Feldspar: Application and Implementation

Emil Axelsson and Mary Sheeran

CSE Dept., Chalmers University of Technology
{emax,ms}@chalmers.se

Abstract. The Feldspar project aims to develop a domain specific language for Digital Signal Processing algorithm design. From functional descriptions, imperative code (currently C) is generated. The project partners are Ericsson, Chalmers and ELTE, Budapest. The background and motivation for the project have been documented elsewhere [3]. We aim to raise the level of abstraction at which algorithm developers and implementors work, and to generate, from Feldspar descriptions, the kind of code that is currently written by hand.

These lecture notes first give a brief introduction to Feldspar and the style of programming that it encourages. Next, we document the implementation of Feldspar as a domain specific language (DSL), embedded in Haskell. The implementation is built using a library called Syntactic that was built for this purpose, but also designed to be of use to other implementors of embedded domain specific languages. We show the implementation of Feldspar in sufficient detail to give the reader an understanding of how the use of the Syntactic library enables the modular construction of an embedded DSL. For those readers who would like to apply these techniques to their own DSL embedded in Haskell, further instructions are given in section 5.

The programming examples are available in the `CEFP` directory of the Feldspar package, version 0.5.0.1:

http://hackage.haskell.org/package/feldspar-language-0.5.0.1

The code can be fetched by running:

```
> cabal unpack feldspar-language-0.5.0.1
```

All code is written in Haskell, and has been tested using the Glasgow Haskell Compiler (GHC), version 7.0.2, and the packages

- `syntactic-0.8`
- `feldspar-language-0.5.0.1`
- `feldspar-compiler-0.5.0.1`

1 Programming in Feldspar

Feldspar is domain specific language for DSP algorithm design, embedded in Haskell. It currently generates sequential C code for individual functions and it is this *Data Path* part that is presented here. The part of Feldspar that coordinates and deploys these kernels in parallel is still under development.

The aim of this part of the notes is to give the reader a brief introduction to programming algorithmic blocks in Feldspar. We first present the core language, which is a purely functional C-like language deeply embedded in Haskell. Next, we show how the constructs of the Vector library bring the user closer to a Haskell-like style of programming. The Vector library is built upon the core, via a shallow embedding. The combination of shallow and deep embedding is characteristic of the Feldspar implementation, and has proved fruitful. It is discussed further in section 2.1. Finally, we illustrate the use of Feldspar in exploring a number of implementations of the Fourier Transform. Our aim in designing Feldspar was to build an embedded language that makes programming DSP algorithms as much like ordinary Haskell programming as possible, while still permitting the generation of efficient C code. As we shall see in the sections on implementation, the main emphasis in the design has been on gaining modularity, and on making the language easily extensible. The most noticeable sacrifice has been the omission of recursion in Feldspar. Feldspar users can make use of Haskell's recursion in program definitions, but such recursion must be completely unrolled during code generation.

1.1 The Core of Feldspar

The basis of Feldspar is a core language with some familiar primitive functions on base types, and a small number of language constructs. A program in the core language has type Data a, where a is the type of the value computed by the program. Primitive constructs have types similar to their Haskell counterparts, but with the addition of the Data constructor to the types.

For example, the Haskell functions

```
(==) :: Eq a ⇒ a → a → a
(&&) :: Bool → Bool → Bool
exp :: Floating a ⇒ a → a
```

are matched by the Feldspar functions

```
(==) :: Eq a ⇒ Data a → Data a → Data a
(&&) :: Data Bool → Data Bool → Data Bool
exp :: Floating a ⇒ Data a → Data a
```

The point to remember is that the type Bool, for instance, indicates a Haskell value, while Data Bool indicates a Feldspar one.

Feldspar functions are defined using Haskell's function abstraction:

```
square :: Data WordN → Data WordN
square x = x*x
```

WordN is meant to represent an unsigned integer whose bit-width is determined by the target platform. However, in the current implementation, WordN is implemented as a 32-bit word. We also provide the following two aliases:

```
type Length = WordN
type Index  = WordN
```

The conditional construct in Feldspar is similar to that in C. For instance, the function f below doubles its input if it is odd.

```
f :: Data Int32 → Data Int32
f i = (testBit i 0) ? (2*i, i)
```

Applying the eval function gives

```
*Main> eval (f 3)
6
*Main> eval (f 2)
2
```

The abstract syntax tree of the function can be drawn using drawAST f:

```
*Main> drawAST f
Lambda 0
|
‘- condition
   |
   +- testBit
   |  |
   |  +- var:0
   |  |
   |  ‘- 0
   |
   +- (*)
   |  |
   |  +- var:0
   |  |
   |  ‘- 2
   |
   ‘- var:0
```

and the result is a lambda function of one variable (numbered 0).

The generated C code (resulting from the call icompile f) is

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

/*
 * Memory information
 *
 * Local: none
 * Input: signed 32-bit integer
 * Output: signed 32-bit integer
 *
 */
void test(struct array * mem, int32_t v0, int32_t * out)
{
    if(testBit_fun_int32(v0, 0))
    {
        (* out) = (v0 << 1);
    }
    else
    {
        (* out) = v0;
    }
}
```

The additional mem parameter that appears in all generated C code is not used in the code body in this case. We will return to it in a later example. The remaining two parameters correspond to the input and output of the Feldspar function. (We will not in future show the #includes that appear in all generated C functions.)

Core Arrays Arrays play a central role in the Digital Signal Processing domain, and so they pervade Feldspar. Core arrays come in parallel and sequential variants, but we will concentrate on the parallel version here. Core parallel arrays are created with the parallel function:

```
parallel :: Type a ⇒ Data Length → (Data Index → Data a) → Data [a]
```

The type Data [a] is the type of core arrays. The two parameters to parallel give the length of the array, and a function from indices to values.

```
arr1n :: Data WordN → Data [WordN]
arr1n n = parallel n (λi → (i+1))

*Main> eval (arr1n 6)
[1,2,3,4,5,6]

evens :: Data WordN → Data [WordN]
evens n = parallel n (*2)

*Main> eval (evens 6)
[0,2,4,6,8,10]
```

Feldspar core arrays become blocks of memory in the generated C code. Although the current version of Feldspar generates *sequential* C code for the parallel construct, the key attribute of parallel is that it is a *data parallel* construct, in that the values at different indices are independent of each other. This opens for future exploitation of parallelism, and also for optimisations based on array fusion.

```
append :: Type a ⇒ Data [a] → Data [a] → Data [a]

getLength :: Type a ⇒ Data [a] → Data Length

setLength :: Type a ⇒ Data Length → Data [a] → Data [a]

getIx :: Type a ⇒ Data [a] → Data Index → Data a

setIx :: Type a ⇒ Data [a] → Data Index → Data a → Data [a]
```

Fig. 1. Functions on core arrays

The types of the remaining functions on core arrays are shown in Figure 1 These functions have the expected semantics. For example, the following function squares each element of its input array:

```
squareEach :: Data [WordN] → Data [WordN]
squareEach as = parallel (getLength as) (λi → square (getIx as i))
```

The resulting C code is

```
/*
 * Memory information
 *
 * Local: none
 * Input: unsigned 32-bit integer array
 * Output: unsigned 32-bit integer array
 *
 */
void test(struct array * mem, struct array * v0, struct array * out)
{
    uint32_t len0;

    len0 = getLength(v0);
    for(uint32_t v1 = 0; v1 < len0; v1 += 1)
    {
        at(uint32_t,out,v1) = (at(uint32_t,v0,v1) * at(uint32_t,v0,v1));
    }
    setLength(out, len0);
}
```

The array inputs have been represented by structs of the form

```
struct array
{
   void*     buffer;   /* pointer to the buffer of elements */
   int32_t   length;   /* number of elements in the array */
   int32_t   elemSize; /* size of elements in bytes; (-1) for nested arrays */
   uint32_t  bytes;    /* The number of bytes the buffer can hold */
};
```

and the at macro indexes into the actual buffer.

For completeness, we also introduce the sequential construct:

```
sequential :: (Type a, Syntax s) ⇒
              Data Length → s → (Data Index → s → (Data a,s))
              → Data [a]
```

Sequential arrays are defined by a length, an initial state and a function from index and state to a value (for that index) and a new state. For instance, the following program computes successive factorials:

```
sfac :: Data WordN → Data [WordN]
sfac n = sequential n 1 g
  where
    g ix st = (j,j)
      where j = (ix + 1)  * st

*Main> eval (sfac 6)
[1,2,6,24,120,720]
```

Loops The two important remaining constructs in the core language are the for and while loops[1]:

```
forLoop :: Syntax a ⇒ Data Length → a → (Data Index → a → a) → a

whileLoop :: Syntax a ⇒ a → (a → Data Bool) → (a → a) → a
```

The loop forLoop n i f takes a number of iterations, n, an initial state, i, and a function f from index and state to a new state. Thus, fib n computes the n^{th} Fibonacci number.

```
fib :: Data Index → Data Index
fib n = fst $ forLoop n (1,1) $ λi (a,b) → (b,a+b)
```

This example also illustrates that it is possible, in Feldspar, to have ordinary Haskell tuples both in patterns and in expressions, due to the overloading provided by the Syntax class.

