Generic Monadic Constructs for Embedded Languages

Anders Persson[1,2], Emil Axelsson[1], and Josef Svenningsson[1]

[1] Chalmers University of Technology
{anders.persson,emax,josefs}@chalmers.se
[2] Ericsson

Abstract. We present a library of generic monadic constructs for embedded languages. It is an extension of Syntactic, a Haskell library for defining and processing generic abstract syntax. Until now, Syntactic has been mostly suited to implement languages based on pure, side effect free, expressions. The presented extension allows the pure expressions to also contain controlled side effects, enabling the representation of expressions that rely on destructive updates for efficiency. We demonstrate the usefulness of the extension by giving examples from the embedded language Feldspar which is implemented using Syntactic.

1 Introduction

A *domain specific language* (DSL) is a programming language dedicated to a particular domain. The purpose of a DSL is often to allow a problem or solution to be expressed more clearly and efficiently than a general purpose language would allow. Haskell has a long history of being a host language for embedded DSLs, where the DSL is implemented as a library in the host language [8, 7].

One such language is Feldspar [5, 4]. Feldspar is a domain specific language for programming performance sensitive embedded systems with focus on digital signal processing. The language is a *strongly typed language with pure semantics* and support for both scalar and vector computations. As is common for deeply embedded DSLs, a Feldspar program results in an intermediate representation or *abstract syntax tree* (AST). For code generation purposes, the AST is then translated into some other language, e.g. C or LLVM assembler, suitable for the target device and compiler. Other interpretations of the AST include, but are not limited to, evaluation, static profiling for memory use and pretty printing.

Different embedded languages often contain similar syntactic constructs. For example, most languages have a notion of variable binding, literals, conditional expressions, etc. In addition, most languages have similar functions for semantic interpretation and transformation. Examples of such functions include evaluation and constant folding. To assist the implementation of embedded languages and enable reuse of common implementation aspects, we have developed *Syntactic.*

Syntactic is a Haskell library [6, 3] for defining generic abstract syntax, and a set of utilities for building embedded languages based on this syntax. By using an encoding of data types open to extension, Syntactic provides mechanisms for assembling individual constructs (e.g. literals, tuples, binding, etc) into larger languages. By basing an embedded language on Syntactic, the language becomes modular and open to extension with new syntax elements, front-end combinator libraries and backend interpretation and compilation functions.

1.1 Problem Description

Syntactic works best to implement languages based on pure expressions, i.e computations that are free of side effects. However, this purity makes it impossible to efficiently express algorithms that, for performance, rely on destructive updates of data. The problem manifests itself in both extra memory for storing intermediate results and extra execution time for copying data between intermediate storage locations. A sophisticated compiler might mitigate the problem somewhat by performing lifetime analysis on the data and optimizing the data structures. Still, the compiler can only apply the optimizations when conditions are right and it might be difficult for the programmer to predict when the optimizations will kick in. This is especially true in a large system where the parts are developed individually and conditions vary with the use site of a function. Furthermore, it is desirable to give the programmer more control of when certain values can be overwritten as long as it can be done while preserving the overall referential transparency provided by pure semantics.

As an example, consider the function `iter`. This Feldspar function generates a syntax tree by repeatedly applying the function `f`. Note that the first argument to `iter` is a regular Haskell value which is evaluated as part of the generation. Thus, `iter 4 f` is equivalent to `f ∘ f ∘ f ∘ f`.

```
import Feldspar
import Feldspar.Compiler


iter :: (Type a) ⇒ Length → (Data a → Data a) → Data a → Data a
iter 0 _ = id
iter n f = f ∘ iter (n-1) f

ex :: Data Int32 → Data Int32
ex = iter 4 (λx → x + x)
```

