On Adding Pattern Matching to Haskell-based Deeply Embedded Domain Specific Languages

Not For Distribution

Anonymous Author(s)

Abstract

Capturing control flow is the Achilles heel of Haskell-based deeply embedded domain specific languages. Rather than use the builtin control flow mechanisms, artificial control flow combinators are used instead.

However, capturing traditional control flow in an deeply embedded domain specific language would support the writing of programs in a natural style by allowing the programmer to use the constructs that are already builtin to the base language, such as pattern matching and recursion.

In this paper, we expand the capabilities of traditional, Haskell-based, deep embeddings with a compiler extension for reifying conditionals and pattern matching. With this new support, the Haskell that we capture for expressing deeply embedded domain specific languages can be cleaner, Haskell-idiomatic, and more declarative in nature.

CCS Concepts • **Software and its engineering** → *Translator writing systems and compiler generators*; *Source code generation*; **Domain specific languages**;

ACM Reference Format:

Anonymous Author(s). 2020. On Adding Pattern Matching to Haskell-based Deeply Embedded Domain Specific Languages Not For Distribution. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20) November 15 – 20, 2020, Chicago, Illinois, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/1

1 Introduction

Embedded domain specific languages (EDSLs) have long been an effective technique for constructing reusable tools for working in a variety of different problem domains. Haskell is a language which is particularly well-suited to EDSLs due to its lazy evaluation, first-class functions and lexical closures.

Despite these advantages Haskell provides for creating EDSLs, there are a few constructs for which representations have proven illusive. One prominent example is that of a `case` expression. Pattern matching is a convenient way to implement control flow structures and to inspect data structures, so it is frequently a desirable feature for many EDSLs.

Additionally, lambdas can also be difficult to implement in EDSLs. A major reason for this is that control flow constructs which are typically *outside* the EDSL, such as pattern matching and tail recursion, can make it challenging to "look inside" the lambda. When these control structures are reified within the EDSL, it is easier to also reify lambdas, as you can simply apply them to a dummy argument to access their body (with the dummy value substituted for the argument).

> **TODO**: Maybe add brief summary of relationship to lambdas?

The following contributions are made by this paper:

- A representation of pattern matching in a Haskell EDSL (Section 3)
- A representation of lambdas (Section 5) in the EDSL, taking advantage of the fact that both pattern matching and tail recursion are already captured in the EDSL.
- A GHC Core plugin which transforms a subset of standard Haskell code into this EDSL for pattern matching, extended with tail recursion, lambdas and primitive operations (Section 8)
- An implementation of a interpreter for the "standard" semantics of the EDSL (Section 6).
- An outline of a C backend for this EDSL (Section 7).

Readers who are primarily interested only in the encoding of pattern matching itself can direct most of their attention to Sections 2, 3 and 6. Most of the technical aspects of the transformation performed by the Core plugin are described in Section 8.

1.1 Motivation

> **TODO**: Expand?

A major benefit of the Haskell language is that it is frequently possible derive a Haskell implementation from a

GPCE '20, November 15 – 20, 2020, Chicago, Illinois, United States

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00

https://doi.org/1

specification, either partially or fully. Examples of this technique include the work in [Elliott 2018] as well as the work in [Elliott 2019]. When it is not possible to fully derive an implementation from a specification, it is often easier to check a Haskell implementation against a specification than it is in many imperative languages due, in part, to the management of side effects in Haskell. This is more difficult to do using an imperative language such as C, as they provide no restrictions on where side effects can occur.

Haskell EDSLs that support powerful language features like pattern matching allow these advantages to be brought to problem domains where it is currently either difficult or impossible to use a language that has these benefits.

> **TODO**: Cite examples where Haskell implementations have been derived specifications. Conal Elliott's latest two papers (as of May 2020), [Elliott 2018] and [Elliott 2019], would be good examples of this. (This is now cited)
> A paper which describes a Haskell implementation derived from an operational semantics would also be a good example.

1.2 Overview

In this EDSL, the type representing values in the expression language is `E t` and there are two basic functions which translate values to and from this expression language with the following type signatures (omitting type class constraints for the moment):

```
rep :: a -> E a
abs :: E a -> a
```

Additionally, the following functions mark the parts of the code which the Core plugin will target:

```
internalize :: E a -> a
externalize :: a -> E a
```

The `E` type is the type of expressions in the EDSL. In this paper, the data constructors of this type will be described incrementally as they are needed.

1.3 Translation Example

> **TODO**: Does it make sense for this to be a subsection of the intro?

