a functional programming style to make maximal use of the massive parallelism available in GPUs. The programming interface is based on the concept of parallel arrays, which originally gave the inspiration for our symbolic vectors.

Our use of fusion for symbolic vectors is closely related to short cut fusion and related techniques for lists and arrays [13], [19], [5], but there are important differences. Firstly, all symbolic vectors – except those operated on by the `memorize` function – are guaranteed to be removed and will never have a physical representation at runtime. This is in contrast to previous fusion techniques where only some intermediate structures are removed. The guaranteed removal of symbolic vectors combined with programmer control using `memorize` provides a transparent and predictable scheme.

Secondly, the range of functions that can be defined for symbolic vectors is slightly different from what we've learnt to expect from shortcut fusion. The two important things that make symbolic vectors different from lists are that they have a fixed length and that each element in the vector is computed independently. The fixed length cuts both ways when it comes to fusion. Functions like `reverse` can be written and fused without problem when using symbolic vectors, while previous approaches cannot remove the runtime representation [12]. However, a function like `filter`, which removes elements according to a predicate, is a problem for symbolic vectors because the length of the resulting vector is not known until the vector is computed in its entirety. The fact that the elements in symbolic vectors are computed independently also limits the set of functions that can be written over them. Any type of scan where an element depends on previous elements is not possible to write using symbolic vectors alone. In order to cope with scan-like functions, we are experimenting with a different kind of symbolic structure, called `Stream` (see the `iir` example in section VI-A).

Feldspar's concept of a program generator is similar to the concept of *macros* in languages like Scheme. The idea of building advanced language features using macros is not new. For instance, Felleisen et al. show how to use macros to extend Scheme with call-by-reference functions [10].

C. Domain Specific Languages

The notion of a domain specific language for DSP is not new. The Silage project had aims very similar to ours, for instance [11]. We differ in building on a fully fledged functional programming language, with an advanced type system, higher order functions and a continuously developing infrastructure for the embedding of DSLs.

We have been inspired by the Spiral project at CMU [21]. The project uses a DSL to express decompositions of transforms, and then searches for the best decomposition for a given context, permitting the generation of high performance library functions. Even though we do not (yet) use search, the decompositions used in the Spiral project, combined with our fusion technology, enable elegant descriptions of transforms that generate compact C implementations. The DCT example illustrates this, showing how the matrix that characterises the transform is fused away.

The MathWorks is developing *embedded MATLAB*, citing many of the same motivations as those in our project, and aiming to generate C code of sufficiently high performance for use in production [25].

Outside the DSP domain, examples of DSLs with similar aims to ours include Cryptol (a DSL for cryptography from Galois, [17]) and Microsoft's Accelerator for programming various platforms, including GPUs and multicores [24]. In both Cryptol and Accelerator, all loops are unrolled during code generation. Feldspar provides a step up in expressiveness, as it keeps the loops in the generated programs.

One path that we feel the need to explore is the lifting of operations from scalars to matrices and other data structures (as widely used in MATLAB). We have begun experiments with this, making use of type classes to give the required overloading. The language sequenceL [4] pushes this idea to extremes, allowing a function to be automatically lifted to any more deeply nested data type than that for which it is defined. This allows, for instance, the elegant definition in sequenceL of complicated matrix operations such as the Jacobi iteration of a partial differential equation almost directly from the mathematical definition.

VI. Results and Evaluation

For the user of Feldspar, we wish to provide a language which, firstly, makes it easy to write natural high level descriptions of algorithms that are close to their mathematical specification, and secondly generates efficient code which can take full advantage of the target machine. As language implementors we also hope to make an implementation which is both flexible enough to accommodate experiments with the language and allows us to effectively produce good code. Feldspar is still an ongoing project which means that we cannot yet hope to have fully achieved all of these goals. We summarize current results below.

A. Ease of specification

We have seen examples of the kind of high level algorithm specification that can be written in Feldspar: convolution (section II) and discrete cosine transform (section IV). Given

Feldspar's modular structure, it is easy to construct new combinators that can help in writing programs in new domains.