Using the Feldspar Compiler, we can generate the code below by evaluating `icompile ex`. Here, we can see that the result of each function application (`v1`, `v2` and `v3`) is stored in its own location, even though the individual results are not needed later than the next statement. This extra storage becomes more problematic when the results are of a more complex type, e.g structures or arrays, where the cost of storing and copying cannot be neglected. Instead, we want the generated code to eliminate as many storage locations as possible and destructively update the remaining locations.

```
void ex(int32_t v0, int32_t * out)
{
    int32_t v1;
    int32_t v2;
    int32_t v3;

    v1 = (v0 + v0);
    v2 = (v1 + v1);
    v3 = (v2 + v2);
    (* out) = (v3 + v3);
}
```

A well-known method of expressing impure computations in a pure functional language is to use monads [15]. In this paper we present an extension to Syntactic that provides generic monadic constructs for use in deeply embedded languages. In order to add support for destructive updates in languages like Feldspar, we also provide specialized constructs for mutable references and arrays.

1.2 Contributions

This paper makes the following contributions:

- We show how to implement monadic combinators (in particular, `return` and `(>>=)`) as deeply embedded language constructs (section 3). By basing the implementation on the Syntactic framework, we make the constructs available to other languages based on Syntactic. This extension makes Syntactic applicable to a wider range of EDSLs.
- We avoid the common problem of defining a `Monad` instance for the embedded language by using a continuation monad wrapper around the embedded expressions (section 3.2). This allows us to use Haskell's `do`-notation and existing monadic combinators together with DSL programs. To our knowledge, this is a novel technique, and it has a great impact on the usability of the resulting DSLs.
- We show how to implement monadic language constructs, based on available Haskell types, to deal with destructive updates of data (section 4). These constructs give a pragmatic solution to the problem of expressing destructive updates in Feldspar programs.

It is worth noting that all of this was done without having to change the Syntactic framework or the existing Feldspar implementation. This serves as a great example of the modularity provided by Syntactic (see section 2.2).

The code in this paper is based on Syntactic, version 0.8 [3]. The Feldspar code is based on version 0.5 [2, 1].

2 Introduction to Syntactic

When implementing deeply embedded DSLs in Haskell, a syntax tree is typically defined using a (generalized) algebraic data type [7, 4, 12]. As an example, consider a small expression language with support for literals and addition:

```
-- A simple expression langugage
data Expr₁ a where
    Lit₁ :: Num a ⇒ a → Expr₁ a
    Add₁ :: Num a ⇒ Expr₁ a → Expr₁ a → Expr₁ a
```

`Expr₁ a` is parameterized on 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 constrain the type, but in later implementations they will also take care of some tedious wrapping.

The `eval₁` function is just one possible interpretation of the expressions; we can extend the implementation with, say, pretty printing or program analysis. This can even be done without changing any existing code. However, adding a new construct to the language is not so easy. To extend the language with multiplication, we would need to add a constructor to the `Expr₁` type as well as adding a new case to `eval₁` (and other interpretations). Thus, with respect to language extension, a GADT representation of a language is not modular. This limitation is one side of the well-known *expression problem* [14].

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, `Lit₁` and `Add₁` 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 [3] is to provide a basis for such modular languages. The library provides assistance for all aspects of an embedded DSL implementation:

– A generic AST type 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.

For more information about Syntactic, see our lecture notes from the CEFP summer school [6].

```
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)

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

infixl 1 :$
infixr :→

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

Listing 1: Type of generic abstract syntax trees in Syntactic

2.1 Using Syntactic

The idea of the Syntactic library is to express all syntax trees as instances of a very general type `AST`[3], defined in listing 1. `Sym` introduces a constructor 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:

```
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
```

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.

[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.

2.2 Extensible syntax

Part of the reason for using the `AST` type instead of a GADT is that it supports definition of generic traversals [6], which are the basis of the generic interpretation and transformation functions in Syntactic.

Another, equally important, reason for using `AST` is that it opens up for making our syntax trees extensible. We cannot that `Expr`$_2$ is closed in the same way as `Expr`$_1$: Adding a constructor requires changing the definition of `NumDomain`$_2$. However, the `AST` type is compatible with *Data Types à la Carte* [13], which is a technique for encoding open data types in Haskell.[4]

The idea is to create domains as co-products of smaller independent domains using the `(:+:)` type operator from Syntactic. To demonstrate the idea, we split `NumDomain`$_2$ into sub-domains and combine them into `NumDomain`$_3$, used to define `Expr`$_3$. The new type `Expr`$_3$ is again isomorphic to `Expr`$_1$.

