The Design and Implementation of Feldspar
an Embedded Language for Digital Signal Processing

Emil Axelsson[1], Koen Claessen[1], Mary Sheeran[1], Josef Svenningsson[1], David Engdal[2], and Anders Persson[1,2]

[1] Chalmers University of Technology
{emax,koen,ms,josefs,anders.persson}@chalmers.se
[2] Ericsson david.engdal@ericsson.com

Abstract. Feldspar is a domain specific language, embedded in Haskell, for programming digital signal processing algorithms. The final aim of a Feldspar program is to generate low level code with good performance. Still, we chose to provide the user with a purely functional DSL. The language is implemented as a minimal, deeply embedded core language, with shallow extensions built upon it. This paper presents full details of the essential parts of the implementation. Our initial conclusion is that this approach works well in our domain, although much work remains.

1 Introduction

The Feldspar project[3] aims to raise the level of abstraction at which Digital Signal Processing (DSP) algorithms are programmed [1]. Today, such algorithms are typically implemented in low level C, which is a poor match for the mathematical notations and concepts used in designing and specifying the algorithms. C is used because performance is critical in applications such as baseband processing in radio base stations. Feldspar is a Domain Specific Language (DSL) embedded in Haskell and generating C. It is designed to raise the level of abstraction at which the programmer works, without sacrificing vital performance.

Feldspar's roots in the DSP domain are reflected in the fact that it is an array programming language. Its design is deliberately minimal, so that it does not contain other DSP-specific features in its core. However, its architecture permits the addition of higher level interfaces built upon the minimal core. The intention to provide a compositional approach to expressing algorithms led to the choice of a purely functional embedded language, and indeed an early design decision was to have Feldspar programs look as much like Haskell as possible. The user works at the GHCi prompt, and the experience is very much like ordinary Haskell programming.

The following example is a Feldspar program that closely resembles the corresponding Haskell function. It computes the bitwise *and* of a mask with each integer in the range `0` to `n`.

[3] The Feldspar project was initiated by Ericsson. Feldspar is available open source [5].

```
mask :: Data Int → Data Int → DVector Int
mask m n = map (m .&.) (0...n)
```

The close resemblance between Feldspar and Haskell programs remains, even in larger examples. However, Feldspar is restricted, to enable the generation of code with reasonable and *predictable* performance. It is the restrictiveness that allows us to find a sweet spot in which modular, reusable high level code still permits the user to control important low level details such as when memory allocation should occur or which loops in the generated code are to be fused. A major restriction is the absence of recursion over C data structures. All operations on arrays must be expressed using higher order functions like `map` and `fold`.

This paper provides full details (including all code) of how to design and implement an embedded language in Haskell that itself resembles Haskell. Combining a minimal core language with an API that gives the user the feeling of writing in a higher level language was fruitful, and the paper documents a (simplified) implementation of an embedded DSL that follows this pattern. A novel combination of implementation techniques is presented, including mixing of deep and shallow language constructs, typed representation of expressions via GADTs [10], and smart constructors that perform optimizations on the fly.

2 Language Architecture

A convenient way to implement a language is to *embed* it within an existing language. The constructs of the embedded language are then represented as functions (or similar) in the host language. In a *shallow* embedding, the language constructs themselves perform the interpretation of the language. In a *deep* embedding, the language constructs produce an intermediate representation of the program. This representation can then be interpreted in different ways.

In general, shallow languages are more modular, allowing new constructs to be added independently of each other. In a deep implementation, each construct has to be represented in the intermediate data structure, making it much harder to extend the language. Embedded languages (both deep and shallow) can usually be interpreted directly in the host language. This is, however, rather inefficient. If performance is an issue, code generation can be employed, and this typically requires a deep embedding.

The design of Feldspar tries to combine the advantages of shallow and deep implementations. We wanted the language to have the modularity and extensibility of a shallow embedding, but we also wanted to use a deep embedding in order to be able to generate high-performance code. A nice combination was achieved by using a deeply embedded core language and building high-level interfaces as shallow extensions on top of the core. The low-level core language is purely functional, but with a small semantic gap to machine oriented languages, such as C. Its intention is to be a suitable interface to the backend code generator, while being flexible enough to support any high-level interfaces.