```
void test(struct array * mem, uint32_t v0, uint32_t * out)
{
    struct s_uint32_t_uint32_t_ e0;
    struct s_uint32_t_uint32_t_ v2;

    e0.member1 = 1;
    e0.member2 = 1;
    for(uint32_t v1 = 0; v1 < v0; v1 += 1)
    {
        v2.member1 = e0.member2;
        v2.member2 = (e0.member1 + e0.member2);
        e0 = v2;
    }
    (* out) = e0.member1;
}
```

In the current version of Feldspar, tuples become structs when compiled into C. In programs, such as fib, where tuples are just used to group state variables, it would make more sense to compile them into separate variables. This behavior is planned for future versions.

In similar style, the integer log base 2 function can be computed using a while loop:

```
intLog :: Data WordN → Data WordN
intLog n = fst $ whileLoop (0,n)
                   (λ(_,b) → (b > 1))
                   (λ(a,b) → (a+1, b ‘div‘ 2))
```

[1] There are also monadic versions of these loops, but we will not consider this extension of the language in this introduction

The Feldspar user has access to the constructs of the core language, and this gives fine control over the generated C code when this is required. However, our intention is to raise the level of abstraction at which programmers work, and to do this, we must move away from the low level primitives in the core.

1.2 Above the Core: Vectors

The core constructs of Feldspar are augmented by a number of additional libraries, implemented as shallow embeddings. This eases experiments with language design, without demanding changes to the backends. Here, we illustrate this idea using the library of *Vectors*, which are *symbolic* or *virtual* arrays. Vectors are intended both to give a user experience resembling the look and feel of Haskell list programming *and* to permit the generation of decent imperative array processing code. We call vectors symbolic because they do not necessarily result in the use of allocated memory (arrays) in the generated C code. A program that uses the vector library should import it explicitly using `import Feldspar.Vector`.

Vectors are defined using an ordinary Haskell type:

```
-- Symbolic vector
data Vector a
    = Empty
    | Indexed
        { segmentLength :: Data Length
        , segmentIndex  :: Data Index → a
        , continuation  :: Vector a
        }
```

A vector is defined, for its first *segment*, by a segment length and by a function from indices to values (as we saw in core parallel arrays). However, it also has a *continuation* vector (possibly empty) corresponding to its remaining segments. The overall length of a vector (given by the function `length`) is the sum of the lengths of its segments. Such segmented vectors are used in order to allow efficient vector `append`. Note, however, that taking the sum of a segmented array results in one for loop per segment.

```
tstLApp n = sum (squares n ++ squares (n+2))

void test(struct array * mem, uint32_t v0, uint32_t * out
{
    uint32_t len0;
    uint32_t v2;
    uint32_t v4;

    len0 = (v0 + 2);
    (* out) = 0;
    for(uint32_t v1 = 0; v1 < v0; v1 += 1)
    {
        uint32_t v5;

        v5 = (v1 + 1);
        v2 = ((* out) + (v5 * v5));
        (* out) = v2;
    }
    for(uint32_t v3 = 0; v3 < len0; v3 += 1)
    {
        uint32_t v6;

        v6 = (v3 + 1);
        v4 = ((* out) + (v6 * v6));
        (* out) = v4;
    }
}
```

We will, in the remainder of these notes, only use vectors whose continuation is Empty. Such single segment vectors are built using the indexed function.

For example, $W_n^k = e^{-2\pi ik/n}$ (also known as a twiddle factor) is a primitive n^{th} root of unity raised to the power of k. For a given n, we can place all the powers from zero to (n-1) of W_n into a vector tws as follows:

```
tw :: Data WordN → Data WordN → Data (Complex Float)
tw n k = exp (−2 * pi * iunit * i2n k / i2n n)

tws n = indexed n (tw n)
```

Here, i2n converts from an integer to a floating-point number.

In the following calls to the tws function, the reader is encouraged to examine the results for interesting patterns. How do tws 4 and tws 8 relate and why?

```
*Main> tws 2
[1.0 :+ 0.0, (-1.0) :+ 8.742278e-8]
*Main> eval (tws 4)
[1.0 :+ 0.0, (-4.371139e-8) :+ (-1.0),
(-1.0) :+ 8.742278e-8, 1.1924881e-8 :+ 1.0]
*Main> eval (tws 8)
[1.0 :+ 0.0, 0.70710677 :+ (-0.70710677),
(-4.371139e-8) :+ (-1.0), (-0.70710677) :+ (-0.70710677),
(-1.0) :+ 8.742278e-8,(-0.70710665) :+ 0.7071069,
1.1924881e-8 :+ 1.0, 0.707107 :+ 0.70710653]
```

To make a program that takes an integer as input and returns the corresponding array of twiddle factors, we simply call `icompile tws`. Because the output of the program is a vector, an array will indeed be manifest in memory in the resulting C code.

```
void test(struct array * mem, uint32_t v0, struct array * out)
{
    float complex v2;

    v2 = complex_fun_float((float)(v0), 0.0f);
    for(uint32_t v1 = 0; v1 < v0; v1 += 1)
    {
        at(float complex,out,v1) = cexpf(((0.0f+0.0fi)
          - ((complex_fun_float((float)(v1), 0.0f) *
              (0.0f+6.2831854820251465fi)) / v2)));
    }
    setLength(out, v0);
}
```

But if we (somewhat perversely) sum the vector, then the resulting C code does not have a corresponding array:

```
void test(struct array * mem, uint32_t v0, float complex * out)
{
    float complex v3;
    float complex v2;

    v3 = complex_fun_float((float)(v0), 0.0f);
    (* out) = (0.0f+0.0fi);
    for(uint32_t v1 = 0; v1 < v0; v1 += 1)
    {
        v2 = ((* out) + cexpf(((0.0f+0.0fi) -
              ((complex_fun_float((float)(v1), 0.0f) *
               (0.0f+6.2831854820251465fi)) / v3))));
        (* out) = v2;
    }
}
```

Mapping a function over a vector behaves as we expect:

```
squares :: Data WordN → Vector1 WordN
squares n = map square (1...n)

*Main> eval (squares 4)
[1,4,9,16]

flipBit ::  Data Index → Data Index → Data Index
flipBit i k = i `xor` (bit k)

flips :: Data WordN → Vector1 WordN → Vector1 WordN
flips k = map (λe → flipBit e k)

*Main> eval $ flips 2 (0...15)
[4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11]

*Main> eval $ flips 3 (0...15)
[8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7]
```

The function flips k flips bit number k of each element of a vector.

The type Vector1 a is shorthand for Vector (Data a). The (1... n) construction builds the vector from 1 to n. This could also have been done using the vector function and a Haskell list:

```
*Main> eval (vector [1..3::WordN])
[1,2,3]
```

Indexing into a vector is done using the infix (!) function. So, for example, the head of a vector is its zeroth element.

```
head :: Syntax a ⇒ Vector a → a
head = (!0)
```

The API of the Vector library is much inspired by Haskell's standard list-processing functions, with functions like map, zip, take, drop splitAt and zipWith.

Composing vector operations results in *fusion*: intermediate data structures are fused away in the resulting generated code. One might expect the following function to produce code with two or even three loops, but it has only one:

```
sumSqVn :: Data WordN → Data WordN
sumSqVn n = fold (+) 0 $ map square (1...n)
```

```
void test(struct array * mem, uint32_t v0, uint32_t * out)
{
    uint32_t v2;

    (* out) = 0;
    for(uint32_t v1 = 0; v1 < v0; v1 += 1)
    {
        uint32_t v3;

        v3 = (v1 + 1);
        v2 = ((* out) + (v3 * v3));
        (* out) = v2;
    }
}
```

This code embodies one of the main aims of Feldspar. We want to write code that looks a lot like Haskell, but to generate efficient imperative code, of a quality acceptable within our domain. The key to succeeding in this is to make the language only just expressive enough for our domain! Now is the point to remember that we have no recursion in the embedded language. This pushes us towards a style of functional programming that relies heavily on familiar list functions like map, fold and zipWith, but in variants that work on vectors.

In return for limited expressiveness, the user is given very strong guarantees about fusion of intermediate vectors (and users should, in general, be programming using vectors rather than core arrays). In Feldspar, a vector may become manifest in generated code *only* in the following circumstances

1. when it is explicitly forced using the function force[2]
2. when it is the input or output of a program
3. when it is accessed by a function outside the vector library API, for example, a conditional or a for loop

These are strong guarantees, and they permit us to advocate a purely functional programming style, even when performance is important. When performance and memory use are over-riding concerns, we have the option of resorting to monads and mutable arrays (see [15]). Our hope, which we will try to confirm in an up-coming case study of part of an LTE uplink processing chain, is that some key kernels will have to be finely tuned for performance and memory use, but that the combination of such kernels will still be possible in a modular data-flow style that uses higher order functions to structure programs.

Although we have shown only the Vector library, Feldspar contains a variety of libraries implemented similarly as shallow embeddings. Examples include a clone of the Repa library [14] and libraries for building filters and stream processing functions. Work is also ongoing on dynamic contract checking for Feldspar,

[2] A vector can also be explicitly forced using the function desugar (see section 4.3), but this function is mostly for internal use.

and on improving feedback to users by trying to relate points in the generated code with points in the source (a notoriously difficult problem for embedded languages).

1.3 Case Study: Programming Transforms

Discrete Fourier Transform The discrete Fourier Transform (DFT) can be specified as

$X_k = \Sigma_{j=0}^{n-1} x_j W_n^{jk}$

where W_n^j is an n^{th} root of unity raised to the power of j that we saw earlier, and encoded in the function `tw n j`. Using vectors and summation, it is straightforward to translate the above specification of DFT into Feldspar.

