The Design and Implementation of Feldspar

Conference Paper · October 2011

DOI: 10.1007/978-3-642-24276-2_8

Emil Axelsson
Chalmers University of Technology

Mary Sheeran
Chalmers University of Technology

David Engdal
Ericsson

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

Emil Axelsson, Mary Sheeran, Koen Claessen,
Josef Svenningsson, David Engdal, Anders Persson

Chalmers University of Technology

Ericsson

September 6, 2010

Feldspar background

Functional **E**mbedded **L**anguage for **DSP** and **PAR**allelism

- Joint project:
  - Ericsson
  - Chalmers University (Gothenburg)
  - Eötvös Loránd University (Budapest)
- Aims to raise the abstraction level of digital signal processing (DSP) software
  - Improving portability, maintainability, development time, etc.

Feldspar background

Functional **E**mbedded **L**anguage for **DSP** and **PAR**allelism

- Joint project:
  - Ericsson
  - Chalmers University (Gothenburg)
  - Eötvös Loránd University (Budapest)
- Aims to raise the abstraction level of digital signal processing (DSP) software
  - Improving portability, maintainability, development time, etc.
- Embedded in Haskell
- Open source:

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

  http://hackage.haskell.org/package/feldspar-compiler

Example: Pre-processing stage

Feldspar code

```
prepare
  :: DVector Float → DVector Float
  → DVector Float → DVector Float

prepare x y z =
    permute (λlen i → len−i−1)
            (norm x .∗ y .∗ z)
  where
    (.∗) = zipWith (∗)
    norm = map (/15)
```

Example: Pre-processing stage

Feldspar code

```
prepare
  :: DVector Float → DVector Float
  → DVector Float → DVector Float

prepare x y z =
    permute (λlen i → len−i−1)
            (norm x .* y .* z)
  where
    (.*) = zipWith (*)
    norm = map (/15)
```

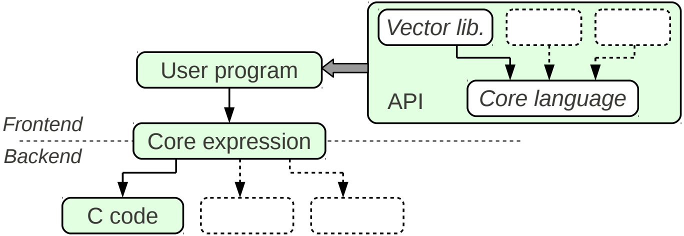
Resulting C code

```
...

for(var9 = 0;
    var9 < (* out_0);
    var9 += 1)
{
  int32_t var11;
  var11 = (((* out_0) − var9) − 1);
  out_1[var9] =
    (((var0_0_1[var11] / 15.0f)
     * var0_1_1[var11])
     * var0_2_1[var11]);
}

...
```

Architecture



- Small, machine-oriented core language – simplifying compilation
- Use the host language to build high-level interfaces
- Extensible language design!

Core language API

```
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]


-- Primitive functions:
not :: Data Bool → Data Bool
mod :: Integral a ⇒ Data a → Data a → Data a
...
```

Core language API

```
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]


-- Primitive functions:
not :: Data Bool → Data Bool
mod :: Integral a ⇒ Data a → Data a → Data a
...
```

Purely functional!

Core language example: Pre-processing stage

Feldspar code

```
prepare
    :: Data Int
    → Data [Float] → Data [Float] → Data [Float] → Data [Float]

prepare len x y z = parallel len $ λi →
                          let j = len−i−1
                           in (x!j / 15) ∗ (y!j) ∗ (z!j)
```

Core expressions

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

Optimization in front-end

- Reusing some standard techniques from
  Elliot, Finne, Moor. Compiling embedded languages. *JFP 2003*.
  - Constant folding on the fly
  - Variable hoisting
- User-assisted inference of static sizes
- Experimental size-based partial evaluation

Extending the core language

Many core language operations are overloaded by Computable:

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

eval :: Computable a ⇒ a → Internal a

icompile :: (Computable a, Computable b) ⇒ (a → b) → IO ()
  -- (Actually more general...)
```

- Computable contains types like Data a, (Data a, Data b), etc.
- Easily extended with more high-level types
- Extensible core language

Vector library

Simple extension of the core language:

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

type DVector a = Vector (Data a)

instance Storable a ⇒ Computable (Vector (Data a))
```

Vector operations

Straightforward definitions of familiar operations:

```
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

permute :: (Data Length → Data Ix → Data Ix) → (Vector a → Vector a)
permute perm (Indexed l ixf) = Indexed l (ixf ∘ perm l)

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

Vector consumption

```
freezeVector :: Storable a ⇒ Vector (Data a) → Data [a]
freezeVector (Indexed l ixf) = parallel l ixf

fold :: Computable a ⇒ (a → b → a) → a → Vector b → a
fold f x (Indexed l ixf) = for 0 (l−1) x (λi s → f s (ixf i))
```

(The for loop is translated to a while loop.)

Fusion

```
prog = fold (+) 0 $ map (*2) $ enumFromTo 1 10
```

Intermediate vectors guaranteed (!) to be removed:

```
prog = fold (+) 0 $ map (*2) $ Indexed 10 (+1)
     = fold (+) 0 $ Indexed 10 ((*2) ∘ (+1))
     = for 0 9 0 $ λi s → (+) s (((*2) ∘ (+1)) i)
```

Fusion can be avoided using memorize:

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

memorize is the only vector operation that allocates memory!

Conclusion

- Simple implementation (included in the paper):
  Powerful combination of simple, extensible core language + high-level interfaces
- Vector library provides high-level coding style with predictable memory usage
- Feldspar already works for (some) DSP algorithms
  - First case-study: Baseband channel estimation at Ericsson
  - Pseudo-random sequence generation required low-level coding (i.e. explicit loops)

Shortcomings and future work

- Vector library does not support "streaming" computations
- Core language too simple – many useful C code patterns are inexpressible
- Pure functions $\Rightarrow$ poor control over resources, timing, etc.

Shortcomings and future work

- Vector library does not support "streaming" computations
  - We are developing a separate high-level library for streams
- Core language too simple – many useful C code patterns are inexpressible
- Pure functions $\Rightarrow$ poor control over resources, timing, etc.

Shortcomings and future work

- Vector library does not support "streaming" computations
  - We are developing a separate high-level library for streams
- Core language too simple – many useful C code patterns are inexpressible
  - We are developing a more expressive core language
- Pure functions $\Rightarrow$ poor control over resources, timing, etc.

Shortcomings and future work

- Vector library does not support “streaming” computations
  - We are developing a separate high-level library for streams
- Core language too simple – many useful C code patterns are inexpressible
  - We are developing a more expressive core language
- Pure functions $\Rightarrow$ poor control over resources, timing, etc.
  - We are developing a “system layer”