The architecture of Feldspar is shown in Figure 1. The user interface (the "API" box) exposes the low-level core language as well as some more convenient

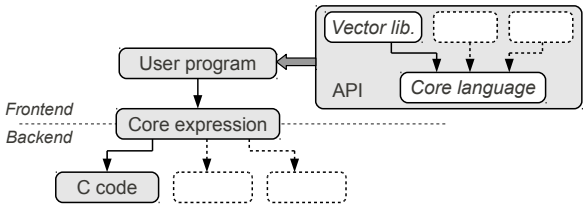

Fig. 1. Feldspar architecture

high-level interfaces. The most prominent of the high level interfaces is the Vector library. But the generality of the core language makes it very easy to implement other interfaces, and this is something we are currently working on. The user's program generates a *core expression*, the internal data structure used as interface to the backends. At the moment, there are two backends: one for producing C code, and one for pretty printing the core expression as Haskell syntax.

3 Core Language

The core language is based around the type constructor Data. For example, a Feldspar program that computes an integer has type Data Int. Simple expressions can be formed using the interface of Haskell's Num class:

```
numExpr = 3*4+5 :: Data Int
```

This expression can be interpreted directly in Haskell:

```
*Main> eval numExpr
17
```

Since the core language is deeply embedded, it is possible to reify the structure of the program, using the function printCore:[4]

```
*Main> printCore numExpr
program = v2
  where
    v1 = 3 * 4
    v2 = v1 + 5
```

In addition to the functions of the Num class, the core language provides its own versions of many basic Haskell functions; for example

```
not :: Data Bool → Data Bool
div :: Data Int → Data Int → Data Int
```

[4] In this example, we have turned off constant folding, which would otherwise have reduced the program to the single value 17.

```
value :: Storable a ⇒ a → Data a

ifThenElse :: (Computable a, Computable b) ⇒
    Data Bool → (a → b) → (a → b) → (a → b)

while :: Computable st ⇒ (st → Data Bool) → (st → st) → (st → st)

parallel :: Storable a ⇒ Data Int → (Data Int → Data a) → Data [a]
```

Listing 1. Basic core language constructs

These functions override the corresponding Prelude definitions, and we will use them without further notice throughout the examples in this paper.

More interesting programs can be built using the constructs in Listing 1. The Computable class generalizes the Data type by allowing various Haskell structures to be treated as programs. This will be further explained in section 4.1; for now, it suffices to know that Data a is a member of Computable (for certain types a).

The construct value turns a Haskell value into a core language literal (a program that computes a constant value). For numeric literals, this constructor is inserted implicitly (by fromInteger). This allowed us to use numeric literals directly in the numExpr 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 lets the user control what expression should go into each branch of the conditional.

The while loop, while, operates on a state of type st. 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 this while loop is a *pure* function with no side-effects. For example, modulus division can be computed by repeated subtraction as follows (assuming a, b > 0):

```
modulus :: Data Int → Data Int → Data Int
modulus a b = while (≥b) (subtract b) a
```

The parallel construct computes an array from a length and a function that maps each index to its element. Arrays are denoted by the type Data [a] (where [a] can be arbitrarily nested). Reusing Haskell's list type for arrays results in compact and readable types. Using parallel, a program that computes the first 10 powers of two is defined as follows:

```
powersOfTwo = parallel 10 (λi → 2^i)
```

```
*Main> eval powersOfTwo
[1,2,4,8,16,32,64,128,256,512]
```

The purpose of parallel is to capture the fact that the array elements are *independent*. This means that they can be computed in any order, or even in parallel. However, our C backend does not yet generate parallel code.

```
data Expr a where
  Value       :: Storable a ⇒ a → Expr a
  Function    :: String → (a → b) → Expr (a → b)
  Application :: Expr (a → b) → Data a → Expr b
  Variable    :: Expr a
  IfThenElse  :: Data Bool → (a :→ b) → (a :→ b) → (Data a → Expr b)
  While       :: (a :→ Bool) → (a :→ a) → (Data a → Expr a)
  Parallel    :: Storable a ⇒ Data Int → (Int :→ a) → Expr [a]
```

Listing 2. Core Language Representation

```
data Data a = Typeable a ⇒ Data (Ref (Expr a))

toData :: Typeable a ⇒ Expr a → Data a
toData = Data ∘ ref

fromData :: Data a → Expr a
fromData (Data a) = deref a
```

Listing 3. Wrapper type for expression nodes

Feldspar's expressiveness comes from the fact that we can use the host language, Haskell, to program powerful abstractions on top of this rather low-level core language. In section 5, we introduce one such language extension: the vector library. This is a substantial extension that more or less eliminates the need for low-level looping constructs in the user's code.