```
dft :: Vector1 (Complex Float) → Vector1 (Complex Float)
dft xs = indexed n (λk → sum (indexed n (λj → xs!j ∗ tw n (j∗k))))
  where
    n = length xs
```

It is also clear that there are n summations, each of n elements, giving the well known O(n^2) complexity of the operation.

Fast Fourier Transforms Any algorithm that gives O(n log n) complexity in computing the same result as the DFT is known as a Fast Fourier Transform or FFT. That one can make such a reduction in complexity is due to the rich algebraic properties of the W_n^k terms – the so-called twiddle factors, and to the sharing of intermediate computations. FFT plays a central role in Digital Signal Processing, where it is one of the most used algorithmic blocks. There are many different FFT algorithms, suited to different circumstances, see reference [10] for an entertaining and informative tutorial.

Radix 2 Decimation in Frequency FFT The best known (and simplest) FFT algorithms are those due to Cooley and Tukey [8]. In the radix two, Decimation in Frequency (DIF) algorithm, for input of length N, the even and odd-numbered parts of the output are each computed by a DFT with $N/2$ inputs. The inputs to those two half-sized DFTs can be computed by N/2 2-input DFTs. This decomposition can be used recursively, giving huge savings in the cost of implementing the algorithm.

We will visualise FFT algorithms by showing how small 2-input, 2-output DFT components are composed, and by indicating where multiplication by twiddle factors happen, see Figure 2. In this style, the structure of the radix 2 DIF FFT is visualised in figure 3.

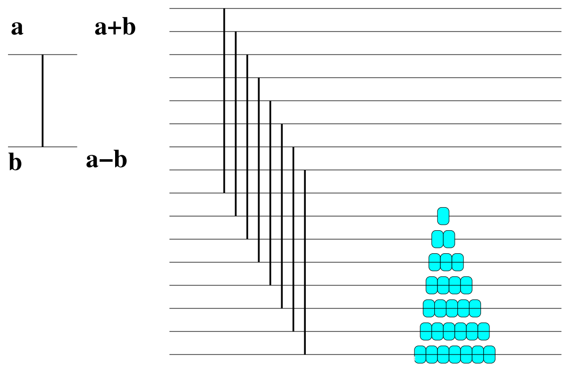

Fig. 2. Introducing the style used to visualise FFT algorithms. The vertical lines are 2-input DFT components, with inputs on the left and outputs on the right. They are drawn linking the elements of the array on which they operate. Thus, the arrangement of vertical lines to the left of the triangle indicates that DFTs are performed between array elements 0 and 8, 1 and 9 and so on. The triangle is intended to suggest multiplication by twiddle factors of increasing powers. A triangle of height n indicates multiplication by W^0_{2n}, W^1_{2n}, and so on, up to W^{n-1}_{2n}. The first of these is indicated by zero blobs, and the last by 7.

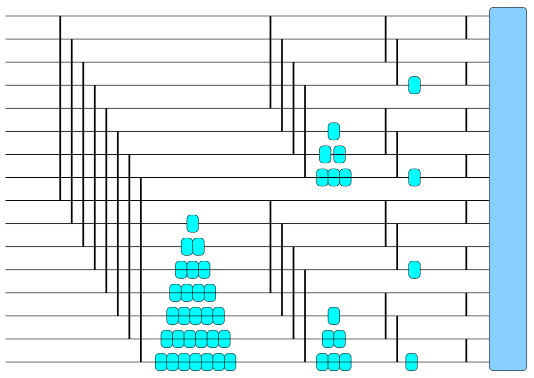

Fig. 3. An illustration of the radix 2 DIF FFT algorithm (for 16 inputs). The box on the right indicates the application of the bit reversal permutation. From left to right, the groups of 2-input DFTs (indicated by the vertical lines) correspond to `bfly 3`, `bfly 2`, `bfly 1` and `bfly 0` in the Feldspar code.

The components of the FFT Let us set about describing the components of the DIF algorithm in Feldspar. Consider first the butterflies, which are made of small (2-input) DFTs. To describe multiple small DFTs, each operating on pairs of values 2^k apart, we might be tempted to first construct a component that works on $2^{(k+1)}$ inputs and then to realise a combinator that allows this component to be applied repeated to sub-parts of an input array. This is how we would have described the construction in Lava (our earlier work on hardware description in Haskell [5]) and indeed the aforementioned paper contains such descriptions of FFT algorithms. Here, we choose a slightly different approach (inspired by our recent work on data-parallel GPU programming in Haskell [7]). We take the repeated application of a function on sub-parts of the input array of a given size to be the default! So, for example, we don't define vector reverse as taking a vector and returning its reverse, but rather `revp k`, which reverses sub-parts of its inputs, each of length 2^k.

```
premap :: (Data Index → Data Index) → Vector a → Vector a
premap f (Indexed l ixf Empty) = indexed l (ixf ∘ f)

revp :: (Bits a) ⇒ Data Index → Vector1 a → Vector1 a
revp k  = premap (‘xor‘ (2^k − 1))

*Main> eval (revp 3 (0...15))
[7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8]
*Main> eval (revp 2 (0...15))
[3,2,1,0,7,6,5,4,11,10,9,8,15,14,13,12]
```

We assume here that if a function like `revp k` is applied to an input then that input must be of length $2^{(k+j)}$, for j a natural number. (We could check this and return an appropriate error message.)

So now we would like to make (repeated) butterflies, each consisting of interleaved 2-input DFTs:

```
bfly :: Data Index → Vector1 (Complex Float)
                   → Vector1 (Complex Float)
bfly k as = indexed (length as) ixf
  where
    ixf i = (testBit i k) ? (b−a, a+b)
      where
        a = as ! i
        b = as ! (flipBit i k)
```

Each individual group of butterflies is of length 2^{k+1} For any index i into the output array, we examine bit k of the index to determine if this output is to be given by an addition or a subtraction. If the bit is high, there should be a subtraction, and `as!i`, which we call `a`, should be subtracted from its partner, `b` which is at a lower index because its index differs from that of `a` only in bit `k`.

Note that the bfly function is a judicious mixture of core functions (including bit manipulations) and vector operations. It is perhaps the explicit indexing into vectors that feels least Haskell-like, but it reflects the mathematics and seems also to bring brevity.

For the multiplication by twiddle factors of increasing power, which takes place only on the second half of the input array, it is again bit k of index i that decides whether or not a multiplication should happen. In calculating the twiddle factor, it is i ‘mod‘ (2^k) that gives the required increasing powers ranging from 0 to $2^k - 1$.

```
twids0 :: Data Index → Vector1 (Complex Float)
                     → Vector1 (Complex Float)
twids0 k as = indexed (length as) ixf
  where
    ixf i = (testBit i k) ? (t∗(as!i),as!i)
      where
        t = tw (2^(k+1)) (i ‘mod‘ (2^k))
```

A first recursive FFT Now we are in a position to compose our first recursive FFT. Remember that the variable that is recursed over must be a Haskell level variable, known at (this first) compile time. For each sub-block of length 2^n, we perform the interleaved butterflies and then the multiplication by twiddles factors. The recursive call that corresponds to two half-size transforms is simply a call of the recursive function with a parameter that is one smaller. We must be careful to convert Haskell values to Feldspar ones (using the value function) where necessary.

```
fftr0 :: Index → Vector1 (Complex Float) → Vector1 (Complex Float)
fftr0 0 = id
fftr0 n = fftr0 n' ◦ twids0 vn' ◦ bfly vn'
      where
        n'  = n − 1
        vn' = value n'
```

This recursive construction demands that the bit-reversal permutation be applied to its output array if it is to produce exactly the same results as the original dft funtion that is now our specification (see [10] for further discussion of this). For blocks of length 2^k, bit reversal should reverse the k least significant bits of the binary representation of each index of the array, leaving all other bits alone.

```
∗Main> eval $ bitRev 4 (0...15)
[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]
∗Main> eval $ bitRev 3 (0...15)
[0,4,2,6,1,5,3,7,8,12,10,14,9,13,11,15]
∗Main> eval $ bitRev 2 (0...15)
[0,2,1,3,4,6,5,7,8,10,9,11,12,14,13,15]
```

For completeness, we give a possible implementation, inspired by the bithacks web site, see http://graphics.stanford.edu/~seander/bithacks.html. However, we also encourage the reader to investigate ways to implement this function. In addition, we note that such permutations of FFT inputs or outputs will sometimes in reality not be performed, but instead the block following the FFT may adjust its access pattern to the data accordingly.