```
data Lit3 a where Lit3 :: Num a ⇒ a → Lit3 (Full a)
data Add3 a where Add3 :: Num a ⇒ Add3 (a :→ a :→ Full a)

type NumDomain3 = Lit3 :+: Add3

type Expr3 a = ASTF NumDomain3 a
```

To get extensible syntax we cannot use a closed domain, such as `NumDomain`$_3$, but instead use constrained polymorphism to abstract away from the exact shape of the domain. The standard way of doing this in 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, Num a) ⇒ a → ASTF dom a
lit3 a = Sym (inj (Lit3 a))

add3 :: (Add3 :<: dom, Num a, Typeable a)
     ⇒ ASTF dom a → ASTF dom a → ASTF dom a
add3 x y = Sym (inj Add3) :$ x :$ y
```

The definition of smart constructors can be automated with `appSym` from Syntactic. The following definitions of `lit`$_3$ and `add`$_3$ are equivalent to the ones above:

```
lit3 = appSym ∘ Lit3
add3 = appSym Add3
```

A constraint `(Lit`$_3$ `:<: dom)` can be read as "`dom` contains `Lit`$_3$". That is, `dom` should be a co-product chain of the general form `(... :+: Lit`$_3$ `:+: ...)`.

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 be in separate modules. Any number of additional constructs can be added in a similar way.

[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.

2.3 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:

```
newtype Expr4 a = Expr4 {unExpr4 :: ASTF NumDomain3 a}
```

Secondly, it is sometimes desirable to use more "high-level" or domain specific representations as long as these representations have a correspondence to an AST.

```
data Pair a where
  Pair :: Pair (a :→ b :→ Full (a,b))
data Select a where
  Sel1 :: Select ((a,b) :→ Full a)
  Sel2 :: Select ((a,b) :→ Full b)
```

Such high-level types are referred to as "syntactic sugar". Syntactic sugar is defined by the class below.

```
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
```

In the Syntactic class the associated type Internal is a type function from the (sugared) user visible type a to its internal representation in the AST. 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.

As a simple example, the instances for $\texttt{Expr}_4$ and (,) would look as follows:

```
instance Typeable a ⇒ Syntactic (Expr4 a) NumDomain3 where
    type Internal (Expr4 a) = a
    desugar = unExpr4
    sugar   = Expr4

instance (Pair :<: dom, Select :<: dom, Syntactic a dom, Syntactic b dom) ⇒
  Syntactic (a,b) dom where
    type Internal (a,b) = (Internal a, Internal b)
    desugar = uncurry $ sugarSym Pair
    sugar a = (sugarSym Sel1 a, sugarSym Sel2 a)
```

The sugarSym function from Syntactic extends the appSym function with support for syntactic sugar. Thus, the following declarations are equivalent:

```
pair1 a b = sugarSym Pair a b
pair2 a b = sugar $ appSym Pair (desugar a) (desugar b)
```

3 Monads in Syntactic

Monads, popularized by Wadler [15], provide ways of adding impure computations to languages with otherwise pure semantics. Such impure computations – effects – are desired when implementing functions that for performance require destructive updates. In this section we will show how to add an embedding of monadic computations to Syntactic.

As with other constructs in Syntactic, the monad embedding consists of two parts: a deep embedding representing the core syntax terms, and a set of library functions providing the programmer interface. To enable the use of Haskell's `do`-notation and existing monadic combinators together with DSL programs, the programmer interface is built on top of the Haskell continuation monad `Cont` (see section 3.2).

3.1 Deep Embedding

Monads in Haskell are types that are members of the `Monad` type class.

```
class Monad m where
    return :: a   → m a
    (>>=)  :: m a → (a → m b) → m b
    (>>)   :: m a → m b → m b
    fail   :: String → m a
```

This type class can be represented in Syntactic by the GADT below.[5]

```
data MONAD m a where
    Return :: MONAD m (a   :→ Full (m a))
    Bind   :: MONAD m (m a :→ (a → m b) :→ Full (m b))
    Then   :: MONAD m (m a :→ m b        :→ Full (m b))
```

The constructors are parameterized over the embedded monad `m` so that `MONAD` can represent different monads. An example of one monad is given in section 4.