4 Core Language Implementation

Core expressions (see Figure 1) are represented by the generalized algebraic data type (GADT) [10] shown in Listing 2. There is a clear correspondence between the constructs in Listing 1 and those of the Expr type, but also some differences. Expr is mutually recursive through the types Data (see Listing 3) and (:→). The latter type is a representation of functions that supports easy introspection and inlining (see Listing 7).

Each node in an expression tree is wrapped by the Data type, tagging it with a unique reference and enabling observable sharing [3]. References are handled through the interface in Listing 4. In this particular implementation, a reference is just a uniqe tag attached to value (rather than a *mutable* reference). Observable sharing is further discussed in section 4.2.

Another purpose of Data is to make sure that each node in an expression tree stays within the set of types supported by Feldspar. Listing 5 summarizes the two classes that define two sets of supported types (not to be confused with the standard Haskell classes of the same name). Storable is the set of zero- or higher-dimensional arrays of primitive types. These are the types with which the

```
data Ref a
instance Eq (Ref a)
ref   :: a → Ref a
deref :: Ref a → a
```

Listing 4. Interface to observable sharing

```
instance Storable Bool
instance Storable Int
  -- Etc. for other primitive types
instance Storable a ⇒ Storable [a]

instance Storable a ⇒ Typeable a
instance (Typeable a, Typeable b) ⇒ Typeable (a,b)
  -- Similarly for larger tuples
```

Listing 5. Supported core language types

user works; when the user has a value of type Data a, a is generally a Storable type. The Typeable class is the set of nested tuples of Storable types. These types are only used internally. That is, the user is not allowed to work with types like Data (a,b), but should use (Data a, Data b) instead (see section 4.1). The methods of the Storable and Typeable classes are for internal use when generating code.

We can now give a simple definition of the value function from Listing 1, which introduces a value from the Storable set of Haskell values (meta level) into Feldspar (object level).

```
value :: Storable a ⇒ a → Data a
value = toData ∘ Value
```

Primitive functions are constructed by the Function constructor of the Expr type. The String argument identifies the function (for use by backends), and the function argument gives the evaluation semantics. The only way to use a Function node is to apply it using the Application constructor. The other constructors use the Data wrapper for their arguments, and the Typeable constraint for Data rules out function types. The separate application operator enables nested application of curried functions. Listing 6 gives handy combinators for defining primitive functions of one and two arguments. Note the nested use of the application operator (|$|) in function2. Now, defining new primitive functions is trivial:

```
not :: Data Bool → Data Bool
not = function "not" Prelude.not

mod :: Data Int → Data Int → Data Int
mod = function2 "mod" Prelude.mod
```

So far, we can define simple values and primitive functions. The remaining core language constructs deal with embedded functions. It is convenient to be able to treat Haskell functions of the form Data a → Data b as functions in the

```
(|$|) :: Expr (a → b) → Data a → Expr b
f |$| a = Application f a

function :: Typeable b ⇒ String → (a → b) → Data a → Data b
function fun f a = toData $ Function fun f |$| a

function2 :: Typeable c ⇒
             String → (a → b → c) → Data a → Data b → Data c
function2 fun f a b = toData $ Function fun f |$| a |$| b
```

Listing 6. Primitive function constructors

```
data a :→ b = Lambda (Data a → Data b) (Data a) (Data b)

freshVar :: Typeable a ⇒ () → Data a
freshVar _ = toData Variable

lambda :: Typeable a ⇒ (Data a → Data b) → (a :→ b)
lambda f = Lambda f var (f var) where var = freshVar ()

apply :: (a :→ b) → Data a → Data b
apply (Lambda f _ _) = f
```

Listing 7. Representation of embedded functions

embedded language. This view is supported by the :→ type, defined in Listing 7. This type is used to represent the sub-functions of the higher-order constructs of the Expr type (for example, the body of the while loop).