```
oneBitsN :: Data Index → Data Index
oneBitsN  k = complement (shiftLU (complement 0) k)

bitr :: Data Index → Data Index → Data Index
bitr n a = let mask = (oneBitsN n) in
    (complement mask .&. a) .|. rotateLU (reverseBits (mask .&. a)) n

bitRev :: Data Index → Vector a → Vector a
bitRev n = premap (bitr n)
```

Finally, we have a first full FFT implementation:

```
fft0 :: Index →  Vector1 (Complex Float) → Vector1 (Complex Float)
fft0 n = bitRev (value n) ∘ fftr0 n
```

We can compare to the small example simulation of an FFT written in Lava shown in reference [5]. Doing so, we find that we have here made a different (and we think reasonable) assumption about the order of elements of an array, so that some calls of reverse are required if we are to mimic the calculation in the Lava paper.

```
dt4 = zipWith (+.) (vector [1,2,3,1 :: Float]) (vector [4,−2,2,2])

*Main> eval dt4
[1.0 :+ 4.0,2.0 :+ (−2.0),3.0 :+ 2.0,1.0 :+ 2.0]

*Main> eval (reverse (fft0 2 (reverse dt4)))
[1.0 :+ 6.0,(−1.0000007) :+ (−6.0),(−3.0) :+ 2.0000002,7.0 :+ 6.0]
```

Of course, much more extensive testing should be employed, including checking that the composition with an inverse FFT is close enough to the identity. This is beyond the scope of thse notes.

An iterative FFT From the recursive description of fft0, it is not difficult to infer a corresponding iterative description:

```
fft1 :: Data Index → Vector1 (Complex Float) → Vector1 (Complex Float)
fft1 n as = bitRev n $ forLoop n as (λk → twids0 (n−1−k) ∘ bfly (n−1−k))
```

The observant reader may wonder why we didn't just add the multiplication by the twiddle factors directly into the definition of bfly, which would allow us to define the entire FFT as

```
fft2 :: Data Index → Vector1 (Complex Float) → Vector1 (Complex Float)
fft2 n as = bitRev n $ forLoop n as (λk → bfly2 (n−1−k))
  where
    bfly2 k as = indexed (length as) ixf
      where
        ixf i = (testBit i k) ? (t∗(b−a), a+b)
          where
            a = as ! i
            b = as ! (flipBit i k)
            t = tw (2ˆ(k+1)) (i ‘mod‘ (2ˆk))
```

This is indeed quite a short and readable FFT definition. However, this kind of manual merging of components is not always desirable. There are two main reasons for this. The first is that it can be easier to replace individual components with modified versions if the components are kept separate and can be modified in isolation. (This kind of modularity is a typical benefit of working in a purely functional language.) The second is that keeping components separate allows easier experiments with new ways of combining them (and such experiments are particularly relevant in the context of FFT, which is known to have many interesting decompositions).

Playing with twiddle factors We would like, eventually, to avoid unnecessary recomputation of twiddle factors. This takes two steps. First, we modify the code so that all stages compute twiddle factors that have the same subscript. Next, we force computation of an array of these twiddle factors, which later parts of the program can access, avoiding recomputation.

Let us consider the component twids0 in isolation (and later we will look at new ways of combining the resulting components). One of the important algebraic properties of the twiddle factors is the following: $W_n^k = W_{2n}^{2k}$. (You may have had an inkling of this when you examined the values of tws 2, tws 4 and tws 8 earlier.) This fact gives us the opportunity to change the twids0 program so that all twiddle factors used in an entire FFT have the same subscript (rather than having different subscripts for each different parameter k in different stages of the computation).

Defining twids1 as follows means that twids1 j k has the same behaviour as twids0 k, as long as j is strictly greater than k.

```
twids1 :: Data Index → Data Index → Vector1 (Complex Float)
                                   → Vector1 (Complex Float)
twids1 n k as = indexed (length as) ixf
  where
    ixf i = (testBit i k) ? (t * (as!i), as!i)
      where
        t = tw (2^n) ((i `mod` (2^k)) .<<. (n−1−k) )
```

This is because we have multiplied both parameters of tw by 2^{n-1-k} (the first by relacing 2^{k+1} by 2^n and the second by shifting left by 2^{n-1-k} bits).

Forcing computation of twiddle factors Now, all stages of the 2^n-input FFT use tw (2^n) when calculating twiddle factors. We can compute the 2^{n-1} twiddle factors needed *before* starting the FFT calculation, using the force function to ensure that they get stored into an array ts. Then the call of tw (2^n) is simply replaced by ts !. This approach avoids repeated computation, which can be the downside of fusion.

```
twids2 :: Data Index → Data Index → Vector1 (Complex Float)
                                   → Vector1 (Complex Float)
twids2 n k as = indexed (length as) ixf
  where
    ts = force $ indexed (2^(n−1)) (tw (2^n))
    ixf i = (testBit i k) ? (t * (as!i), as!i)
      where
        t = ts ! ((i `mod` (2^k)) .<<. (n−1−k))
```

The resulting FFT is then

```
fft3 :: Data Index → Vector1 (Complex Float) → Vector1 (Complex Float)
fft3 n as = bitRev n $ forLoop n as (λk → twids2 n (n−1−k) ∘ bfly (n−1−k))
```

and it gives C code that starts as follows:

```
void test(struct array * mem, uint32_t v0, struct array * v1, struct array *
out)
{
    uint32_t v13;
    float complex v14;
    uint32_t len0;
    uint32_t v24;
    uint32_t v25;
    uint32_t len2;

    v13 = (v0 - 1);
    v14 = complex_fun_float((float)((1 << v0)), 0.0f);
    len0 = (1 << v13);
    for(uint32_t v6 = 0; v6 < len0; v6 += 1)
    {
     at(float complex,&at(struct array,mem,0),v6) =
      cexpf(((0.0f+0.0fi) - ((complex_fun_float((float)(v6), 0.0f) *
      (0.0f+6.2831854820251465fi)) / v14)));
    }
    setLength(&at(struct array,mem,0), len0);
```

Note how one of the C arrays in the mem parameter is used to store the twiddle factors, for use by the remainder of the program. This is the role of that parameter: to provide storage for local memory in the function. Our generated functions do not themselves perform memory allocation for the storage of arrays. The necessary memory must be given to them as the first input. For the twiddlle factor array, another option would be simply to pass it as an input to the FFT function.

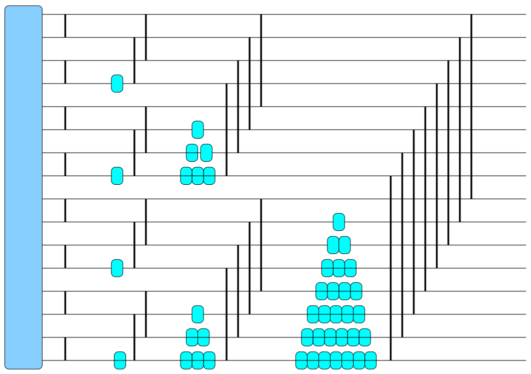

Fig. 4. An illustration of radix 2 DIT FFT algorithm (for 16 inputs). This diagram was (literally) produced from that for the DIF algorithm by flipping it vertically.

Radix 2 Decimation in Time FFT The final FFT that we will program in Feldspar is the Decimation in Time (DIT) radix two variant of the algorithm. One can think of it as being almost the result of running the data-flow graph that we just built for the DIF algorithm *backwards*. That is, we start with the bit reversal, then `twid2 n 0`, then `bfly 0`, then `twids2 n 1` and so on. Note that we do twiddle multiplications *before* butterflies in this case.