In the example below, the Core plugin transforms `example` into `example'`:

```
import Data.Char (ord)

x :: Either Char Int
x = Left 'a'

example :: Int
example =
  internalize (externalize
    (case x of
      Left  c -> ord c
      Right i -> i))

example' :: Int
example' =
  abs (CaseExp
        (LeftExp 'a')
        (SumMatchExp (OneProdMatch
                       (Lam 1
                         (Ord (Var 1))))
                     (OneSumMatch
                       (OneProdMatch
                         (Lam 2
                           (Var 2))))))
```

2 Representing Algebraic Datatypes

> **TODO**: Look into `generic-sop` and compare then cite corresponding paper.

Briefly, a algebraic type `T` with an automatically generated `ERep` instance is given a representation in terms of `Either`, `(,)`, `()` and `Void`. This "standardized" type representation is given by `ERepTy T`. `ERepTy` is an type family associated to the type class `ERep`. This type will be called the *canonical type* of `T`.

Only the "outermost" type will be deconstructed into these fundamental building blocks, and further deconstruction can take place on these pieces later on. For example, consider this type:

```
data ComplexPair where
  ComplexPair :: Complex Double
              -> Complex Double
              -> ComplexPair
  deriving (Generic, Show)

instance ERep ComplexPair
```

Note that the instance definition is automatically generated from the `Generic` instance.

Given this code, `ERepTy ComplexPair ~ (Complex Double, Complex Double)`. Here is an example that demonstrates why this is more useful than if it *fully* deconstructed each type into `Either`, `(,)` and `()`:

```
sumComplexPair ::
  ComplexPair -> Complex Double
sumComplexPair p =
```

```
  internalize (externalize
    (case p of
      ComplexPair a b -> a + b))
```

If ERepTy ComplexPair were fully deconstructed into ((Double, Double), (Double, Double)), we would need a Num instance for (Double, Double). What we really want is to use the fact that a Num instance already exists for Complex Double. This is exactly what preserving this type information allows us to do. We can later use this preserved type information in backends, through the corresponding Typeable instances (for instance, to provide special support for arithmetic on complex numbers).

It is important to note that we are still able to further deconstruct this type with further pattern matches, since ERepTy (Complex Double) ~ (Double, Double):

```
realSum :: ComplexPair -> Double
realSum p =
  internalize (externalize
    (case p of
      ComplexPair a b ->
        case a of
          a_real :+ _ ->
            case b of
              b_real :+ _ ->
                a_real + b_real))
```

Note that the above pattern matches can be written as a single nested pattern match. Both forms compile down to the same Core representation.

An additional benefit of this is that recursive types require no special handling. For example, consider:

```
data IntList = Nil | Cons Int IntList
  deriving (Generic, Show)

instance ERep IntList
```

Note that ERepTy IntList ~ Either () (Int, IntList). If ERepTy attempted to "fully deconstruct" IntList, it would send the compiler into an infinite loop.

This allows us to implement functions on such recursive types:

```
isEmpty :: IntList -> Bool
isEmpty t =
  internalize (externalize
    (case t of
      Nil -> True
      Cons x xs -> False))

intListSum :: (Int, IntList) -> Int
intListSum p =
  internalize (externalize
    (case p of
      (acc, t) ->
        case t of
          Nil -> acc
          Cons x xs ->
            intListSum
              (x+acc, xs)))
```

2.1 E, ERep, ERepTy

There are three interconnected foundational parts: E, ERep and ERepTy. E is the deep embedding of the EDSL (a GADT that encodes expressions in the DSL language). ERep is a type class which represents all Haskell types which can be represented in the DSL. ERepTy is a type family associated to the ERep type class, which represents a "canonical form" of the given type. This canonical form can be immediately constructed in the EDSL. Canonical form types crucially include Either and (,), which allow all combinations of basic sum types and product types to be encoded, in the manner described at the beginning of Section 2.

With GHC Generics, any data type T with a Generic instance has a corresponding Rep T type, which gives a generic representation of T. Conversion between values of this type and values of the original type T is given by the functions to and from. The generic representation Rep T can be traversed by functions which operate solely on the structure of the data type. This generic representation contains additional metadata which we do not need. However, we can automatically generate a ERep instance for any type which has a Generic instance.

As Generic instances are automatically generated, this provides a simple mechanism to automaticly generate ERep instances.