Inspecting a function of type Data a → Data b demands that a suitable argument be conjured up. The function freshVar creates a fresh variable represented by the Variable constructor. Observable sharing makes it possible to uniquely identify each such variable.[5] An embedded function is constructed by lambda, which applies a function to a fresh variable, and stores the original function with the variable and the applied expression. Thus, a term Lambda subst a b can be seen as a lambda expression, where b is an expression in which the bound variable a occurs free. The original function subst gives an effective way of substituting a different expression for the free variable, as done by the apply function.

We now have the building blocks to give simplified definitions of the remaining core constructs. For example, a kind of while loop can be defined as follows:

```
whileData :: Typeable st ⇒
    (Data st → Data Bool) → (Data st → Data st) → (Data st → Data st)
whileData cont body = toData ∘ While (lambda cont) (lambda body)
```

[5] This use of observable sharing is arguably dangerous. We have to be careful to prevent different uses of freshVar from being accidentally shared. We are currently investigating safer techniques for handling variable binding.

```
evalE :: Expr a → a
evalE (Value a)         = a
evalE (Function _ f)    = f
evalE (Application f a) = evalE f (evalD a)

evalE (IfThenElse c t e a) | evalD c   = evalD (apply t a)
                           | otherwise = evalD (apply e a)

evalE (While cont body init) = evalD $ head
    $ dropWhile (evalD ∘ apply cont)
    $ iterate (apply body) init

evalE (Parallel l ixf) = map (evalD ∘ apply ixf ∘ value) [0 .. n−1]
  where n = evalD l

evalD :: Data a → a
evalD = evalE ∘ fromData
```

Listing 8. Semantics of expressions

```
tup2 :: Typeable (a,b) ⇒ Data a → Data b → Data (a,b)
tup2 = function2 "tup2" (,)

get21 :: Typeable a ⇒ Data (a,b) → Data a
get22 :: Typeable b ⇒ Data (a,b) → Data b
get21 = function "get21" fst
get22 = function "get22" snd

  — Similarly for larger tuples: tup3, get31, get32, get33, tup4, etc.
```

Listing 9. Tuple operations

The semantics of core expressions is given in Listing 8. This is generalized in section 4.1.

4.1 Extended Interface

The simple loop whileData has only a single value Data st in its state. We may often require more values in the state, for example to accumulate a sum while increasing an index. One way to use multiple state variables is to make a compound state using the tuple operations in Listing 9. However, it is very inconvenient for the user to have to insert those operations explicitly in the code. Luckily, it turns out that the tupling/untupling can be automated. Listing 10 introduces the Computable class. Generally speaking, this class provides an interface between the Data type and an open set of other, more convenient types. For example, it is much more convenient to work with (Data Int, Data Bool) than Data (Int,Bool), since the former can be constructed and decomposed using Haskell's ordinary tuple syntax. Computable allows us to convert easily between the two types using

```
class Typeable (Internal a) ⇒ Computable a where
  type Internal a
  internalize :: a → Data (Internal a)
  externalize :: Data (Internal a) → a

instance Storable a ⇒ Computable (Data a) where
  type Internal (Data a) = a
  internalize = id
  externalize = id

instance (Computable a, Computable b) ⇒ Computable (a,b) where
  type Internal (a,b) = (Internal a, Internal b)
  internalize (a,b)   = tup2 (internalize a) (internalize b)
  externalize ab      = (externalize (get21 ab), externalize (get22 ab))

  -- Similarly for larger tuples

lowerFun :: (Computable a, Computable b) ⇒
    (a → b) → (Data (Internal a) → Data (Internal b))
lowerFun f = internalize ∘ f ∘ externalize

liftFun :: (Computable a, Computable b) ⇒
    (Data (Internal a) → Data (Internal b)) → (a → b)
liftFun f = externalize ∘ f ∘ internalize
```

