Programmable Signatures

Anders Persson and Emil Axelsson

Chalmers University of Technology

Abstract. When compiling Embedded Domain Specific Languages (EDSLs) into other languages, the compiler translates types in the source language into corresponding types in the target language. The translation is often driven by a small set of rules that map a single type in the source language into a single type in the target language. This simple approach is limiting when there are multiple possible mappings, and it may lead to poor interoperability and performance in the generated code.
Instead of hard-wiring a single set of translation rules into a compiler, this paper introduces a small language that lets the programmer describe the mapping of each argument and function separately.

1 Introduction

Feldspar is an embedded domain specific language written in Haskell [1, 2].[1] The purpose of Feldspar is to implement high-performance software, especially in the domain of signal processing in embedded systems , see reference [3].

Feldspar comes with an optimizing compiler that translates Feldspar expressions into C99 code.[2] When translating a function signature, the compiler uses a specific calling convention as detailed in chapter 1.4.2 in reference [3]:

- All functions return `void`
- Scalar values are passed by value
- Structured values (structs, arrays) are passed by reference
- Arrays are represented using a data structure `struct array`
- Return values are passed through caller provided pointers

For example, the following is the type of an FFT function in Feldspar:

```
1 fft :: Data [Double] → Data [Double]
```

The type constructor `Data` denotes a Feldspar expression, and its parameter denotes the type of value computed by that expression. For historical reasons, Feldspar uses `[]` to denote immutable arrays.

The compiler translates the `fft` function into the following C99 signature,

[1] https://hackage.haskell.org/package/feldspar-language
[2] https://hackage.haskell.org/package/feldspar-compiler

```
void fft(struct array* v0, struct array** out);
```

where `struct array` is a Feldspar specific data structure with metadata, such as the number of elements, and a pointer to the data area.

The Feldspar compiler uses its calling convention for a number of reasons, but the primary reasons are consistency and generality. The convention ensures that all arguments fit into a register, which helps avoid spilling arguments to the call stack. By passing arrays as references bundled with their length, the compiler can generate code that works with different array sizes and still preserve the same number of arguments.

A hard-wired set of mapping rules can be restrictive and introduce performance penalties. Code generated from Feldspar will be part of a larger system, and the calling convention is naturally dictated by the system rather than by the Feldspar compiler.

1.1 Issues with Fixed Mappings

With a fixed signature mapping, it is easy to derive the target language type from the source language type. But the fixed mapping leaves little room to change the generated signature to fit into existing software. Instead, separate wrapper functions have to be written and maintained.

In a typical embedded system, arrays are passed as two arguments: a pointer to the data buffer and an integer that gives the number of elements of the array. However, there are many variations on this theme. Should the length come before or after the buffer? Can the length argument be used for more than one array if they always have the same length? And so on.

Even if we allow flags to customize the compiler, a fixed set of mapping rules will never be able to cover all possible situations. Instead, we would like to put the exported signature in the hands of the programmer.

As a concrete example, take the following function for computing the scalar product of two vectors:

```
scProd :: Data [Double] → Data [Double] → Data Double
```

The generated signature with the default mapping is:

```
void scProd(struct array* v0, struct array* v1, double* out);
```

By default, the Feldspar compiler automatically makes up names for the arguments. Apart from the problem that Feldspar's `struct array` is an unconventional array representation, this code may also be considered too general: it has to cater for the fact that the arrays may have different lengths. Since it does not make

sense to call `scProd` with arrays of different lengths, a more appropriate signature might be:

```
1 double scProd(uint32_t len, double* v1_buf, double* v2_buf);
```

Here, the arrays are passed as two pointers to the corresponding data buffers and a single length argument. This signature is more likely to occur in a practical system, and it has the advantage that the function does not have to decide what to do if the lengths are different. However, the system may expect a different order of the arguments, and might expect the result to be passed by value instead of by reference.