This information is brought into the E type via the constructor ConstructRep. The E also contains constructors representing Either and (,) values:

```
data E t where
  ...
  ConstructRep ::
    (Typeable a, ERep a)
    => E (ERepTy a) -> E a

  LeftExp :: E a -> E (Either a b)
  RightExp :: E b -> E (Either a b)

  PairExp :: E a -> E b -> E (a, b)
  ...
```

ERep and ERepTy provide an interface for transferring values between the EDSL expression language and the source Haskell language:

```
class Typeable t => ERep t where
```

```haskell
  type ERepTy t

  construct :: t -> E (ERepTy t)

  rep :: t -> E t

  default rep ::
    (ERep (ERepTy t))
    => t -> E t
  rep x = ConstructRep (construct x)
  {-# INLINABLE rep #-}

  unrep' :: ERepTy t -> t

  rep' :: t -> ERepTy t
```

The key algebraic instances mentioned before are as follows:

```haskell
instance (ERep a, ERep b)
  => ERep (Either a b) where

  type ERepTy (Either a b) = Either a b

  rep (Left x)  = LeftExp (rep x)
  rep (Right y) = RightExp (rep y)

  unrep' = id
  rep'   = id

instance (ERep a, ERep b)
  => ERep (a, b) where

  type ERepTy (a, b) = (a, b)

  rep (x, y) = PairExp (rep x) (rep y)
  unrep' = id
  rep'   = id
```

3 Representing pattern matches

Within the E expression language, a pattern match is represented by the CaseExp constructor:

```haskell
data E t where
  ...
  CaseExp ::
    (ERep t, ERepTy (ERepTy t) ~ ERepTy t)
      => ERep t
         -> E (SumMatch (ERepTy t) r)
         -> E r
  ...
```

The equality constraint ensures that a canonical type is its own canonical type.

> **TODO**: Should we make a synonym for this constraint since it is used in multiple places? Like this:
>
> ```haskell
> type CanonicalIsIdem t =
>   ERepTy (ERepTy t) ~ ERepTy t
> ```

(The above code block is a slight simplification of the actual implementation, which has an additional type variable which is used at an "intermediate" point. The expanded form is in place to simplify the Core transformation.)

The type `SumMatch` is defined as

```haskell
newtype SumMatch a b =
  MkSumMatch { runSumMatch :: a -> b }
```

For the moment, we will primarily use this type as a type tag and ignore the values it can take on.

A value of type `E (SumMatch a b)` represents a computation within the EDSL which destructures a value of type `E a` and produces a value of type `E b`. Therefore, `E (SumMatch (ERepTy t) r)` represents a computation which destructures a value of type `E (ERepTy t)` and produces a value of type `E b`.

3.1 E (SumMatch a b)

The overall structure of a `E (SumMatch a b)` value is a (heterogeneous) list of `E (ProdMatch x b)` values. Each item of this list corresponds exactly to one branch in the original case match.

The following constructors generate `SumMatch`-tagged values in the expression language:

```haskell
data E t where
  ...
  SumMatchExp ::
    (ERep a, ERep b, ERepTy b ~ b)
      => E (ProdMatch a r)
         -> E (SumMatch b r)
         -> E (SumMatch (Either a b) r)

  OneSumMatch ::
    (ERep a, ERep b, ERepTy a ~ a)
      => E (ProdMatch a b)
         -> E (SumMatch a b)


  EmptyMatch ::
    (ERep b) => E (SumMatch Void b)
  ...
```

Note the `ERepTy a ~ a` constraints. This constraint ensures that the type `a` is already in canonical form (that is, consists entirely of `Either`, `(,)` and base types).

3.2 `E (ProdMatch s t)`

`E (ProdMatch x y)` is equivalent to a curried function from `x` to `y` in the expression language . Note that `s` is a (potentially nested) pair type. For example, `E (ProdMatch (a, (b, c)) r)` is equivalent to `E (a -> E (b -> E (c -> E r)))`.

```
data E t where
  ...
  ProdMatchExp ::
    (ERep a, ERep b)
      => E (a -> ProdMatch b r)
         -> E (ProdMatch (a, b) r)

  NullaryMatch ::
    (ERep a)
      => E r -> E (ProdMatch a r)

  OneProdMatch ::
    (ERep a)
      => E (a -> b) -> E (ProdMatch a b)
  ...
```

The `ProdMatch` type is defined similarly to the `SumMatch` type and is similiarly used primarily as a type tag:

```
newtype ProdMatch a b =
  MkProdMatch { runProdMatch :: a -> b }
```

3.3 Connection to Church encodings

> **TODO**: Work on this subsection. Maybe this shouldn't be a separate section at all and maybe its contents should be incorporated throughout the rest of the paper.