Listing 10. Computable class

the internalize / externalize functions. It is possible to automate the insertion of internalize / externalize so that the user will never see a type like Data (Int,Bool). For example, here is the general definition of the while loop, and an example of its use:

```
while :: Computable st ⇒ (st → Data Bool) → (st → st) → (st → st)
while cont body = liftFun (toData ∘ While contL bodyL)
  where contL = lambda (lowerFun cont)
        bodyL = lambda (lowerFun body)

gcd :: Data Int → Data Int → Data Int
gcd a b = fst $ while cont body (a,b)
  where cont (_,b) = b > 0
        body (a,b) = (b,a `mod` b)
```

For the system to remain sound, the functions internalize and externalize are not allowed to change the semantics of the program, as formalized by the rule:

```
evalD ∘ internalize ∘ externalize == evalD
```

Computable is very powerful, as it gives a modular way to extend the language. For example, in section 5, we extend the language with a new type of

```
eval :: Computable a ⇒ a → Internal a
eval = evalD ∘ internalize

ifThenElse :: (Computable a, Computable b) ⇒
    Data Bool → (a → b) → (a → b) → (a → b)
ifThenElse cond t e = liftFun (toData ∘ IfThenElse cond thenSub elseSub)
  where thenSub = lambda (lowerFun t)
        elseSub = lambda (lowerFun e)

parallel :: Storable a ⇒ Data Int → (Data Int → Data a) → Data [a]
parallel l ixf = toData $ Parallel l (lambda ixf)
```

Listing 11. Remaining core language definitions

vectors (seen in the introductory examples). The remainder of the core language implementation is given in Listing 11.

4.2 Inspecting and Optimizing Expressions

Backends, such as `printCore`, work by inspecting the `Expr` data structure produced by core language programs, performing a number of simple but powerful optimizations. To save space, this section can only give a brief summary of the techniques used. The main ideas are described in the work on Pan [4].

At the highest level, the language constructors perform local optimizations on the fly: constant folding, algebraic simplification, etc. Thus, the initial `Expr` data structure is already optimized to a certain extent. In addition, we have experimental support for range-based partial evaluation. A range is an over-approximation of the set of values a variable might take on. This information can be used to fold even non-constant expressions. For example, if the ranges of `a` and `b` are known to be disjoint, the comparison `a==b` can be statically replaced by `value False`. This kind of partial evaluation is also performed "on the fly".

Code duplication can be avoided by the use of the reference equality provided by observable sharing. Essentially, observable sharing allows us to view the `Expr` structure as a directed graph rather than a tree. Once this graph has been generated, a global transformation pass performs hoisting of loop-invariant code. The resulting graph can relatively easily be translated to reasonable C code. While the original motivation behind observable sharing was to reify potentially cyclic graphs, we use it mainly as a means to make an efficient implementation of common sub-expression elimination (CSE). However, since there may exist equal expressions that are not shared, our implementation of CSE is not complete. A complete CSE can be implemented (less efficiently) by using structural equality instead of observable sharing.

4.3 Arrays

Core language arrays are denoted by the type `Data [a]`. Constant arrays are constructed using the `value` function, as in `value [[1,2,3],[4]] :: Data [[Int]]`,

which creates a constant 2×3 matrix with the first row initialized to [1,2,3] and the second to [4]. Arrays are always rectangular, so the above constant has two uninitialized values at the end of the second row.

There are a few more functions that deal with arrays. We have already seen parallel which constructs an array from an index function. In addition, we have the two primitive functions

```
getIx :: Storable a ⇒ Data [a] → Data Int → Data a
setIx :: Storable a ⇒ Data [a] → Data Int → Data a → Data [a]
```

The expression getIx arr i returns the element at index i in the array arr. Similarly, setIx arr i a returns a modified version of arr where the element at index i has been replaced by a.

5 Vector Library

Many algorithms in the DSP domain operate on ordered collections of data, which is why we have added special support for vectors. A vector in Feldspar is much like an array, with one important difference: a vector is guaranteed *not to be represented in memory* at runtime, unless it is explicitly converted into a core-level array. This difference is why we sometimes call vectors *virtual*.