As an example of this we present a cryptographic algorithm, Blake, which is a cryptographic hash function and is a second round candidate for SHA-3 [2]. Cryptographic algorithms have not been our main target when designing Feldspar but our experiments with these algorithms show that it can easily be extended to handle these kinds of algorithms as well. Below is the Feldspar code for two key functions in the Blake specification; the function for computing one iteration over a particular message block `blakeRound`, and its helper function `g`.

```
blakeRound :: MessageBlock -> State -> Round -> State
blakeRound m state r =
 (invDiagonals .
  zipWith (g m r) (4 ... 7) .
  diagonals .
  transpose .
  zipWith (g m r) (0 ... 3) .
  transpose)
  state

g :: MessageBlock -> Round -> Int
  -> DVector Unsigned32 -> DVector Unsigned32
g m r i v = fromList [a'',b'',c'',d'']
  where [a,b,c,d] = toList 4 v
        a' = a + b +
             (m!(sigma!r!(2*i)) ⊕ (co!(sigma!r!(2*i+1))))
        d' = (d ⊕ a') >> 16
        c' = c + d'
        b' = (b ⊕ c') >> 12
        a'' = a' + b' +
              (m!(sigma!r!(2*i+1)) ⊕ (co!(sigma!r!(2*i))))
        d'' = (d' ⊕ a'') >> 8
        c'' = c' + d''
        b'' = (b' ⊕ c'') >> 7
```

The type `Unsigned32` is a Feldspar type for 32 bit unsigned integers.

It is instructive to compare these functions to the specification of Blake [2]. The operations done by `blakeRound` are explained twice in the specification, once using indices and once using pictures. We find that our code has the double advantage of clearly showing what is happening while at the same time being executable. Turning to the function `g`, it is a direct transcription of the specification. It can hardly be any closer to the mathematical description.

This implementation was greatly aided by the existing vector library. However, we had to add the functions `diagonals`, `invDiagonals`, `toList` and `fromList` in order to complete our example, something that was very easily done.

The performance of our Blake implementation is currently rather poor. One major reason is that the code currently copies arrays instead of updating them in place. We are currently working on adding language support for expressing destructive updates.

Many DSP algorithms that we have tried to define in Feldspar have proved to fit easily into either Haskell list-processing style or a more index-oriented approach. But there is also a class of DSP algorithms which involve feedback and which have been difficult to model. Currently, we can express a restricted set of these algorithms using a new higher order function `recurrence`. One such algorithm is an IIR filter, typically specified as $y[n] = \frac{1}{a_0}(\Sigma_{i=0}^{P} b_i x[n-i] - \Sigma_{j=1}^{Q} a_j y[n-j])$. Using `recurrence`, it would be defined as follows:

```
iir :: Data Float -> DVector Float -> DVector Float
    -> Stream Float -> Stream Float
iir a0 a b input =
    recurrence (replicate q 0) input
             (replicate p 0)
      (\x y -> 1 / a0 *
         ( sum (indexed p (\i -> b!i * x!(p-i)))
         - sum (indexed q (\j -> a!j * y!(q-j))))
      )
  where p  = length b
        q  = length a
```

The `recurrence` combinator uses a `Stream` type which encapsulates potentially infinite streaming computations. Our work on streams and feedback algorithms is still in its infancy and is expected to change which is why we refrain from going into further details.

Finally, for some example programs we have found that Feldspar currently doesn't give enough control over memory access patterns. We aim to address this in future versions of Feldspar.

	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 (CPU CYCLES) OF FELDSPAR GENERATED AND REFERENCE CODE ON A DSP TARGET

B. *Performance*

The ISO C99 code produced from high level Feldspar descriptions has, for a number of small examples from the DSP domain, compared favourably with hand-coded reference implementations. Table I is taken from reference [6], a paper which describes the backend compiler and the experiments that produced these results.