As usual, we use `appSym` to turn the monad symbols into `AST` constructors.

```
ret = appSym Return
thn = appSym Then
```

However, `Bind` requires special care. A straightforward definition using `appSym` does not give us a very satisfying type:

```
bnd :: (MONAD m :<: dom, Typeable1 m, Typeable a, Typeable b)
    ⇒ ASTF dom (m a) → ASTF dom (a → m b) → ASTF dom (m b)
bnd = appSym Bind
```

The problem is the second argument. This is an expression that computes a function – but we do not have a way to construct such expressions. The solution chosen in Syntactic is to use *higher-order abstract syntax* (HOAS) [11] as a means to embed functions in a syntax tree, see listing 2.

[5] We omit the method `fail`, since we do not want to transfer its functionality to the embedded language. Exceptions can be implemented separately using the monadic support presented in this paper.

```
data HOLambda ctx dom a where
    HOLambda :: (Typeable a, Typeable b, Sat ctx a)
             ⇒ (ASTF (HODomain ctx dom) a → ASTF (HODomain ctx dom) b)
             → HOLambda ctx dom (Full (a → b))

type HODomain ctx dom = HOLambda ctx dom :+: Variable ctx :+: dom

lambda :: (Typeable a, Typeable b, Sat ctx a)
       ⇒ (ASTF (HODomain ctx dom) a → ASTF (HODomain ctx dom) b)
       → ASTF (HODomain ctx dom) (a → b)
lambda = appSym ∘ HOLambda

class Sat ctx a where
    data Witness ctx a
    witness :: Witness ctx a

instance ( Syntactic a (HODomain ctx dom), Syntactic b (HODomain ctx dom)
         , Sat ctx (Internal a)) ⇒
    Syntactic (a → b) (HODomain ctx dom) where
        type Internal (a → b) = Internal a → Internal b
        desugar f = lambda (desugar ∘ f ∘ sugar)
```

Listing 2: Higher-Order Abstract Syntax in Syntactic

To make a smart constructor based on HOAS for `Bind`, we first have to wrap the function argument with the combinator `lambda`:

```
bnd :: (MONAD m :<: dom, Typeable a, Typeable b, Typeable1 m, Sat ctx a)
    ⇒ ASTF (HODomain ctx dom) (m a)
    → (ASTF (HODomain ctx dom) a → ASTF (HODomain ctx dom) (m b))
    → ASTF (HODomain ctx dom) (m b)
bnd k f = appSym Bind k (lambda f)
```

Note that the second argument of `bnd` is an ordinary Haskell function from `AST` to `AST`, which leads to a type much closer to that of the ordinary `(>>=)` operator. Using the `Syntactic (a → b)` instance `bnd` can be expressed as `bnd = sugarSym Bind`.

The `Sat` class is used to parameterize over class constraints by associating them with a type `ctx`. This technique is inspired by restricted data types [9]. While the `Sat` class is not important in the implementation of monads, it is included in this paper to complete the presentation of HOAS in Syntactic.

The smart constructors – `ret`, `thn` and `bnd` – can be used to implement generic monad combinators like for example `liftM`.

```
-- Haskell implementation of liftM
liftM :: (Monad m) ⇒ (a → r) → m a → m r
liftM f m = do { x ← m; return (f x) }

-- Syntactic (deep) implementation of liftM
liftM_deep :: (MONAD m :<: dom, Typeable1 m, Typeable a, Typeable r, Sat ctx a)
           ⇒ (ASTF (HODomain ctx dom) a → ASTF (HODomain ctx dom) r)
           → ASTF (HODomain ctx dom) (m a) → ASTF (HODomain ctx dom) (m r)
liftM_deep f m = m ‘bnd‘ λx → ret (f x)
```

Note how the use of higher-order syntax allows us to use an ordinary λ-abstraction to construct the continuation argument to `bnd`.