The support for Vectors in Feldspar is implemented as a shallow embedding on top of the core language. Implementing vectors as a shallow embedding has had the benefit of allowing us to experiment with various different vector implementations easily, without having to change any other aspect of the language and its implementation. Furthermore the backend need not be aware of vectors and can therefore be simpler.

The vector library provides a set of functions inspired by standard list processing functions found in Haskell and other functional languages. This allows the programmer to write very high level code that is typically rather close to the mathematical specification of the algorithm. Some example vector functions are shown in Listing 12. Vector is the type of our virtual vectors and its argument is the type of its elements. It is very common that the elements of vectors are of type Data. We often use the abbreviation DVector in those cases.

An example of how to program with the vector library is the function to compute the moving average of a vector, specified as $s_i = \frac{1}{n} \Sigma_{j=i}^{i+n-1} a_j$.

```
movingAvg :: Data Int → DVector Int → DVector Int
movingAvg n = map ((`div` n) ∘ sum ∘ take n) ∘ tails
```

The function tails produces a vector of all the suffixes of the input vector. The use of take in the argument of map creates a window into the original vector of a fixed size. The average of a window is computed using summation and division.

The implementation of the vector type in Feldspar is shown in Listing 12. A vector, which is zero-indexed, is represented by a pair containing the length and an index function, as in the core arrays constructed by parallel. This particular representation has the advantage that all elements in the vector are computed

```
data Vector  a = Indexed { length :: Data Int, index :: Data Int → a }
type DVector a = Vector (Data a)

instance Storable a ⇒ Computable (Vector (Data a))
  where type Internal (Vector (Data a)) = (Int, [Internal (Data a)])

map :: (a → b) → Vector a → Vector b
map f (Indexed l ixf) = Indexed l (f ∘ ixf)

take :: Data Int → Vector a → Vector a
take n (Indexed l ixf) = Indexed (min n l) ixf

drop :: Data Int → Vector a → Vector a
drop n (Indexed l ixf) = Indexed (max 0 (l − n)) (λx → ixf (x + n))

tails :: Vector a → Vector (Vector a)
tails vec = Indexed (length vec + 1) (λn → drop n vec)

(...) :: Data Int → Data Int → Vector (Data Int)
(...) m n = Indexed (n − m + 1) (+ m)

memorize :: Storable a ⇒ Vector (Data a) → Vector (Data a)
memorize (Indexed l ixf) = Indexed l (getIx (parallel l ixf))
```

Listing 12. Implementation of Vector with some smart constructors

independently and can therefore possibly be computed in parallel. Listing 12 shows examples of functions that use this representation. The Computable instances enable vectors to work seamlessly together with the Core language as explained in section 4.1.

The chosen representation also allows for a very lightweight yet powerful implementation of vector fusion. Indeed, fusion comes as a byproduct of the way we have chosen to represent vectors. It is best illustrated by an example. Consider the following toy function:

```
squares :: Data Int → DVector Int
squares n = map square (1...n) where square x = x * x
```

When given an argument m, squares reduces as follows:

```
map square (1...m) ⇒ map square (Indexed m (+1)) ⇒ Indexed m (square ∘ (+1))
```

The vector computation is reduced to a single vector. No intermediate vector is used in the computation. This style of fusion has a significant advantage: vectors are guaranteed to be fused away and take up no memory during runtime. This is a very strong guarantee and by far exceeds the kinds of guarantees a typical optimizing compiler gives. If the programmer wishes to avoid fusing a vector and store the vector in memory, it is a simple matter of inserting a call to the function memorize, the effect of which is to store a vector in memory. This function is useful when elements of a vector are used more than once. If the vector is not written

to memory, then the elements are recomputed each time they are accessed. Such recomputations can in some cases be very costly, in particular when they consist of looping over other vectors or arrays. In such cases, using memorize will often improve the runtime complexity of the function.

In the Computable instance for vectors, the internalize function introduces memory allocation, similarly to memorize. This means that memory is allocated in two situations: (1) explicitly by the memorize function, and (2) implicitly by functions overloaded using Computable when operating on vectors. This scheme provides a simple and easy to remember contract to the programmer which offers both *predictability* and *control*. It is predictable because fusion will always happen, except in the above mentioned situations, and the programmer can control memory allocation and prevent fusion using memorize.