The Feldspar version of the Overdrive program performed better than the reference, because the structure of the formulas in the reference implementation made the optimization for the C compiler harder. On the contrary, the Feldspar compiler has unified the invariant parts in the formulas easily, which explains the results. On the other hand, octave up performed worse than the reference, because the Feldspar-generated code fills the output vector sequentially, while the C implementation parses the input vector once, writing the output vector in an interleaving fashion. As a consequence, the generated code involves an extra modulo operation in each iteration.

Much remains to be done (as discussed in section III-C) if the approach is to scale up from the few small examples that have so far been measured. Our work will be guided by the use of standard DSP benchmarks.

C. Implementation techniques

As discussed in section III, Feldspar employs a hybrid approach, having a core language which is deeply embedded, with a number of extensions supplied as shallow embeddings. This provides several benefits. The deep embedding of the core language means that it can not only be interpreted within Haskell, it can also be sent to the compiler backend to produce efficient code. The shallow embedding of the rest of the language makes it very easy to change those parts, such as the implementation of symbolic vectors. This setup works like a workbench for language experimentation which allows for quite drastic changes to the language without having to change the compiler. Having this separation has been extremely valuable during the development of Feldspar, as the language development and compiler implementation happen in different countries.

VII. Discussion

Domain specific high level languages are once again being explored as a means to resolve the tension between the need for higher abstraction level when designing software – in order to reduce cost and development lead-time – and the efficiency of deployed code. Our goal with the Feldspar language is to tackle this tension in the context of DSP algorithms, traditionally a domain that requires almost assembler level coding, in a very target hardware specific manner. We have designed Feldspar as a layered system: at the highest, productivity level, the programmer is offered a highly specialized vector library that allows compact description of most DSP algorithms; efficiency programmers will also program largely with the vector library and similar extensions, but will also use the C-like but functional core language when necessary; finally, the platform-specific backend (when completed) will generate efficient, target-specific C code. (See reference [1] for a discussion of these productivity and efficiency levels, and also of the effects of the move to manycore processors on programming models.)

The goal of the productivity level (the vector library) is to offer a high degree of expressiveness, through tailor-made constructs specific to the vocabulary of domain experts. The vector library is also easily extensible, allowing any programmer to create new constructs. In fact, Feldspar can be extended and changed without impacting the core language or the compiler. We see this as one of the key assets of Feldspar. At the core language level we offer familiar, C-like constructs, but without the usual whistles of C: it has no pointers, it is purely functional and can be run in interpreted mode on any machine. Vector fusion, which can successfully fuse multiple vector operations into one core loop, is an important contribution of Feldspar. The compiler infrastructure has shown promising results on the small set of benchmarks we used [6]. We plan to validate these results using standard benchmark suites. Our preliminary experiments indicate that more hardware-specific optimizations can easily be added to the compiler, using the plug-in infrastructure.

VIII. Future work

Feldspar is a promising approach for attacking the decades old issue of high level DSP code design. In future work, we must continue to develop programming idioms for the DSP domain, to give better coverage of those algorithms that do not fit well in our current programming model. In particular we will continue to build upon our initial results for stream processing, feedback algorithms, programs requiring in-place update and specific memory layouts. To better support the use of built-in accelerators, we will improve the Feldspar foreign function interface, making it possible to call arbitrary library C functions.

We must also develop methods and tools to support testing, debugging and profiling of software written in Feldspar. For the algorithms developed so far, comparison against MATLAB models have turned out to be enough for functional verification, but we have also initiated work on more sophisticated, property-based verification techniques. Debugging is one area that will demand innovations. This is because Feldspar programs are really just code generators, which means that we must find good ways to help the user to understand the relationship between the generated code to the source code.

In the longer term, we plan to evolve Feldspar by adding support for building more complex applications. We will direct research into expressing and optimizing combinations of computational kernels, their communication and orchestration. We will also explore search based exploration as part of a (user-)directed automatic deployment of Feldspar software across multiple processor cores.