```
fft4 :: Data Index → Vector1 (Complex Float) → Vector1 (Complex Float)
fft4 n as = forLoop n (bitRev n as) (λk → bfly k ∘ twids2 n k)
```

The resulting C code is reproduced in the Appendix. It is reasonably satisfactory, but contains one annoying array copy inside the outer loop of the main FFT calculation. This copying could be avoided by using ping-ponging between two arrays, perhaps using a specially designed for loop. This would be easy to arrange, and is the approach used in the Obsidian embedded language for GPU programming [7] (although there all loops are unrolled). To get completely satisfactory performance, we would need to make an in place implementation using monads. The structure of the `bflys` component is well prepared for this, since each 2-input DFT has its inputs and outputs at the same indices.

The duality between the decimation in frequency (DIF) and in time (DIT) variants can be seen by examining the definitions of `fft2` and `fft4` and by studying the diagrams illustrating these constructions (Figures 3 and 4).

Many FFT algorithms remain to be explored. Readers wishing to experiment with Feldspar will find a wealth of interesting algorithms to program in the FFT survey in reference [10]. We should be clear that DSP algorithm designers most likely expect to be provided with fast FFT components, rather than to have to write them. However, FFT algorithms can help us to develop useful programming idioms. The development of new programming idioms is part of our current research on Feldspar. We welcome input (and code snippets) from the readers of this document.

This concludes your introduction to programming in Feldspar.

Exercise 1. Implement Batcher's bitonic sort in Feldspar [4]. See http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm. Note that the Radix 2 DIF FFT (as shown in Figure 3) has recursive structure similar to Batcher's bitonic merger. If you ignore the blobs in that diagram, and consider the vertical lines to be 2-input, 2-output comparators, you have exactly the bitonic merger. So you may find some inspiration in the `bfly` and `fft1` functions.

2 Implementation

The development of Feldspar has not only focused on the problem of making a language for the embedded signal processing domain. Feldspar has also served as a workbench for experimenting with different implementation techniques for embedded languages in general. This has been partly motivated by the fact that there are several partners involved in the project, and we need a very flexible design in order to make collaboration easier. In this part we will look at the general implementation techniques that have emerged out of Feldspar, and show an implementation of Feldspar based on the general techniques.

2.1 Overview

A convenient way to implement a domain-specific language (DSL) is to *embed* it within an existing language [13]. Often, the constructs of the embedded language are then represented as functions in the host language. In a *shallow* embedding, the language constructs themselves perform the interpretation of the language [12]. 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 done using a deep embedding [11].

The design of Feldspar tries to combine the advantages of shallow and deep implementations. The goal is to have the modularity and extensibility of a shallow embedding, while retaining the advantages of 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 code generator, while being flexible enough to support any high-level interfaces.

The architecture of Feldspar's implementation is shown in figure 5. The deeply embedded core language consists of an intermediate representation (the "Core expression" box) and a user interface ("Core language"). Additionally, the user interface consists of a number of high-level libraries with shallow implementation (their meaning is expressed in terms of the core language constructs). The most prominent high level library is the vector library (section 4.3). There are also some more experimental libraries for synchronous streams, bit vectors, etc. The

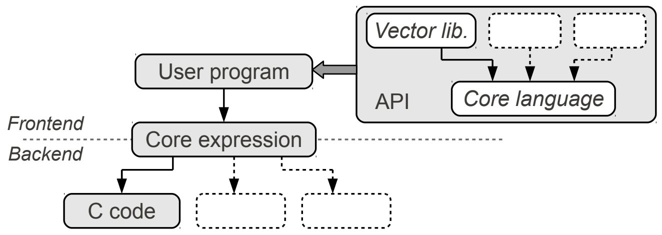

Fig. 5. Feldspar architecture

user's program generates a *core expression*, the internal data structure used as interface to the back ends. At the moment, there is only one back end – a code generator producing C code.

This architecture gives us certain kinds of modularity, as indicated in figure 5: high-level interfaces and back ends can be added independently of everything else. However, the core language expression type has, so far, been hard-coded in the implementation. This has made the implementation quite inflexible when it comes to changing the core language.

2.2 Early Implementations

```
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]

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

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

Fig. 6. Previous core language representation

The implementation style of the initial Feldspar versions is described in reference [2]. There, the core language expressions are defined using the data type in figure 6. Ignoring some details, this is a standard abstract syntax tree, where each constructor corresponds to a specific language construct. It is worth noting the (:↠) type, which captures the notion of variable binding. For example, the second argument of Parallel is a representation of a function $\lambda i \rightarrow body$, where $body$ is an expression of the element at index i.

Even though the definition in figure 6 is quite simple, it lacks the desired modularity. We do have the ability to extend the library with new high-level types by providing a translation to Data a:

```
frontEnd₁  :: MyType₁ a → Data a
frontEnd₂  :: MyType₂ a → Data a
...
```

(A more general translation mechanism is provided by the Syntactic class described in section 3.3.) We can also add any number of back ends:

```
backEnd₁ :: Data a → Back₁
backEnd₂ :: Data a → Back₂
...
```

But adding a constructor to Expr/Data requires editing the module containing their definition as well as the modules of all back ends to handle the new constructor.

Most of the constructors in the Expr type are general language constructs that are likely to be useful in other languages than Feldspar. This is especially true for variable binding, which is a tricky concept that gets reimplemented over and over again in various embedded languages. If we managed to make the language definition more modular, it should also be possible to put the most basic constructs in a library so that they can be reused in many different language implementations.

We have developed a library, Syntactic [1], that provides the extensibility and reuse described above. Section 3 introduces the Syntactic library, and section 4 gives an overview of how Feldspar is implemented using Syntactic.

3 Syntactic Library

When implementing deeply embedded DSLs in Haskell, a syntax tree is typically defined using an algebraic data type [11,2]. As an example, consider a small expression language with support for literals and addition:

```
data Expr₁ a
  where
    Lit₁  :: Num a ⇒ a → Expr₁ a
    Add₁ :: Num a ⇒ Expr₁ a → Expr₁ a → Expr₁ a
```

Expr₁ a is a generalized algebraic data type (GADT) [16] whose parameter a is used to denote the type of the value computed by the expression. It is easy to add a user friendly interface to this language by adding smart constructors and interpretation functions.

```
lit₁ :: Int → Expr₁ Int
lit₁ x = Lit₁ x

add₁ :: Expr₁ Int → Expr₁ Int → Expr₁ Int
add₁ x y = Add₁ x y

eval₁ :: Expr₁ Int → Int
eval₁ (Lit₁ x)   = x
eval₁ (Add₁ x y) = eval₁ x + eval₁ y
```

(In this case, the smart constructors only serve to hide implementation details and constraining the type, but in later implementations they will also take care of some tedious wrapping.)

The eval_1 function is just one possible interpretation of the expressions; we can easily extend the implementation with, say, pretty printing or any kind of program analysis. This can be done even without changing any existing code. However, adding a new construct to the language is not so easy. If we would like to extend the language with, say, multiplication, we would need to add a constructor to the Expr_1 type as well as adding a new case to eval_1 (and other interpretations). Thus, with respect to language extension, a simple GADT representation of a language is not modular. This limitation is one side of the well-known *expression problem* [18].

There are several reasons why modularity is a desired property of a language implementation. During the development phase, it makes it *easier to experiment* with new language constructs. It also allows constructs to be developed and tested independently, *simplifying collaboration*. However, there is no reason to limit the modularity to a single language implementation. For example, Lit_1 and Add_1 are conceptually generic constructs that might be useful in many different languages. In an ideal world, language implementations should be assembled from a library of generic building blocks in such a way that only the truly domain-specific constructs need to be implemented for each new language.

The purpose of the Syntactic library is to provide a basis for such modular languages. The library provides assistance for all aspects of an embedded DSL implementation:

– A generic AST representation that can be customized to form different languages.

– A set of generic constructs that can be used to build custom languages.

– A set of generic functions for interpretation and transformation.

– Generic functions and type classes for defining the user interface of the DSL.

3.1 Using Syntactic

```
data AST dom a
  where
    Sym  :: Signature a ⇒ dom a → AST dom a
    (:$) :: Typeable a  ⇒ AST dom (a :→ b) → AST dom (Full a)
                                          → AST dom b

type ASTF dom a = AST dom (Full a)

infixl 1 :$
```

Fig. 7. Type of generic abstract syntax trees

```
newtype Full a   = Full { result :: a }
newtype a :→ b = Partial (a → b)

infixr :→

class    Signature a
instance Signature (Full a)
instance Signature b ⇒ Signature (a :→ b)
```

Fig. 8. Types of symbol signatures

The idea of the Syntactic library is to express all syntax trees as instances of a very general type AST[3], defined in Figure 7. Sym introduces a symbol from the domain dom, and (:$) applies such a constructor to one argument. By instantiating the dom parameter with different types, it is possible to use AST to model a wide range of algebraic data types. Even GADTs can be modeled.

To model our previous expression language using AST, we rewrite it as follows:

[3] The Typeable constraint on the (:$) constructor is from the standard Haskell module Data.Typeable, which, among other things, provides a type-safe cast operation. Syntactic uses type casting to perform certain syntactic transformations whose type-correctness cannot be verified by the type system. The Typeable constraint on (:$) leaks out to functions that construct abstract syntax, which explains the occurrences of Typeable constraints throughout this paper. It is possible to get rid of the constraint, at the cost of making certain AST functions more complicated.

```
data NumDomain₂ a
  where
    Lit₂  :: Num a ⇒ a → NumDomain₂ (Full a)
    Add₂ :: Num a ⇒ NumDomain₂ (a :→ a :→ Full a)

type Expr₂ a = ASTF NumDomain₂ a
```

The result type signatures of Lit₂ and Add₂ have a close correspondence to the Lit₁ and Add₁ constructors. In general, a constructor of type

```
C₂ :: T₂ (a :→ b :→ ... :→ Full x)
```

represents an ordinary GADT constructor of type

```
C₁ :: T₁ a → T₁ b → ... → T₁ x
```

Types built using (:→) and Full are called *symbol signatures*, and they are defined in Figure 8.

In this encoding, the types Expr₁ and Expr₂ are completely isomorphic (up to strictness properties). The correspondence can be seen by reimplementing our smart constructors for the Expr₂ language:

```
lit₂ :: Int → Expr₂ Int
lit₂ a = Sym (Lit₂ a)

add₂ :: Expr₂ Int → Expr₂ Int → Expr₂ Int
add₂ x y = Sym Add₂ :$ x :$ y
```

The implementation of eval₂ is left as an exercise to the reader. Note that, in contrast to Add₁, the Add₂ constructor is *non-recursive*. Types based on AST normally rely on (:$) to handle all recursion.