6 Related Work

Embedded Domain Specific Languages are growing in popularity, and are used in many different domains. We cannot survey the entire field, but will restrict our attention to work that has influenced ours, or has aspects in common. Feldspar is compiled, rather than interpreted or used as a library in the host language. An early forerunner is Pan [4] which is similar to Feldspar in many respects. In particular, Pan's treatment of images is similar to Feldspar's vectors, with their associated combinator-based style of programming. However, Feldspar is more general and can handle a larger domain. In implementation, Feldspar differs from Pan in that it uses observable sharing to control intermediate code size, and supports fusion of vectors as an optimization.

Several embedded languages support vector programming in the style of Feldspar. Obsidian [11] is an embedded language for GPU programming that is in many ways similar to Feldspar. Feldspar's vectors were inspired by a similar construct in Obsidian. The main differences arise because Obsidian is specifically targeted to graphics processors. So, for example, loops are unrolled in Obsidian, and the programmer has greater control over the location of data in memory than is currently the case in Feldspar. Repa [6] is a library for array programming in Haskell that shares many similarities with Feldspar's vectors; the two approaches were developed independently. Repa uses the same model of fusion as Feldspar, and also offers programmer controlled memory allocation. It provides greater reusability through a notion of shape polymorphism, which allows functions to work over vectors with different shapes. We intend to adopt this approach in Feldspar.

Other projects aimed at the DSP domain include Spiral [8], Single Assignment C (SAC) [9] and Embedded MATLAB [7]. Spiral automates the production of high performance libraries for DSP applications (among others). To that end, it has a high-level language, SPL, for specifying transforms. SPL has no notion of time or space usage; instead search is used to try to find the best implementation. SAC is a language aimed at efficient array programming and is similar to Feldspar in many respects, including the fact that the array programming

model is implemented modularly as a library. However, SAC inherits from C in the sense that it is a sequential, first-order language; thus, the programming experience is rather different, and in particular less modular, than in Feldspar. Embedded MATLAB is an effort to compile MATLAB to C suitable for running on embedded hardware. Of necessity, Embedded MATLAB is a subset of full MATLAB. Since it is common to develop and prototype DSP algorithms in MATLAB, it makes sense to compile these prototypes directly. The compiler is developed using standard methods, and, as with many optimizing compilers, it can be difficult to predict the results when many optimizations are combined.

Several methods exist to aid the embedding of a language in Haskell. The finally tagless technique [2] provides a very powerful and compositional way of embedding languages. We chose not to use it for two reasons. Firstly, the types become more awkward; the (result-) type of an embedded program is simply a qualified type variable. We find it hard to motivate this for Feldspar beginners. Also, it exposes details of the implementation to the user. Secondly, we have found finally tagless to be incompatible with our use of observable sharing. Since language constructs in finally tagless are overloaded, they are typically implemented as projection functions on dictionaries. When the type is known at compile time, optimization might remove the dictionary. However, this optimization will influence whether the term will be shared under observable sharing or not.

7 Discussion and Future Work

The language presented in this paper has some good sides and some sides that need more work. Our main impression, based on case studies, is that it actually works! Case studies have been performed by Ericsson Baseband Research engineers without prior knowledge of functional programming. They successfully and efficiently implemented a set of signal processing functions and compared with reference implementations in existing C code.

One of the keys to this result was the decision to use a minimalistic low-level core language with a high-level interface implemented as a shallow extension to the core. The minimal core language is quite close to the hardware, making it relatively easy for the backend to produce C code. At the same time, the core is flexible enough to support different kinds of high-level interfaces.

Feldspar aims to offer the Haskell style of programming: pure functions, list-like processing, higher-order functions, etc. It was not obvious that this would be a good fit for the DSP domain. But now, based on the experience of Ericsson research engineers, we can say that pure functional programming appears to be quite well-suited for this task. Admittedly, we need to work on larger examples, and get feedback from more users before we can draw any real conclusions. But the initial results are very promising. We believe that a key to achieving high-level code with good performance is the vector library, which enables powerful code optimization in a predictable and controllable manner.