Algebraic datatypes can be represented as lambda calculus terms using *Church encodings*. For example, the pair constructor can be represented as

```
church_mkPair
  :: a -> b -> (a -> b -> r) -> r
church_mkPair x y = \ f -> f x y
```

Coproducts can be represented with the constructors

```
church_mkLeft
  :: a -> (a -> r) -> (b -> r) -> r
church_mkLeft x = \ f g -> f x

church_mkRight
  :: b -> (a -> r) -> (b -> r) -> r
church_mkRight y = \ f g -> g y
```

In this representation:

- A pair of type `A` and type `B` is represented with a value of type `forall r. (A -> B -> r) -> r`.
- A coproduct of type `A` and type `B` is represented with a value of type `forall r. (A -> r) -> (B -> r) -> r`.

Both of those representations represent the type as a function that performs the corresponding case analysis. A deep embedding of the operations `churchMkPair`, `church_mkLeft` and `church_mkRight` is given by the type

```
data Church r where
  ChurchPair
    :: a -> b -> (a -> b -> r) ->
       Church r
  ChurchLeft
    :: a -> (a -> r) -> (b -> r) ->
       Church r
  ChurchRight
    :: b -> (a -> r) -> (b -> r) ->
       Church r
```

with the evaluation function

```
churchEval :: Church r -> r
churchEval (ChurchPair x y f) = f x y
churchEval (ChurchLeft x f g) = f x
churchEval (ChurchRight y f g) = g y
```

However, we only want to keep track of the functions used for elimination. By moving the values actually being contained in the products and coproducts out to the evaluation function, we arrive at

```
data Church' t r where
  ChurchPair'
    :: (a -> b -> r) ->
       Church' (a, b) r

  ChurchEither'
    :: (a -> r) -> (b -> r) ->
       Church' (Either a b) r

churchEval' :: Church' t r -> t -> r
churchEval' (ChurchPair' f) (x, y) = f x y
churchEval' (ChurchEither' f g) e =
  case e of
    Left  x -> f x
    Right y -> g y
```

> **TODO**: Figure out how recursive types fit into this framework. Church encodings can handle recursive types and so can our representation. This is also the case for Scott encodings, etc, and is the difference between Church and Scott encodings.
> Does our way of handling recursive types correspond to how any of those encodings handle recursive types?
> Maybe this is relevant: Generalized Church encoding is the Curry-Howard of Knaster-Tarski

3.4 An aside on ProdMatch and SumMatch values

Though ProdMatch and SumMatch are used throughout this EDSL as type tags, they do have values and they are not trivial values. The reason for this is that it connects the encoded pattern matches (values from the E) type to their semantics. A value of type E (SumMatch a b) is an expression which takes in a value of type a (embedded within the expression language), internally performs some pattern matching, and produces a value of type b (again, embedded within the expression language). This is exactly the semantics of a function from a to b. Likewise for ProdMatch a b.

Recall the previously mentioned function

```
abs :: E a -> a
```

Now consider at the type of abs when it is specialized to take E (SumMatch a b) values:

```
abs :: E (SumMatch a b) -> SumMatch a b
```

If we postcompose with runSumMatch, we get:

```
runSumMatch . abs
  :: E (SumMatch a b) -> (a -> b)
```

The SumMatch a b value which abs returns is exactly the function which pattern matches according to its input value. Likewise for ProdMatch a b values.

4 Representing tail recursion

Tail recursion is given a direct-style representation using a simple sum type, using the technique described in [Grebe et al. 2017].

Consider a tail recursive function of the type f :: a -> b. Each recursive call can be seen as a simple "update" of the values of the arguments, since these calls are all in tail position. This is why tail recursive functions can be easily compiled to simple loops or conditional jumps.

We can take advantage of this view by transforming a tail recursive function f :: a -> b into a function f' :: a -> Iter b a. Iter b a is a type which can either correspond to an argument "update" (in the sense mentioned previously) or a final result. This type is implemented as a sum of the types a and b. Recursive calls are transformed to Step applications and non-recursive branches are wrapped in Done applications.

```
data Iter a b = Step b | Done a
  deriving (Functor, Generic)
```

To use the new f' function, we repeatedly call it until it gives a Done value. If it gives a Step value, we pass the the value wrapped in the Step back into f' and continue.