In addition to being able to customize the calling convention, we might also want to affect non-functional aspects of functions. For example, we can name arguments for readability and debugging purposes. This is helpful since Feldspar is an embedded language and that syntactic information is lost when the Haskell compiler reads the source file.

In future work we want to extend the annotations to include attributes to help the C compiler, including `restrict` and `volatile`.

1.2 Contributions

To address the problems above, this paper presents two contributions:

- We define a simple EDSL to specify type conversions and annotations when exporting a Feldspar function to an external system (section 2).
- We give an implementation of the EDSL as a small wrapper around the existing Feldspar compiler (section 3). The implementation relies on a simple interface to the underlying compiler, and the technique can easily be ported to other EDSLs for which the compiler implements the same interface.

2 The Signature Language

Dissatisfied with hard-wired rules and global compiler options, we propose a small language as a more flexible way to drive the compiler.

The Signature language allows the programmer to express the mapping of individual arguments separately. Specifically it allows the programmer to add annotations to every argument and control the data representation. These annotations can be as simple as just giving a name to a parameter, using the `name` combinator. Or, it can change the arity of the function by introducing new parameters, like the `native` and `exposeLength` combinators, which we will show later in this chapter.

```
-- | Capture an argument
lam :: (Type a) => (Data a → Signature b) → Signature (a → b)

-- | Capture and name an argument
name :: (Type a) => String → (Data a → Signature b) → Signature (a → b)

-- | Create a named function return either by value or reference
ret :: (Type a) => String → Data a → Signature a
ptr :: (Type a) => String → Data a → Signature a
```

Listing 1.1. Signature language (shallow embedding)

Like the Feldspar language, the Signature language is a typed embedded domain specific language, embedded in Haskell. The Signature language preserves the type safety of the Felspar language.

The Signature language interface is given in listing 1.1. The combinators `lam` and `name` are used to bind (and possibly annotate) an argument, while `ret` and `ptr` are used to return the result of the function to be generated.

As our running example, we will reuse the `scProd` function from section 1.1.

```
scProd :: Data [Double] → Data [Double] → Data Double
```

We can mimic the standard rules of the Feldspar compiler by wrapping the function in our combinators.

```
ex1 = lam $ λxs → lam $ λys → ptr "scProd" (scProd xs ys)
```

which generates the following C signature when compiled

```
void scProd(struct array* v0, struct array* v1, double* out);
```

Using `name` instead of `lam`, we change the embedding to name the first argument

```
ex2 = name "xs" $ λxs → lam $ λys → ptr "scProd" (scProd xs ys)
```

resulting in

```
void scProd(struct array* xs, struct array* v1, double* out);
```

Finally, we change the function to return by value, by using `ret` instead of `ptr`

```
ex3 = name "xs" $ λxs → name "ys" $ λys → ret "scProd" (scProd xs ys)
```

which produces

```
double scProd(struct array* xs, struct array* ys);
```

The basic constructors in the language are useful for simple annotations on the arguments. But it is also possible to create constructors that will change the arity or introduce interface code into the embedded function. The interface code can bridge different representation formats.

Without the `Signature` language, we would have to write a C wrapper around the generated function. A wrapper written in C is not polymorphic, but declared with concrete types, like `int` or `double`. In contrast, the Feldspar functions are often polymorphic and the concrete types are decided at compile time. A handwritten wrapper would have to change for different concrete types, and thus becomes a maintenance burden. Also, the wrapper code is a separate function and can not be optimized together with the generated code. In contrast, the `Signature` language combinators are applied before optimization and code generation, and the wrapper code fuses with the function.

For example, consider the `scProd` function again. In earlier versions it suffered from two problems.