While our current language shows great potential, we are also aware of some quite serious problems. Our simple core language has been a success. It supports powerful high-level interfaces, such as the vector library, while enabling decent code generation. However, the current core language fails to produce code of sufficiently high performance in commonly occurring cases. As an example, take the append function ++. When compiled to C, it generates the following loop:

```
for(var2 = 0; var2 < (* out_0); var2 += 1)
{
  var8 = (var2 < var0_0_0);
  if   (var8) { out_1[var2] = var0_0_1[var2]; }
  else        { out_1[var2] = var0_1_1[(var2 - var0_0_0)]; }
}
```

In each iteration, a conditional decides whether to pick elements from the first argument (var0_0_1) or the second (var0_1_1). In general, having a conditional inside a loop can prevent the C compiler from doing crucial optimizations. It would be better to have two loops in sequence, each reading from one of the arguments. The problem with our current core language is that *this desired loop structure cannot be expressed.* There are a large number of useful C code patterns that are out of reach from the core language. To deal with this problem, we are working on improving the core language.

Another limitation is the lack of control over memory, due to referential transparency. The user can control whether or not to use memory for vectors (via the memorize function), but it is not possible to control memory layout and memory reuse. A "system layer" being built on top of current Feldspar will handle memory usage and parallelism. In principle, this system layer will act as Feldspar's "IO monad", but the aim is to have a more declarative interface.

The openness of Feldspar makes it easy to add new domain-specific combinators that capture patterns commonly used in DSP. We are working on combinators for describing streaming computations with feedback, for use, for example, in defining digital filters. We are developing a more extensive library for algebraic description of DSP transforms, heavily inspired by the Spiral project [8].

Given that we have chosen to keep Feldspar very close to Haskell one might wonder why we did not program the DSP algorithms directly in Haskell, and spend our efforts improving the compilation of Haskell programs. Haskell programs need an extensive runtime system in order to run, taking up precious space on embedded platforms where resources are scarce. Furthermore, the cost model of Haskell is complex, both for time and space consumption. One of the key design philosophies of Feldspar was to keep these things predictable and under programmer control.

It remains to be seen how well programmers without functional programming background can cope with the embedded nature of Feldspar. The reactions from Ericsson programmers so far have been encouraging. If this turns out to be a real obstacle, one option might be to implement a stand-alone language instead, perhaps with a high-level Haskell front end.

8 Conclusion

Feldspar is implemented as an embedded language in Haskell, and its implementation makes essential use of advanced Haskell features, such as GADTs and overloading. The implementation is based around a simple, low-level, functional core language, which can be fairly easily translated to C code. The power of the implementation comes from the ability to program high-level interfaces as shallow extensions to the core language. We have presented one such extension – the vector library – which enables list-like processing and powerful fusion of vector traversals.

Acknowledgements

This research is funded by Ericsson, Vetenskapsrådet, and the Swedish Foundation for Strategic Research. The Feldspar project is an initiative of and is partially funded by Ericsson Software Research and is a collaboration between Chalmers, Ericsson and ELTE University, Budapest. We wish to thank Peter Brauer of Ericsson for working with us on the case studies.

References

[1] Axelsson, E., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In: Proc. 8^{th} ACM/IEEE International Conference on Formal Methods and Models for Codesign. IEEE (2010)

[2] Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Func. Prog. 19(05) (2009)

[3] Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: ASIAN. LNCS 1742, Springer Verlag (1999)

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

[5] Feldspar: `http://feldspar.inf.elte.hu/feldspar/`

[6] Keller, G., Chakravarty, M., Leshchinskiy, R., Jones, S.P., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in Haskell. In: Proc. 15th ACM SIGPLAN international conference on Functional programming. pp. 261–272. ACM (2010)

[7] Martin, G., Zarrinkoub, H.: From MATLAB to Embedded C, The Mathworks (2009), available at `http://www.mathworks.com/company/newsletters/news_notes/2009/matlab-embedded-c.html`

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

[9] Scholz, S.: Single Assignment C: efficient support for high-level array operations in a functional setting. J. Func. Prog. 13(06), 1005–1059 (2003)

[10] Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proc. 14th ACM SIGPLAN international conference on Functional programming. pp. 341–352. ACM (2009)

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