It would of course be nice if we could do without `liftM`$_{deep}$ and just use `liftM` for embedded programs. However, this is not directly possible, since we cannot make `AST` an instance of the `Monad` class. In the following sections, we will give a solution to this problem.

3.2 User Interface

For the monad extension, we wanted to use Haskell's existing `Monad` class as the interface. A `Monad` instance is valuable since it gives access to a wealth of combinators for monadic expressions and to the convenient `do`-notation.

A naive, but incorrect, implementation of the `Monad` instance is:

```
newtype Mon ctx dom m a = Mon { unMon :: ASTF (HODomain ctx dom) (m a) }

-- Incorrect implementation
instance (MONAD m :<: dom) ⇒ Monad (Mon ctx dom m) where
    return a = Mon $ ret a
    ma >>= f = Mon $ unMon ma ‘bnd‘ (unMon ∘ f)
    ma >> mb = Mon $ unMon ma ‘thn‘ unMon mb
```

However, that instance does not type-check for any of the methods. Consider `(>>=)` in Haskell, which has the type `Monad m ⇒ m a → (a → m b) → m b`. That signature is incompatible with the signature of the suggested implementation:

```
(MONAD m :<: dom, Sat ctx a, Typeable1 m, Typeable a, Typeable b)
⇒ Mon ctx dom m a
→ (ASTF (HODomain ctx dom) a → Mon ctx dom m b)
→ Mon ctx dom m b
```

This signature has two problems: (1) It constrains the type variables `a` and `b`, which are required to be parametrically polymorphic by `(>>=)`, and (2) the value passed to the continuation has type `ASTF ... a` rather than just `a`, which would be required by `(>>=)`.

A way around this problem would be to use a restricted monad [9]. However, such a solution would be incompatible with existing monad libraries in Haskell. Furthermore, it is not possible to use `do`-notation with restricted monads without the pervasive language extension `RebindableSyntax`.

3.3 Continuation Monad Wrapper

To work around the first problem above and make the parameter `a` polymorphic again, we introduce a continuation monad wrapper around the `AST`.

```
newtype MS ctx dom m a = MS { unMS :: forall r. Typeable r
                                   ⇒ Cont (ASTF (HODomain ctx dom) (m r)) a }
```

Here the Haskell Continuation Monad holds expressions that will eventually result in a syntax tree over the embedded monad `m`.

```
instance ( MONAD m :<: dom, Typeable1 m, Sat ctx (Internal a)
         , Syntactic a (HODomain ctx dom)
         ) ⇒
    Syntactic (MS ctx dom m a) (HODomain ctx dom) where
        type Internal (MS ctx dom m a) = m (Internal a)
        desugar a = runCont (unMS a) (sugarSym Return)
        sugar   a = MS $ cont (sugarSym Bind a)
```

Listing 3: Syntactic sugar for monads

The parameter `a` is polymorphic since the constraints imposed by the `AST` type only affect the result `r` of the continuation monad. By encapsulating the continuation monad in a **`newtype`** we can make the type parameter `r` universally quantified and prevent it from leaking out to clients.

The `Monad` instance can be written as usual for a **`newtype`**.

```
instance Monad (MS ctx dom m) where
    return a = MS $ return a
    ma >>= f = MS $ unMS ma >>= unMS ∘ f
```

There is still one piece of the puzzle missing. While the `Monad` instance gives us access the `do`-notation and monadic combinators, it is just a continuation monad. However, since the side effect of our `MS` monad is to build an `AST`, we need to convert each action into its corresponding syntax element. The syntactic sugar in the next section will automate the process.

3.4 Monads as Syntactic Sugar

With the `Syntactic` instance in listing 3 we can convert between monadic computations and syntax trees.

The function `desugar` can build a syntax tree from an expression in the `MS` monad by running the contuation-passing computation and returns the resulting tree using `sugarSym Return` as the final continuation.