1. The two arrays may have different lengths and the generated code has to defensively calculate the minimum length (see line 6 below).
2. The arrays are passed using a `struct` **`array`** pointer which results in extra dereferencing (line 9 below).

```
1  void scProd(struct array* v0, struct array* v1, double* out)
2  {
3      double e4;
4      uint32_t len5;
5
6      len5 = min(getLength(v0), getLength(v1));
7      e4 = 0.0;
8      for (uint32_t v2 = 0; v2 < len5; v2 += 1) {
9          e4 = e4 + at(double, v0, v2) * at(double, v1, v2);
10     }
11     *out = e4;
12 }
```

To help alleviate these problems we can define smart constructors that modify the code before optimization. Note that these smart constructors are extensions to the `Signature` language and can be expressed by the end user.

```
1  -- | Pass the argument as a native array of length @len@
2  native :: (Type a)
3         => Data Length → (Data [a] → Signature b) → Signature ([a] → b)
4  native l f = Lam (Native l) $ λa → f $ F.setLength l a
5
6  -- | Expose the length of an array
7  exposeLength :: (Type a)
8               => (Data [a] → Signature b) → Signature (F.Length → [a] → b)
9  exposeLength f = name "len" $ λl → native l f
```

The `native` function changes the array type to a native C array with length `l`. By using the Feldspar `setLength` function, size information is added to the array arguments. In section 3 we show how the `Native` constructor produces the interface code needed to translate between native and `struct array` formats.

The `exposeLength` function adds an extra length argument to the signature and passes this length to `native`. The effect is to break up a standard array argument into two arguments: a length and a native array.

With our new combinators, we can create a version of the `scProd` function that accepts native arrays of a fixed (runtime specified) length.

```
scProdNative = name "len" $ λlen →
               native len $ λas  →
               native len $ λbs  →
               ret "scProd" $ scProd as bs
```

which compiles to:

```
double scProd(uint32_t len, double* v1_buf, double* v2_buf)
{
    struct array v1 = {.buffer =v1_buf, .length =len, .elemSize =sizeof(double),
                       .bytes =sizeof(double) * len};
    struct array v2 = {.buffer =v2_buf, .length =len, .elemSize =sizeof(double),
                       .bytes =sizeof(double) * len};
    double e5;

    e5 = 0.0;
    for (uint32_t v3 = 0; v3 < len; v3 += 1) {
        e5 = e5 + at(double, &v1, v3) * at(double, &v2, v3);
    }
    return e5;
}
```

Note how the Feldspar compiler now realizes that both vectors have the same length, and thus removes the defensive minimum length calculation.

The first two declarations in the generated code are for converting the native array in the interface to `struct array` which is what the body of the function expects. In the future, we plan to make it possible to use native arrays throughout the generated code, when stated so in the signature, but that requires a change to the Feldspar compiler and is out of scope for this paper.

3 Implementation

The language is implemented as a combination of a shallow and a deep embedding. The shallow embedding (listing 1.1), which is also the programmer interface, provides combinators to describe the mapping of a function. The deep embedding (listing 1.2) is interpreted by the compiler to apply the rules.

```
-- | Annotations to place on arguments or result
data Ann a where
  Empty  :: Ann a
  Native :: Type a => Data F.Length → Ann [a]
  Named  :: String → Ann a

-- | Annotation carrying signature description
data Signature a where
  Ret    :: (Type a) => String → Data a → Signature a
  Ptr    :: (Type a) => String → Data a → Signature a
  Lam    :: (Type a)
         => Ann a → (Data a → Signature b) → Signature (a → b)
```

Listing 1.2. Signature Language (deep embedding)

By using two separate embeddings it is possible to have a small set of constructs that the compiler has to deal with, while at the same time provide a rich set of combinators to the end user. For example, the `exposeLength` function could be implemented purely in terms of simpler constructs. This way of combining deep and shallow embeddings has been shown to be very powerful for implementing EDSLs [4].

We can think of `Signature` as adding top-level lambda abstraction and result annotations to the existing expression language `Data`. The use of a host-language function in the `Lam` constructor is commonly known as *higher-order abstract syntax* (HOAS) [5]. HOAS allows us to construct signatures without the need to generate fresh variable names. As we will see in section 3.1, names are instead generated when we generate code from the signature.