Part of the reason for using the somewhat unnatural AST type instead of an ordinary GADT is that it directly supports definition of generic tree traversals. Generic programming using AST is not the subject of these notes, but the basic idea can be seen from a simple function returning the number of symbols in an expression:

```
size :: AST dom a → Int
size (Sym _)  = 1
size (s :$ a) = size s + size a
```

Note that this function is defined for all possible domains, which means that it can be reused in all kinds of language implementations. Such traversals are the basis of the generic interpretation and transformation functions provided by Syntactic.

3.2 Extensible Syntax

Support for generic traversals is one of the key features of the AST type. Another – equally important – feature is support for extensible syntax trees. We can note that Expr_2 is closed in the same way as Expr_1: Adding a constructor requires changing the definition of $\mathsf{NumDomain}_2$. However, the AST type turns out to be compatible with *Data Types à la Carte* [17], which is a technique for encoding open data types in Haskell.[4]

The idea is to create symbol domains as co-products of smaller independent domains using the (:+:) type operator (provided by Syntactic). To demonstrate the idea, we split $\mathsf{NumDomain}_2$ into two separate sub-domains and combine them into $\mathsf{NumDomain}_3$, used to define Expr_3:

```
data Lit3  a where Lit3  :: Int → Lit3 (Full Int)
data Add3 a where Add3 :: Add3 (Int :→ Int :→ Full Int)

type NumDomain3 = Lit3 :+: Add3

type Expr3 a = ASTF NumDomain3 a
```

The new type Expr_3 is again isomorphic Expr_1.

Now, the trick to get extensible syntax is to not use a closed domain, such as $\mathsf{NumDomain}_3$, but instead use constrained polymorphism to abstract away from the exact shape of the domain. The standard way of doing this for Data Types à la Carte is to use the inj method of the (:<:) type class (provided by Syntactic). Using inj, the smart constructors for Lit_3 and Add_3 can be defined thus:

```
lit3 :: (Lit3 :<: dom) ⇒ Int → ASTF dom Int
lit3 a = Sym (inj (Lit3 a))

add3 :: (Add3 :<: dom) ⇒ ASTF dom Int → ASTF dom Int → ASTF dom Int
add3 x y = Sym (inj Add3) :$ x :$ y
```

The definition of smart constructors can even be automated by using the function appSym (provided by Syntactic). The following definitions of lit_3 and add_3 are equivalent to the ones above:

```
lit3 :: (Lit3 :<: dom) ⇒ Int → ASTF dom Int
lit3 a = appSym (Lit3 a)

add3 :: (Add3 :<: dom) ⇒ ASTF dom Int → ASTF dom Int → ASTF dom Int
add3 = appSym Add3
```

[4] The original Data Types à la Carte uses a combination of type-level fixed-points and co-products to achieve open data types. Syntactic only adopts the co-products, and uses the AST type instead of fixed-points.

A constraint such as (Lit$_3$:<: dom) can be read as "dom contains Lit$_3$", which simply means that dom should be a co-product chain of the general form

```
(... :+: Lit₃ :+: ...)
```

One domain of this form is NumDomain$_3$, but any other domain that includes Lit$_3$ is also valid.

The fact that we have now achieved a modular language can be seen by noting that the definitions of Lit$_3$/lit$_3$ and Add$_3$/add$_3$ are *completely independent*, and could easily live in separate modules. Obviously, any number of additional constructs can be added in a similar way.

3.3 Syntactic Sugar

It is not very convenient to require all embedded programs to have the type AST. First of all, one might want to hide implementation details by defining a closed language:

```
type MyDomain = Lit₃ :+: Add₃

newtype Data a = Data {unData :: ASTF MyDomain a}
```

In fact this is exactly how Feldspar's Data type (see section 1) is defined (although with a different symbol domain).

Secondly, it is sometimes more convenient to use more "high-level" representations as long as these representations have a correspondence to an AST. Such high-level types are referred to as "syntactic sugar". Examples of syntactic sugar used in Feldspar are:

– The Data type

– Haskell tuples

– The Vector type (section 1.2)

One illustrating example is the fib function from section 1.1:

```
fib :: Data Index → Data Index
fib n = fst $ forLoop n (1,1) $ λi (a,b) → (b,a+b)
```

Here, the initial state is the *ordinary Haskell pair* (1,1). The body matches on the state a Haskell pair and constructs a new one as the next state. Finally, the fst function selects the first part of the state as the final result.

Syntactic sugar is defined by the class in Figure 9. The desugar method converts from a high-level type to a corresponding AST representation, and sugar converts

```
class Typeable (Internal a) ⇒ Syntactic a dom | a → dom
  where
    type Internal a
    desugar :: a → ASTF dom (Internal a)
    sugar   :: ASTF dom (Internal a) → a

instance Typeable a ⇒ Syntactic (ASTF dom a) dom
  where
    type Internal (ASTF dom a) = a
    desugar = id
    sugar   = id
```

Fig. 9. Syntactic sugar

back. The associated type function Internal maps the high-level type to its internal representation. Note that this type function does not need to be injective. It is possible to have several syntactic sugar types sharing the same internal representation.

The Syntactic instance for Data looks as follows:

```
instance Typeable a ⇒ Syntactic (Data a) MyDomain
  where
    type Internal (Data a) = a
    desugar = unData
    sugar   = Data
```

In order to make a user interface based on syntactic sugar, such as Data, we simply use the function sugarSym instead of appSym that was used in section 3.1:

```
lit :: Int → Data Int
lit a = sugarSym (Lit₃ a)

add :: Data Int → Data Int → Data Int
add = sugarSym Add₃
```

As we can see, sugarSym is a highly overloaded function. But as long as it is given a sufficiently constrained type signature (that is compatible with the signature of the given symbol), it will just do "the right thing".

4 Feldspar Implementation

In this section, we give an overview of Feldspar's implementation. Although the back-end is a large part of the implementation (Figure 5), it will not be treated in this text. See reference [9] for more information about the back-end.

To make the presentation simpler and to highlight the modularity aspect, we will focus on a single language construct: parallel arrays.

4.1 Parallel Arrays

The syntactic symbols of Feldspar's array operations are defined in the Array type:

```
data Array a
  where
    Parallel :: Type a ⇒ Array (Length :→ (Index → a) :→ Full [a])
    Append   :: Type a ⇒ Array ([a] :→ [a] :→ Full [a])
    GetIx    :: Type a ⇒ Array ([a] :→ Index :→ Full a)
    SetIx    :: Type a ⇒ Array ([a] :→ Index :→ a :→ Full [a])

    ...
```

As we saw in section 1.1, we use [a] to denote an array with elements of type a. From now on, we will focus on the implementation of Parallel , and just note that the other constructs are implemented in a similar way.

After we have defined the syntactic symbol, we need to give it semantics. This is done by declaring the following instances:

```
instance Semantic Array
  where
    semantics Parallel = Sem "parallel"
        (λlen ixf → genericTake len $ map ixf [0..])

    ...


instance Render   Array where renderPart = renderPartSem
instance Eval     Array where evaluate = evaluateSem
instance EvalBind Array where evalBindSym = evalBindSymDefault
```

The Semantic instance says that Parallel has the name " parallel ", and that it is evaluated using the given lambda expression. The succeeding instances give access to functions like drawAST and eval, by deriving their behavior from the Semantic instance. This means that whenever we run something like:

```
*Main> eval $ parallel 10 (*2)
[0,2,4,6,8,10,12,14,16,18]
```

it is the function in the above semanticEval field that does the actual evaluation.

The implementation contains a number of other trivial class instances, but we will omit those from the presentation.

Now it is time to define the user interface to Parallel . This follows the exact same pattern as we saw in section 3.3:

```
parallel :: Type a ⇒ Data Length → (Data Index → Data a) → Data [a]
parallel = sugarSym Parallel
```

Note how the function (Index → a) in the signature for Parallel became a function (Data Index → Data a) in the user interface. All of this is handled by the sugarSym function.

In addition to the above simple declarations, the implementation of parallel also consists of optimization rules and code generation, which are out of the scope of these notes. However, it is important to look at what we get from those few lines of code that we have given so far. It turns out to be quite a lot:

We have defined a typed abstract syntax symbol Parallel and its corresponding user function parallel . We have derived various interpretation functions (evaluation, rendering, alpha-equivalence, etc.) by providing very minimal information about the specific nature of parallel . We even get access to various syntactic transformations (constant folding, invariant code hoisting, etc.) without adding any additional code. All of this is due to the generic nature of the Syntactic library. Note also that the implementation of parallel is completely independent of the other constructs in the language, a property due to the extensible syntax provided by Syntactic (section 3.2).

Having seen the important bits of how the core language is implemented we can now move on to see how the vector library is implemented on top of the core (recall Figure 5).