To construct a continuation-passing computation, `sugar` binds the result of previous computations using `sugarSym Bind`.

As an illustration of how the continuation monad and the `Syntactic` instance work together, we consider the following reduction of the expression

```
λx → desugar $ MS $ return x
```

Inline `desugar` from the `Syntactic (MS ...)` instance

```
λx → runCont (unMS (MS $ return x)) (sugarSym Return)
```

Simplify `unMS/MS` and inline `runCont`

```
λx → sugarSym Return x
```

Note that `runCont` applied the final continuation to the result of the computation `return x`. We continue the reduction by inlining the definition of `sugarSym`.

```
λx → sugar $ appSym Return (desugar x)
```

Recall that for `Syntactic (ASTF dom a)` the definition of `sugar` is `id`.

```
λx → appSym Return (desugar x)
```

By inlining the definition of `appSym` the reduction is complete

```
λx → Sym (inj Return) :$ desugar x
```

Further examples of the sugared syntax will be shown in the context of the `Mutable` monad in section 4.

4 Mutable Monad

With the monad feature added to Syntactic, it is now possible to extend it further and embed specific monads into the `AST` type. To address our problem with destructive updates we choose to implement a subset of the `IO` monad from Haskell and the accompanying `Data.IORef` representing *mutable references* in the `IO` monad. It is also possible to represent the mutable monad using the *state transformer monad* from `Control.Monad.ST` and mutable references using `Data.STRef`. The run function (`runST :: (forall s. ST s a) → a`) of the `ST` monad has a universally quantified parameter `s` ensuring that the state does not leak out of the monad, which increases safety. However, that implementation is slightly more verbose and `IO` is sufficient to show mutability for this paper.

We add two constructs to our domain:

- `Mutable` represents a run function that will evaluate the mutations in sequence, returning the result as a pure value.
- `MutableRef` provides initialization, query and update of mutable references.

```
data Mutable a where
    Run    :: Mutable (IO a :→ Full a)

data MutableRef a where
    NewRef :: MutableRef (a :→ Full (IO (IORef a)))
    GetRef :: MutableRef (IORef a :→ Full (IO a))
    SetRef :: MutableRef (IORef a :→ a :→ Full (IO ()))

type MyDomain6 = MONAD IO :+: Mutable :+: MutableRef :+: NumDomain3
```

Then we define a monad `M` for mutable updates as a specialisation of the `MS` type and close the language under `Data` to make type signatures nicer.

```
type M a = MS Poly MyDomain6 IO a

newtype Data a = D { unD :: ASTF (HODomain Poly MyDomain6) a }

instance Typeable a ⇒ Syntactic (Data a) (HODomain Poly MyDomain6) where
    type Internal (Data a) = a
    desugar = unD
    sugar   = D
```

The context `Poly` is defined in `Syntactic` and denotes a fully polymorphic constraint.

Equipped with the `Syntactic` instance in listing 3, we create smart constructors and combinators with friendly types.

```
runMutable₆ :: (Typeable a) ⇒ M (Data a) → Data a
runMutable₆ = sugarSym Run

newRef₆ :: (Typeable a) ⇒ Data a → M (Data (IORef a))
newRef₆ = sugarSym NewRef

getRef₆ :: (Typeable a) ⇒ Data (IORef a) → M (Data a)
getRef₆ = sugarSym GetRef

setRef₆ :: (Typeable a) ⇒ Data (IORef a) → Data a → M (Data ())
setRef₆ = sugarSym SetRef

modifyRef₆ :: (Typeable a) ⇒ Data (IORef a) → (Data a → Data a) → M (Data ())
modifyRef₆ r f = getRef₆ r >>= setRef₆ r ∘ f
```

We can now rewrite the example function `iter` from the introduction, making use of mutable references to provide destructive updates. Note, to be able to use the Feldspar compiler, the example below is written using types and functions from Feldspar.

```
iter₆ :: (Type a) ⇒ Length → (Data a → Data a) → Data a → Data a
iter₆ n f i = runMutable $ do
    r ← newRef i
    go n r
    getRef r
  where
    go 0 _ = return ()
    go j r = modifyRef r f >> go (j-1) r

ex₆ :: Data Int32 → Data Int32
ex₆ = iter₆ 4 (λx → x + x)
```