The function runIter provides the standard semantics for executing such a function representing a tail recursive function:

```
runIter :: (ERep a, ERep b)
  => (a -> Iter b a)
     -> (a -> b)
runIter f = go
  where
    go x =
      case f x of
        Done r  -> r
        Step x' -> go x'
```

> **TODO**: runIter has the following alternate implementation. Would this be useful to use somewhere?
>
> ```
> runIter f = getDone . fix (f >=>)
>
> getDone :: Iter b a -> b
> getDone (Done x) = x
> getDone _ = error "getDone"
> ```
>
> where Iter is given a standard Monad instance (essentially the same as the Either Monad instance).
>
> Maybe this could be useful in some proofs or derivations somewhere (in this paper or elsewhere)?
> Note that error is never called under any circumstance in this implementation of runIter.

This technique can be contrasted with the more traditional trampolining technique for implementing tail recursion. Conventional trampolining uses a sum type of the result type and a thunk with the code necessary to continue execution. [Ganz et al. 1999]

In the technique presented here, we do not need to allocate thunks or closures. We actually do not to use higher-order functions at all in this tail recursion representation.

In the E type, this is used to represent tail recursion by the following constructors:

```
data E t where
  ...
  StepExp :: E b -> E (Iter a b)
  DoneExp :: E a -> E (Iter a b)
```

```
  TailRec :: (ERep a, ERep b)
                 => E (b -> Iter a b)
                    -> E (b -> a)
  ...
```

5 Representing lambdas

This representation of pattern matching depends on the ability to bring function values into the expression language. This is accomplished with the following constructors:

```
data E t where
  ...
  Lam ::
    (ERep a, Typeable a)
      => Name a -> E b -> E (a -> b)

  Var :: (Typeable a) => Name a -> E a
```

The `Typeable` constraints are necessary to lookup correctly typed values in the variable binding environment later on.

The `Name t` type represents a lambda variable identifier together with its type `t` (Note that the `ScopedTypeVariables` extension is enabled):

```
newtype Name a = Name Int
  deriving (Eq, Show)

namesEq :: forall a b.
  (Typeable a, Typeable b)
    => Name a -> Name b -> Maybe (a :~: b)
namesEq (Name n) (Name n') =
  case eqT :: Maybe (a :~: b) of
    Just Refl
      | n == n'    -> Just Refl
      | otherwise -> Nothing
    Nothing        -> Nothing
```

In the Core transformation, each lambda is given a `Name` with a globally unique `Int`, sidestepping any name capture issues.

The following datatypes are used to represent a variable binding environment of typed names to expression language values:

```
data EnvMapping where
  (:=>) :: forall a.
    Typeable a
    => Name a -> E a -> EnvMapping
```

This type encodes a single variable binding, with values of the form `n :=> v`, where `n` is a typed name and `v` is the value it is bound to.

These bindings are grouped together in the `Env` type:

```
newtype Env = Env [EnvMapping]

emptyEnv :: Env
emptyEnv = Env []

extendEnv :: Env -> EnvMapping -> Env
extendEnv (Env maps) m = Env (m:maps)

envLookup :: Typeable a
  => Env -> Name a -> Maybe (E a)
envLookup (Env maps) = go maps
  where
    go []                   = Nothing

    go ((n' :=> e):rest) =
      case namesEq n n' of
        Just Refl -> Just e
        Nothing   -> go rest
```

6 Recovering standard semantics for pattern matches

The standard semantics for the EDSL is given by `abs`. Two helper functions, `sumMatchAbs` and `prodMatchAbs`, are used to provide the standard semantics for matches on sum types and matches on product types, respectively. These helper functions are based on the mechanism described in section 3.4.

```
abs :: forall t. E t -> t
abs = absEnv emptyEnv

sumMatchAbs ::
  (ERepTy (ERepTy s) ~ ERepTy s, ERep s)
  => Env
    -> E (SumMatch (ERepTy s) t)
    -> s
    -> t
...

prodMatchAbs :: (ERep s)
  => Env
     -> E (ProdMatch s t)
     -> s
     -> t
...
```

See Appendix A for more implementation details of `absEnv`.

7 C Backend

TODO: Should this section be expanded?

The C backend makes use of a small EDSL for generating C code. See Appendix B for implementation details of this ESDL.

This backend is based on a globally unique name generator. Similar to SSA form, a new C variable is created for each Haskell subexpression. It differs from SSA, however, in that these variables can be assigned multiple times. In particular, a variable can be assigned in multiple branches of an `if` statement or inside of a `while` loop. This simplifies the implementation, as ϕ nodes are not needed.