IX. Conclusion

We have presented Feldspar, a DSL for DSP algorithm design. The deeply embedded core language allows C-like programming, but in a purely functional setting. Extensions to this core language are provided as shallow embeddings, with the *symbolic vectors* being the main example developed so far. This layered approach allows us to provide the user with an expressive programming language, and to experiment with the design of that interface, while keeping the interface to the backend compiler unchanged. An important aspect of this frontend is the extensive use of fusion, which allows the production of efficient core programs. The high level executable specification language has proved very suitable for specifying DSP algorithms, and the next step is to build larger specifications. The backend compiler operates on the core language (as a graph) and currently produces C code of reasonable efficiency. The backend is built in a modular way, which will facilitate the generation of platform-specific code. Future work also includes the extension of Feldspar to cover control of the data-paths currently described.

X. Acknowledgements

This research is funded by Ericsson, Vetenskapsrådet, the Swedish Foundation for Strategic Research and the Hungarian National Development Agency.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, UC Berkeley, Dec 2006.

[2] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST, 2008.

[3] K. Claessen, M. Sheeran, and S. Singh. Using Lava to design and verify recursive and periodic sorters. *STTT*, 4(3):349–358, 2003.

[4] D. E. Cooke and J. N. Rushton. SequenceL – An Overview of a Simple Language. In *Proc. Int. Conf. on Programming Languages and Compilers (PLC)*, pages 64–70, 2005.

[5] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. 12th ACM SIGPLAN Int. Conf. on Functional Programming*. ACM, 2007.

[6] G. Dévai, M. Tejfel, Z. Gera, G. Páli, G. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård, and A. Persson. Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs. In *Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, workshop associated with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[7] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *J. Func. Prog.*, 13:3:455– 481, 2003.

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

[9] Feldspar. Functional Embedded Language for DSP and PARallelism. http://feldspar.inf.elte.hu/.

[10] M. Felleisen, B. R. Findler, M. Flatt, and S. Krishnamurthi. Building little languages with macros. *Dr. Dobb's Journal*, pages 45–49, April 2004.

[11] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. De Man. DSP specification using the Silage language. In *Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP-90)*, pages 1056– 1060. IEEE, 1990.

[12] A. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[13] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proc. Int. Conf. on Functional programming languages and computer architecture (FPCA)*, pages 223–232. ACM, 1993.

[14] P. Hudak. Modular domain specific languages and tools. In *Proc. Fifth Int. Conf. on Software Reuse*, pages 134–142. IEEE, 1998.

[15] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[16] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37(1):270–282, 2002.

[17] J. R. Lewis and W. B. Martin. Cryptol: High assurance, retargetable crypto development and validation. In *Proceedings of the IEEE/AFCEA Conference on Military Communications (MILCOM)*, pages 820– 825. IEEE, 2003.

[18] Microsoft. DSL Developers Conference: applied topics in domain specific languages. http://msdn.microsoft.com/en-us/data/devcon.aspx, April 2009.

[19] S. L. Peyton Jones, A. Tolmach, and C. A. R. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proc. Haskell Workshop*. ACM SIGPLAN, 2001.

[20] S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. eleventh ACM SIGPLAN int. conf. on Functional programming (ICFP)*, pages 50–61. ACM, 2006.

[21] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. IEEE*, 93(2):232– 275, 2005.

[22] N. Ramsey, J. Dias, and S. L. Peyton Jones. Hoopl: Dataflow optimization made simple, 2009. http://research.microsoft.com/en-us/um/people/simonpj/papers/c--/dfopt.pdf.

[23] J. Svensson, M. Sheeran, and K. Claessen. GPGPU Kernel Implementation and Refinement using Obsidian. In *Proc. Seventh International Workshop on Practical Aspects of High-level Parallel Programming, ICCS*. Procedia, 2010.

[24] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS*, pages 325–335. ACM, 2006.

[25] H. Zarrinkoub. Embedded MATLAB, part 1: From MATLAB to embedded C, 2008. Article on DSP Designline website, http://www.dspdesignline.com/howto/207800773.