Again, using the Feldspar Compiler we can generate the code by evaluating `icompile ex₆`. By virtue of the mutable references the calculation is now done in-place without the need for storing intermediate values.

```
void ex₆(int32_t v0, int32_t * out)
{
    int32_t e0;

    e0 = v0;
    e0 = (e0 + e0);
    e0 = (e0 + e0);
    e0 = (e0 + e0);
    (* out) = (e0 + e0);
}
```

5 Conclusion

We have shown an extension to the Syntactic library, enabling the representation of monadic computations in the generic `AST` type. Using the monad together with the constructs `Mutable` and `MutableRef`, we can model expressions with mutable references. By embedding the monads in a continuation monad we avoid the problem of creating a `Monad` instance, unlocking access to Haskell `do`-notation and monadic combinators. That the extensions were possible without changing the underlying Syntactic library or the target Feldspar language is a showcase of the modularity of these languages.

The monadic constructs and combinators in this paper are not specific to one language, but reusable by any language built using Syntactic, making Syntactic applicable for a wider range of EDSLs.

The monadic constructs in Syntactic can be reused to implement other monads and interfaces. The mutable arrays in Feldspar are similar to the mutable references presented in this paper.

We have also extended Feldspar with an experimental implementation of the `Par` monad from `Control.Monad.Par`, which is a monad for *deterministic parallelism* [10]. Future work includes studying the integration of different scheduling algorithms and code generation.

6 Related Work

To the best of our knowledge, the technique of embedding a monad syntacticly into an extensible EDSL presented in this paper is a novel technique. While others have embedded monads into the continuation monad, the technique has not been used in conjunction with Data Types à la Carte to define EDSLs.

Swierstra shows how to implement free monads on top of his Data Types à la Carte [13]. By limiting the available operations, the available side effects can be controlled at the type level. Since our work builds on Data Types à la Carte, we get the same control over available side effects. However, Swierstra's monad is not applicable to deeply embedded DSLs, as it does not include a deep embedding of the monadic combinators `return` and `(>>=)`.

Hughes develops *restricted monads* in [9]. However, he does not create a proper `Monad` instance, but instead a `WfMonad` class which is parameterized on a dictionary. This makes the method incompatible with Haskell's `do`-notation and existing monad combinators.

Acknowledgements

This research is funded by Ericsson, Vetenskapsrådet, and the Swedish Foundation for Strategic Research. The Feldspar project is an initiative of and is partially funded by Ericsson Software Research and is a collaboration between Chalmers, Ericsson and ELTE University.

References

[1] Feldspar compiler: http://hackage.haskell.org/package/feldspar-compiler

[2] Feldspar language: http://hackage.haskell.org/package/feldspar-language

[3] Syntactic library: http://hackage.haskell.org/package/syntactic

[4] 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, IFL 2010. LNCS, vol. 6647 (2011)

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

[6] Axelsson, E., Sheeran, M.: Feldspar: Application and implementation. In: Lecture Notes of the Central European Functional Programming School (to appear), LNCS, vol. 7241 (2012)

[7] Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming 13:3, 455– 481 (2003)

[8] Hudak, P.: Modular domain specific languages and tools. In: ICSR '98: Proceedings of the 5th International Conference on Software Reuse. p. 134. IEEE Computer Society, Washington, DC, USA (1998)

[9] Hughes, J.: Restricted data types in Haskell. In: Proceedings of the 1999 Haskell Workshop (1999)

[10] Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. In: Proceedings of the 4th ACM symposium on Haskell. pp. 71–82. Haskell '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2034675.2034685

[11] Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. pp. 199–208. PLDI '88, ACM (1988)

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

[13] Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (2008)

[14] Wadler, P.: The expression problem. http://www.daimi.au.dk/~madst/tool/papers/expression.txt (1998)

[15] Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM conference on LISP and functional programming. pp. 61–78. LFP '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/91556.91592