The following `structs` provide the C representation for Haskell expressions:

```
typedef enum var_type_tag {
  EXPR_INT
, EXPR_PAIR
...
, EXPR_UNIT
} var_type_tag;

typedef enum semantic_type_tag {
  NO_SEMANTIC_TAG
, EXPR_COMPLEX     // Complex numbers
} semantic_type_tag;

struct closure_t;

typedef struct var_t {
  var_type_tag tag;
  semantic_type_tag semantic_tag;
  void* value;
} var_t;

typedef struct closure_t {
  var_t* fv_env;
  var_t (*fn)(var_t, struct closure_t*);
} closure_t;
```

The `semantic_tag` allows type information preserved by the `Typeable` constraints in the EDSL to be brought over to the generated C code. In our implementation, this is used to distinguish between pairs of `Doubles` and `Complex Double` values. In the case of a `Complex Double`, the `var_t` would have a `tag` field with the value `EXPR_PAIR` and a `semantic_tag` field with the value `EXPR_COMPLEX`.

8 Core Plugin

The Core plugin translates marked expressions. Expressions are marked by the `externalize` function:

```
externalize :: a -> E a
```

For example, `externalize x` marks the expression `x`.

If an expression already has an EDSL type (a type of the form `E a` for some `a`), then the marking procedure ignores it and does not wrap it with `externalize` (see M in Section 8.2).

In the flowchart given in Figure 1, the names in parentheses in a node refers to the names of transformations given in the rewrite rules in Figure 2. Calls to `unrep :: E a -> a` are only used internally and will not exist in the final result of the transformation.

Note that:

- The tail recursion transformation given by D, T'_{f} and T_{f} is completely independent of the `E` type and the transformations associated to the `E` type. As a result, the tail recursion transformation can be used on its own.
- The total number of marks introduced is upper bounded by the number of subexpressions in the original Core given to the transformation by GHC and each full transformation step (that is, C) eliminates one mark. Therefore, the transformation will terminate.