3.1 Code generation

`Signature` is defined as a wrapper type around the Feldspar expression type `Data`. In order to generate code for signatures, we first need to be able to generate code for `Data`. To this end, the Feldspar compiler provides the following interface:

```
varExp    :: Type a             => VarId → Data a
compExp   :: (MonadC m)         => Data a → m C.Exp
compTypeF :: (MonadC m, Type a) => proxy a → m C.Type
```

The first function, `varExp`, is used to create a free variable in Feldspar. Naturally, this function is not exported to ordinary users. The function `compExp` is used to compile a Feldspar expression to a C expression `Exp`. Since compilation normally results in a number of C statements in addition to the expression, `compExp` returns in a monad `m` capable of collecting C statements that can later be pretty printed as C code. Finally, `compTypeF` is used to generate a C type from a type `a` constrained by Feldspar's `Type` class. The argument of type `proxy a` is just used to determine the type `a`.

```
-- | Compile a @Signature@ to C code
translateFunction :: forall m a. (MonadC m) => Signature a → m ()
translateFunction sig = go sig (return ())
  where
    go :: forall d. Signature d → m () → m ()
    go (Ret n a) prelude = do
      t ← compTypeF a
      inFunctionTy t n $ do
        prelude
        e ← compExp a
        addStm [cstm| return $e; |]
    go (Ptr n a) prelude = do
      t ← compTypeF a
      inFunction n $ do
        prelude
        e ← compExp a
        addParam [cparam| $ty:t *out |]
        addStm [cstm| *out = $e; |]
    go fun@(Lam Empty f) prelude = do
      t ← compTypeF (argProxy fun)
      v ← varExp <$> freshId
      C.Var n _ ← compExp v
      go (f v) $ prelude >> addParam [cparam| $ty:t $id:n |]
    go fun@(Lam n@(Native l) f) prelude = do
      t ← compTypeF (elemProxy n fun)
      i ← freshId
      let w = varExp i
      C.Var (C.Id m _) _ ← compExp w
      let n = m ++ "_buf"
      withAlias i ('&':m) $ go (f w) $ do
        prelude
        len ← compExp l
        addLocal [cdecl| struct array $id:m = { .buffer = $id:n
                                              , .length=$len
                                              , .elemSize=sizeof($ty:t)
                                              , .bytes=sizeof($ty:t)*$len
                                              }; |]
        addParam [cparam| $ty:t * $id:n |]
    go fun@(Lam (Named s) f) prelude = do
      t ← compTypeF (argProxy fun)
      i ← freshId
      withAlias i s $ go (f $ varExp i) $ prelude >> addParam [cparam| $ty:t $id
     :s |]

    argProxy :: Signature (b → c) → Proxy b
    argProxy _ = Proxy

    elemProxy :: Ann [b] → Signature ([b] → c) → Proxy b
    elemProxy _ _ = Proxy
```

Listing 1.3. Signature translation

The code generator is defined in listing 1.3. Before explaining how it works, we will explain the code generation technique used.

We use a C code generation monad for producing the C code. Operations of this monad are accessed via the `MonadC` type class. Among other things, it provides a method for generating fresh names, methods for adding statements to the generated code and for adding parameters to the currently generated function definition.

The concrete pieces of C code to be generated are written as actual C code using quasi-quoters [6] for C code, provided by the package `language-c-quote`[3].

For example, consider the following two lines from listing 1.3:

```
17 addParam [cparam| $ty:t *out |]
18 addStm [cstm| *out = $e; |]
```

The first line adds a parameter to the generated C function, and the second line adds a statement that assigns the result to the output pointer. The `[q| ... |]` syntax is for quasi-quotation, where `q` is the name of the quoter. The quoter parses the C code inside the brackets, and turns it into a representation of a piece of code that can be collected in the code generation monad.

Quasi-quoters also allow the splicing of Haskell values into the quoted code. In the above example, `$ty:t` splices in the Haskell value `t` as a C type, and `$e` splices in `e` as a C expression. For the code to type check, `t` must have the type `C.Type` and `e` must have the type `C.Exp`.