4.2 Assembling the Language

Once a number of symbol types (such as Array above) have been defined, they are assembled using the same pattern as in section 3.3,

```
newtype Data a = Data {unData :: ASTF FeldDomainAll a}
```

where FeldDomainAll is the complete symbol domain. Additionally, to make type signatures look nicer, we define the Syntax class recognized from the examples in section 1:

```
class
    ( Syntactic a FeldDomainAll
    , SyntacticN a (ASTF FeldDomainAll (Internal a))
    , Type (Internal a)
    ) ⇒
      Syntax a

instance Type a ⇒ Syntax (Data a)
```

Syntax does not have any methods; it is merely used as an alias for its super-class constraints. The most important constraint is Syntactic a FeldDomainAll, which can now be written more succinctly as Syntax a. In other words, all functions overloaded by Syntax get access to the syntactic sugar interface described in section 3.3.

The Type class is the set of all value types supported by Feldspar (for example, Bool, Int32, (Float,Index), etc.). The SyntacticN class is beyond the scope of these notes; interested readers are referred to the API documentation [1].

4.3 Vector Library

The vector library (module Feldspar.Vector) provides a type for "virtual" vectors – vectors that do not (necessarily) have any run-time representation. Vectors are defined as:

```
data Vector a
    = Empty
    | Indexed
        { segmentLength :: Data Length
        , segmentIndex  :: Data Index → a
        , continuation  :: Vector a
        }
```

This recursive type can be seen as a list of segments, where each segment is defined by a length and an index projection function. The reason for having vectors consisting of several segments is to allow efficient code generation of vector append. However, in this presentation, we are going to look at a simpler vector representation, consisting only of a single segment:[5]

```
data Vector a
    = Indexed
        { length :: Data Length
        , index  :: Data Index → a
        }
```

This is essentially a pair – at the Haskell-level – of a length and an index projection function. The meaning of a non-nested vector is given by the following function:

```
freezeVector :: Type a ⇒ Vector (Data a) → Data [a]
freezeVector vec = parallel (length vec) (index vec)
```

[5] Note that for programs that do not use the (++) operation (which is the case for all but one of the examples in this document), there will only ever be a single segment, in which case the two representations are equivalent.

That is, a Vector with a given length and index projection has the same meaning as a parallel with the same length and projection function. A small example:

```
*Main> eval $ freezeVector $ Indexed 10 (*2)
[0,2,4,6,8,10,12,14,16,18]
```

With this simple representation of vectors, it becomes straightforward to define many of Haskell's standard operations on lists. Some examples are given in figure 10.

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

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

zip :: Vector a → Vector b → Vector (a,b)
zip a b = Indexed (length a `min` length b)
                  (λi → (index a i, index b i))

zipWith :: (a → b → c) → Vector a → Vector b → Vector c
zipWith f a b = map (uncurry f) $ zip a b

fold :: Syntax a ⇒ (a → b → a) → a → Vector b → a
fold f a (Indexed len ixf) = forLoop len a (λi st → f st (ixf i))

sum :: (Num a, Syntax a) ⇒ Vector a → a
sum = fold (+) 0
```

Fig. 10. Definition of some vector operations

Does it work? Let us check:

```
*Main> eval $ freezeVector $ map (*2) $ Indexed 10 (*2)
[0,4,8,12,16,20,24,28,32,36]
```

This is all very well, but things start to get really interesting when we note that we can actually make Vector an instance of Syntactic. A first attempt at doing this might be:

```
instance Type a ⇒ Syntactic (Vector (Data a)) FeldDomainAll
  where
    type Internal (Vector (Data a)) = [a]
    desugar = desugar ∘ freezeVector
    sugar   = thawVector ∘ sugar
```

```
thawVector :: Type a ⇒ Data [a] → Vector (Data a)
thawVector arr = Indexed (getLength arr) (getIx arr)
```

The function thawVector is the inverse of freezeVector. This works, but only for non-nested vectors. A better solution is given in figure 11. This instance works for elements of any Syntax type, which means that it even handles nested vectors.

```
instance Syntax a ⇒ Syntactic (Vector a) FeldDomainAll
  where
    type Internal (Vector a) = [Internal a]
    desugar = desugar ∘ freezeVector ∘ map (sugar ∘ desugar)
    sugar   = map (sugar ∘ desugar) ∘ thawVector ∘ sugar

instance Syntax a ⇒ Syntax (Vector a)
```

Fig. 11. Syntactic instance for Vector

Having a Syntactic instance for Vector means that they can now work seamlessly with the rest of the language. Here is an example of a function using (?) to select between two vectors:

```
f :: Vector (Data Index) → Vector (Data Index)
f vec = length vec > 10 ? (take 10 vec, map (*3) vec)
```

Since Feldspar's eval function is also overloaded using Syntax,

```
eval :: Syntax a ⇒ a → Internal a
```

we can even evaluate vector programs directly just like any other Feldspar program:

```
*Main> eval f [5,6,7]
[15,18,21]
```

It is important to note here that Vector is an ordinary Haskell type that is not part of Feldspar's core language. Relating to figure 5, Vector lives in one of the top boxes of the API, and is not part of the core language. This means that the back ends have no way of knowing what a Vector is. The reason vectors are still useful is that we have an automatic translation between vectors and core expressions via the Syntactic class. This technique provides a very powerful, yet very simple, way of extending the language with new constructs.

Vector Fusion The fact that vectors are not part of the core language, has the nice consequence that they are guaranteed to be removed at compile time. This is the underlying explanation for the kind of fusion that was seen in section 1.2. Take, for example, the scalar product function:

```
scalarProd :: (Num a, Syntax a) ⇒ Vector a → Vector a → a
scalarProd as bs = sum (zipWith (∗) as bs)
```

Using the definitions of sum, zipWith, zip and map in figure 10, scalarProd can be transformed in the following sequence of steps:

```
−− Definition of zipWith and zip
scalarProd2 as bs
    = sum (
        map (uncurry (∗)) (
          Indexed
            (length as `min` length bs)
            (λi → (index as i, index bs i))
        )
      )

−− Definition of map
scalarProd3 as bs
    = sum (
        Indexed
          (length as `min` length bs)
          (λi → index as i ∗ index bs i)
      )

−− Definition of sum
scalarProd4 as bs
    = forLoop
        (length as `min` length bs)
        0
        (λi st → st + index as i ∗ index bs i)
```

As we can see, the end result is a single forLoop, where the multiplication and the accumulation have been fused together in the body. Note that these reductions are performed by Haskell's evaluation, which is why we can guarantee *statically* that expressions of this form will always be fused.

The only exception to this static guarantee is the function freezeVector which will compute the vector using parallel (which is not guaranteed to be fused). Functions outside of the vector API (such as forLoop) can only access vectors using desugar/sugar. Since desugar implicitly introduces freezeVector, this means that functions outside of the vector API will not be able to guarantee fusion.

We have chosen to implement vectors as an additional high-level library. It would have been possible to express all vector operations directly in the core language, and implement fusion as a syntactic transformation. However, then we would not have been able to guarantee fusion in the same way as we can now. Imagine we had this core-level implementation of `reverse` (available in the vector library):

```
rev :: Type a ⇒ Data [a] → Data [a]
rev arr = parallel l (λi → getIx arr (l−i−1))
  where
    l = getLength arr
```

Then we would generally not be able to tell whether it will be fused with the array `arr`. If `arr` is produced by another `parallel`, fusion is possible, but if `arr` is produced by, for example, `sequential` (section 1.1), fusion is not possible. This is because `sequential` can only produce its element in ascending order, while `rev` indexes in reverse order. The vector library `reverse`, on the other hand, will *unconditionally* fuse with its argument.

5 Discussion

We have presented Feldspar, an embedded language for DSP algorithm design. One key aspect of Feldspar is that it is purely functional, despite the fact that what we wish to do is to provide an alternative to C, which is currently used for DSP programming. We have shown how Feldspar consists of a small core at about the same abstraction level as C, and libraries built upon the core that raise the level of abstraction at which the programmer works. Thus we intend to bring the benefits of functional programming to a new audience. As a result we do not really have a novel language design to present, but rather a new setting in which functional programming with a strong emphasis on higher order functions can be used. Doing array programming in a relatively simple, purely functional language allows the Feldspar user to construct algorithmic blocks from smaller components. The purely functional setting gives a kind of modularity that is just not present in C. It is easy to explore algorithms by plugging components together in new ways. One can remove just part of an algorithm and replace it with a function with the same input-output behaviour, but perhaps different performance. The fact that Feldspar programs are compact is important here. In the first part of these notes, we tried to illustrate this aspect of Feldspar. Our hope is that this ease of algorithm exploration will be a key benefit of taking the step from C to Feldspar.

In the implementation sections, we tried to convey the most important parts of Feldspar's implementation, focusing mainly on the underlying principles (the Syntactic library), but also showing concrete details of the implementation of `parallel` and the vector library.

There was not enough room to go into all details of the implementation. Readers who are interested more details are encouraged to look at NanoFeldspar, a small proof-of-concept implementation of Feldspar shipped with the Syntactic package. To download NanoFeldspar, simply run:

```
> cabal unpack syntactic-0.8
> cd syntactic-0.8/Examples/NanoFeldspar
```

NanoFeldspar contains simplified versions of Feldspar's core language and the vector library. There is no back-end, but it is possible to print out the syntax tree to get an idea of what the generated code would look like. NanoFeldspar follows the modular implementation style described in section 3.2, and it should be perfectly possible to use NanoFeldspar as a basis for implementing other embedded languages.

Some additional details of the Feldspar implementation can be found in our report on adding support for mutable data structures to Feldspar [15]. This paper gives a very nice example of modular language extension using Syntactic.

An important part of Feldspar's implementation is the ability to add new libraries, such as the vector library, without changing the existing core language or the code generator. In addition to the vector library, Feldspar has (more or less experimental) libraries for synchronous streams, Repa-style arrays [14], bit vectors and fixed-point numbers, etc.

5.1 Limitations

Feldspar currently only generates pure algorithmic functions. What is missing in order to develop a complete application in Feldspar is the ability to coordinate the generated functions. This requires language support for parallelism and concurrency, memory management, real-time scheduling, etc. We are currently working on adding such support to the language.

Syntactic Although Syntactic has worked very well for the implementation of Feldspar, we do not expect it to be suitable for all kinds of embedded languages. While the AST type can model a wide range of data types, it does not handle mutually recursive types. For example, AST is not suited to model the following pair of data types:

```
type Var = String

data Expr a where
  Var  :: Var → Expr a
  Lit  :: Num a ⇒ a → Expr a
  Add  :: Num a ⇒ Expr a → Expr a → Expr a
  Exec :: Stmt → Var → Expr a

data Stmt where
  Assign :: Var → Expr a → Stmt
  Seq    :: Stmt → Stmt → Stmt
  Loop   :: Expr Int → Stmt → Stmt
```

```
    ...
```

Here, Expr is an expression language capable of embedding imperative code using the Exec constructor. Stmt is an imperative language using the Expr type for pure expressions. In the AST type, all symbols are “first-class”, which means that we cannot easily group the symbols as in the example above.

Note, however, that the above language can easily be modeled as a single data type with monadic expressions. In fact, the latest Feldspar release has support for mutable data structures with a monadic interface. Their implementation is described in [15].

It is also important to be aware that many of the reusable components provided by Syntactic (syntactic constructs, interpretations, transformations, etc.) assume that the language being implemented has a pure functional semantics. However, this is not a limitation of the AST type itself, but rather of the surrounding utility library. There is nothing preventing adding utilities for different kinds of languages if the need arises.

5.2 Related Work

Work related to Feldspar and its implementation has been covered by previous publications [3,2].

Syntactic shares common goals with a lot of related work on implementation of domain-specific languages. However, in the context of strongly typed embedded languages, Syntactic is rather unique in providing a library of reusable building blocks for language implementation. Its support for language extension is derived from Data Types à la Carte [17]. A quite different approach to extensible embedded languages is Finally Tagless [6]. Although very elegant, neither of these techniques provides libraries of reusable implementation tools.

6 Conclusion

Feldspar is a slightly strange beast: an embedded language in Haskell that tries to be as much like Haskell (or at least a simple subset of Haskell) as possible. Once one has chosen this direction, the hard work is not in language design but in finding ways to present the user with this illusion, while generating high performance C code. We have (so far) performed only one brief test in which Ericsson engineers used Feldspar. The generated code was of satisfactory quality but what was most striking about the experiment was the realisation, by observing the reactions of our Ericsson colleagues, that it is purely functional programming, with all its familiar benefits, that we are trying to sell, and not a new language called Feldspar!

Building on the Syntactic library, Feldspar has a modular, easily extensible implementation. Much of its functionality is derived from the generic building blocks provided from Syntactic. This has been demonstrated concretely in the implementation of `parallel` (section 4.1):

- The implementation of `parallel` is independent of other language constructs.
- The implementation of `parallel` is covered by a few tens of lines of code, mostly in declarative form.
- A lot of details are handled by the Syntactic library: evaluation, rendering, alpha-equivalence, certain optimizations, etc. Very little extra code is needed to make these generic functions work for `parallel`.

Furthermore, the vector library (section 4.3) is implemented as an additional library completely separate from the core language. This design allows us to implement many of Haskell's list processing functions in just a few lines each, and still be able to generate high-performance C code from vector-based programs.

Appendix: C code from the `fft4` function

The first page of code contains loops for computing twiddle factors and for doing bit reversal. The second page contains the two nested for loops that do the FFT calculation. (The code is split only for display purposes.)

```
/*
 * Memory information
 *
 * Local: complex float array, complex float array
 * Input: unsigned 32-bit integer, complex float array
 * Output: complex float array
 *
 */
void test(struct array * mem, uint32_t v0, struct array * v1, struct array *
out)
{
    uint32_t v19;
    uint32_t v20;
    float complex v21;
    uint32_t len0;
    uint32_t v23;
    uint32_t len1;

    v19 = ~(~((4294967295 << v0)));
    v20 = ~((4294967295 << v0));
    v21 = complex_fun_float((float)((1 << v0)), 0.0f);
    len0 = (1 << (v0 - 1));
    for(uint32_t v15 = 0; v15 < len0; v15 += 1)
    {
        at(float complex,&at(struct array,mem,0),v15) =
        cexpf(((0.0f+0.0fi) - ((complex_fun_float((float)(v15), 0.0f) *
              (0.0f+6.2831854820251465fi)) / v21)));
    }
    setLength(&at(struct array,mem,0), len0);
    v23 = (v0 - 1);
    len1 = getLength(v1);
    for(uint32_t v11 = 0; v11 < len1; v11 += 1)
    {
        at(float complex,out,v11) = at(float complex,v1,((v19 & v11) |
rotateL_fun_uint32(reverseBits_fun_uint32((v20 & v11)), v0)));
    }
    setLength(out, len1);
    for(uint32_t v12 = 0; v12 < v0; v12 += 1)
    {
        uint32_t v24;
        uint32_t v25;
        uint32_t v26;
        uint32_t len2;

        v24 = (1 << v12);
        v25 = pow_fun_uint32(2, v12);
        v26 = (v23 - v12);
        len2 = getLength(out);
        for(uint32_t v14 = 0; v14 < len2; v14 += 1)
        {
            uint32_t v27;
            uint32_t v28;
            float complex v29;
            float complex v30;

            v27 = testBit_fun_uint32(v14, v12);
            v28 = (v14 ^ v24);
            if(testBit_fun_uint32(v28, v12))
            {
                v29 = (at(float complex,&at(struct array,mem,0),
                       ((v28 % v25) << v26)) * at(float complex,out,v28));
            }
            else
            {
                v29 = at(float complex,out,v28);
            }
            if(v27)
            {
                v30 = (at(float complex,&at(struct array,mem,0),
                       ((v14 % v25) << v26)) * at(float complex,out,v14));
            }
            else
            {
                v30 = at(float complex,out,v14);
            }
            if(v27)
            {
                at(float complex,&at(struct array,mem,1),v14) = (v29 - v30);
            }
            else
            {
                at(float complex,&at(struct array,mem,1),v14) = (v30 + v29);
            }
        }
        setLength(&at(struct array,mem,1), len2);
        copyArray(out, &at(struct array,mem,1));
    }
}
```

Acknowledgements

Work on Feldspar has been funded by Ericsson, the Swedish basic research funding agency (Vetenskapsrådet), the Swedish Foundation for Strategic Research (SSF), and the Hungarian National Development Agency.

Feldspar is a cooperation between Ericsson, Chalmers University and Eötvös Loránd (ELTE) University. The authors would like to thank the Feldspar team at Chalmers and ELTE for fruitful collaboration.

We acknowledge the constructive suggestions of an anonymous referee, who helped us to improve these notes.

References

1. Syntactic library, version 0.8. http://hackage.haskell.org/package/syntactic-0.8.
2. E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar – an embedded language for digital signal processing. In *22nd International Symposium, IFL 2010*, volume 6647 of *LNCS*, 2011.
3. Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and Andras Vajda. Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE Computer Society, 2010.
4. Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.
5. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 174–184. ACM, 1998.
6. J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.
7. Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30. ACM, 2012.
8. J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, April 1965.
9. 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 *ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems*, 2010.
10. P. Duhamel and Vetterli M. Fourier Transforms: A Tutorial Review and a State of the Art. In *Digital Signal Processing Handbook, Ed. Vijay K. Madisetti and Douglas B. Williams*. CRC Press LLC, 1999.

11. Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13:3:455– 481, 2003.
12. Andy Gill. Type-safe observable sharing in Haskell. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 117–128. ACM, 2009.
13. Paul Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
14. Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272. ACM, 2010.
15. Anders Persson, Emil Axelsson, and Josef Svenningsson. Generic monadic constructs for embedded languages. In *IFL '11: Implementation and Application of Functional Languages*, 2011. To be published.
16. Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. 14th ACM SIGPLAN international conference on Functional programming*, pages 341–352. ACM, 2009.
17. Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(04):423–436, 2008.
18. P. Wadler. The expression problem. http://www.daimi.au.dk/~madst/tool/papers/expression.txt, 1998.