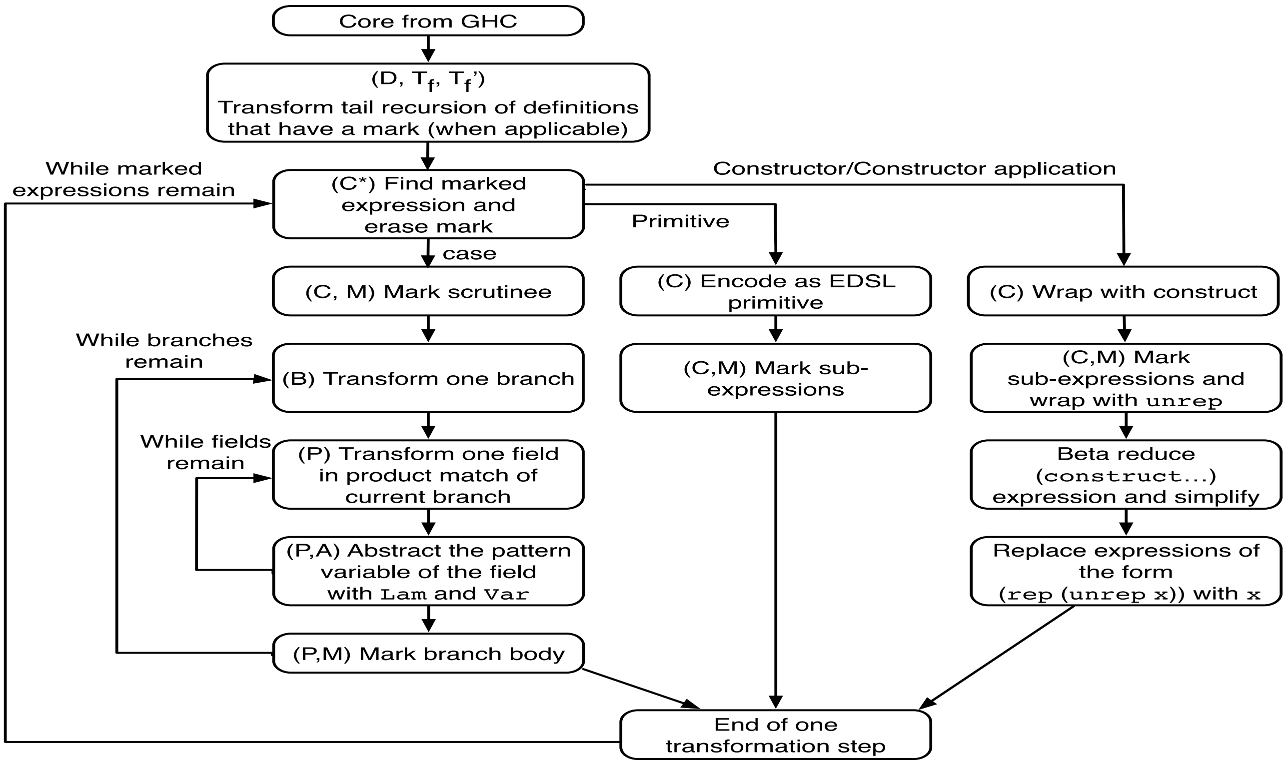

Figure 1. Control Flow of the Core Plugin

$$D(\texttt{f x = internalize (externalize (case s of \{ ... \}))})$$

$$\Longrightarrow \begin{cases} \texttt{f = internalize } C^*(\texttt{externalize (runIter (}\lambda\texttt{x} \rightarrow \texttt{case s of \{ } T'_{\texttt{f}}(\ldots)\texttt{ \}))}) & \text{if } \texttt{f} \text{ occurs free in } \texttt{\{ ... \}} \\ \texttt{f x = internalize } C^*(\texttt{externalize (case s of \{ ... \})}) & \text{otherwise} \end{cases}$$

$$T'_{\texttt{f}}(\texttt{K x0 } \ldots_U \texttt{ xN} \rightarrow \texttt{case s of \{ } \ldots_V \texttt{ \}; } \ldots_W)$$
$$\Longrightarrow \texttt{K x0 } \ldots_U \texttt{ xN} \rightarrow T_{\texttt{f}}(\texttt{case s of \{ } \ldots_V \texttt{ \}}); \ T'_{\texttt{f}}(\ \ldots_W \ )$$

$$T'_{\texttt{f}}(\texttt{K x0 } \ldots_U \texttt{ xN} \rightarrow \texttt{body0; } \ldots_V \ )$$

$$\Longrightarrow \begin{cases} \texttt{K x0 } \ldots_U \texttt{ xN} \rightarrow \texttt{body0[f} \mapsto \texttt{Step]; } T'_{\texttt{f}}(\ \ldots_V \ ) & \text{if } \texttt{f} \text{ occurs free in } \texttt{body0} \\ \texttt{K x0 } \ldots_U \texttt{ xN} \rightarrow \texttt{Done body0; } T'_{\texttt{f}}(\ \ldots_V \ ) & \text{otherwise} \end{cases}$$

$$T'_{\texttt{f}}(\varepsilon) \Longrightarrow \varepsilon \quad \text{(Note: } \varepsilon \text{ represents the empty string)}$$

$$T_{\texttt{f}}(\texttt{case s of \{ ... \}})$$

$$\Longrightarrow \begin{cases} \texttt{case s of \{ } T'_{\texttt{f}}(\ \ldots \ ) \texttt{ \}} & \text{if } \texttt{f} \text{ occurs free in } \texttt{\{ ... \}} \\ \texttt{Done (case s of \{ ... \})} & \text{otherwise} \end{cases}$$

$$C^*(\texttt{x}) \Longrightarrow \begin{cases} \texttt{x} & \text{if } \texttt{x} \text{ has no subexpressions marked with } \texttt{externalize} \\ C^*(C(\texttt{y})) & \text{if } \texttt{externalize y} \text{ is the first marked subexpression of } \texttt{x} \end{cases}$$

$$C(\texttt{runIter x}) \Longrightarrow \texttt{TailRec } M(\texttt{x})$$
$$C(\texttt{x + y}) \Longrightarrow \texttt{Add } M(\texttt{x}) \ M(\texttt{y})$$
$$C(\texttt{case scrutinee of \{ ... \}}) \Longrightarrow \texttt{CaseExp } M(\texttt{scrutinee}) \ B(\ \ldots \ )$$
$$C(\lambda\texttt{x} \rightarrow \texttt{body}) \Longrightarrow A(\lambda\texttt{x} \rightarrow \texttt{body})$$
$$C(\texttt{K x0 ... xN }) \Longrightarrow \texttt{construct (K (unrep } M(\texttt{x0})\texttt{) ... (unrep } M(\texttt{xN})\texttt{))} \quad \text{(Where } \texttt{K} \text{ is a constructor)}$$
$$C(\texttt{f x}) \Longrightarrow \texttt{App } M(f) \ M(x)$$

$$B(\texttt{K x0 } \ldots_U \texttt{ xN} \rightarrow \texttt{body0; } \ldots_V) \Longrightarrow \texttt{SumMatchExp } P(\texttt{x0 } \ldots_U \texttt{ xN} \rightarrow \texttt{body0}) \ B(\ \ldots_V \ )$$
$$B(\texttt{K x0 ... xN} \rightarrow \texttt{body}) \Longrightarrow \texttt{OneSumMatchExp } P(\texttt{x0 ... xN} \rightarrow \texttt{body})$$
$$B(\varepsilon) \Longrightarrow \texttt{EmptyMatch}$$