The signature is compiled by recursively traversing the `Lam` constructors and building up the argument list. Finally, the `Ret` or `Ptr` case combines the arguments to produce the function signature. The compilation of the function body is delegated to the Feldspar compiler (by calling `compExp`).

The `Lam (Native l)` case (lines 24–38 from listing 1.3) is an example of how the `Signature` language can generate interface code.

```
24     go fun@(Lam n@(Native l) f) prelude = do
25       t ← compTypeF (elemProxy n fun)
26       i ← freshId
27       let w = varExp i
28       C.Var (C.Id m _) _ ← compExp w
29       let n = m ++ "_buf"
30       withAlias i ('&':m) $ go (f w) $ do
31         prelude
32         len ← compExp l
33         addLocal [cdecl| struct array $id:m = { .buffer = $id:n
34                                               , .length=$len
35                                               , .elemSize=sizeof($ty:t)
36                                               , .bytes=sizeof($ty:t)*$len
37                                               }; |]
38         addParam [cparam| $ty:t * $id:n |]
```

[3] http://hackage.haskell.org/package/language-c-quote

Apart from allocating a fresh parameter, it creates a local `struct array` object (lines 33–37) on the function stack and initializes it with the length `l` and the buffer parameter. Then compilation continues with `f` applied to the address of the local `struct array` object.

4 Related Work

MATLAB Coder [7][4] is a tool that generates standalone C and C++ code from MATLAB code. One purpose of MATLAB Coder is to export MATLAB functions to an external system. Since MATLAB is dynamically typed, the same function can operate on values of different type. When generating C code, the user must specify a type for the function, and optionally sizes or size bounds for matrix arguments. This can be done on the command line using what can be seen as a restricted DSL.

However, judging from code examples provided by MathWorks, the signature mapping of MATLAB Coder appears to be rather restricted. For example, stack allocated matrices are passed as two arguments: a pointer to a data buffer and a length vector. If a static size is given for the matrix, the length vector goes away. But if a different argument order is needed, or if one wants to use the same length vector for two different matrices, this likely requires introducing a wrapper function with a different interface.

5 Discussion and Future Work

The `Signature` language enables us to customize the signature of compiled Feldspar functions. It also allows generation of interface code fused with the original function.

Why is a new language needed? Why not just add annotations to the `Lam` abstraction constructor in the Feldspar Core language?

The `Signature` language is a proper extension of the Feldspar Core language, which it means it is optional and can co-exist with other extensions. Since the `Signature` is built using a combination of deep and shallow embedding, the language is possible to extend by the end user. Also, the `Signature` language can be seen as a replacement for the top-level lambda abstractions in the Feldspar expression.

In future work, we will generalize the `Signature` language to work with any expression language that supports the same interface (see section 3.1) as the Feldspar compiler.

[4] Matlab Coder http://www.mathworks.com/products/matlab-coder

Acknowledgements

This research is funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and the Swedish Research Council.

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: Formal Methods and Models for Codesign, MemoCode. IEEE Computer Society (2010).
2. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The Design and Implementation of Feldspar – an Embedded Language for Digital Signal Processing. In: 22nd International Symposium on Implementation and Application of Functional Languages, IFL 2010. Springer (2011).
3. Persson, A.: Towards a Functional Programming Language for Baseband Signal Processing. Thesis for the Degree of Licentiate of Engineering. 1652-876X, (2014).
4. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Trends in Functional Programming. pp. 21–36. Springer Berlin Heidelberg (2013).
5. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. pp. 199–208. ACM, New York, NY, USA (1988).
6. Mainland, G.: Why It's Nice to Be Quoted: Quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell. pp. 73–82. ACM, New York, NY, USA (2007).
7. MathWorks: MATLAB User's Guide (MATLAB Coder), Version: 2011a, http://www.mathworks.com/help/toolbox/coder/index.html.