$$P(\texttt{x0 x1 ... xN} \rightarrow \texttt{body}) \Longrightarrow \texttt{ProdMatchExp } A(\lambda \texttt{ x0} \rightarrow P(\texttt{x1 ... xN} \rightarrow \texttt{body}))$$
$$P(\texttt{x} \rightarrow \texttt{body}) \Longrightarrow \texttt{OneProdMatchExp } A(\lambda \texttt{ x} \rightarrow \texttt{body})$$
$$P(\rightarrow \texttt{body}) \Longrightarrow \texttt{NullaryMatch } M(\texttt{body})$$

$$A(\lambda(\texttt{x :: a}) \rightarrow \texttt{body}) \Longrightarrow \texttt{Lam (Name @a uniq) } M(\texttt{body[x} \mapsto \texttt{unrep (Var @a uniq)]})$$
(where `uniq` is a globally unique identifier)

$$M(\texttt{unrep x}) \Longrightarrow \texttt{x}$$

$$M(\texttt{x :: a}) \Longrightarrow \begin{cases} \texttt{externalize x} & \text{if } \nexists t, a \sim E \ t \\ \texttt{x} & \text{if } \exists t, a \sim E \ t \end{cases}$$

Figure 2. Rewrite rules

9 Related Work

TODO: Does this need more fleshing out?

A similar EDSL-oriented representation of pattern matching is given in [Atkey et al. 2009, Section 3.3]. In that paper, patterns were given their own representation which allows for compound patterns. This is useful in the context of that work, as the programmer works directly with these patterns.

In the present paper, however, the representation is generated automatically by a compiler plugin. As a result of GHC's desugared Core (which does not have compound patterns), there is no need to directly express compound patterns at the representation-level.

There is other recent work using deep embeddings in functional languages for system development. One example is the Ivory language [Elliott et al. 2015] which provides a deeply embedded DSL for use in programming high assurance systems, However, it's syntax is typical of a deep EDSL and requires additional keywords and structures above idiomatic Haskell.

The Feldspar project [Axelsson et al. 2010; Svenningsson and Axelsson 2013] is a Haskell embedding of a monadic interface that targets C, and focuses on high-performance. Both Feldspar and our work mix deep and shallow language constructs [Persson et al. 2012; Sculthorpe et al. 2013; Svenningsson and Svensson 2013].

Svenningsson and Axelsson [Svenningsson and Axelsson 2013] explored combining deep and shallow embedding. They used a deep embedding as a low level language, then extended the deep embedding with a shallow embedding written on top of it. Haskell type classes were used to minimize the effort of adding new features to the language.

Yin-Yang [Jovanovic et al. 2014] provides a framework for DSL embedding in Scala which uses Scala macros to provide the translation from a shallow to deep embedding. Yin-Yang goes beyond the translation by also providing autogeneration of the deep DSL from the shallow DSL. The focus of Yin-Yang is in generalizing the shallow to deep transformations, and does not include recursive transformations.

Forge [Sujeeth et al. 2013] is a Scala based meta-EDSL framework which can generate both shallow and deep embeddings from a single EDSL specification. Embeddings generated by Forge use abstract `Rep` types, analogous to our EDSL's `E` types. Their shallow embedding is generated as a pure Scala library, while the deeply embedded version is generated as an EDSL using the Delite [Sujeeth et al. 2014] framework.

Elliott developed GHC plugins [Elliott 2015a][Elliott 2016] for compiling Haskell to hardware [Elliott 2015b], using worker-wrapper style transformations [Gill and Hutton 2009] equivalent to the `abs` and `rep` transformations described in Section 1.2. These plugins were later generalized to enable additional interpretations [Elliott 2017].

References

Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding Domain-Specific Languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1596638.1596644

Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and Andras Vajdax. 2010. Feldspar: A domain specific language for digital signal processing algorithms.. In *MEMOCODE'10*. 169–178.

Conal Elliott. 2015a. https://github.com/conal/lambda-ccc

Conal Elliott. 2015b. https://github.com/conal/talk-2015-haskell-to-hardware

Conal Elliott. 2016. https://github.com/conal/reification-rules

Conal Elliott. 2017. Compiling to categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 48 (Sept. 2017), 24 pages. https://doi.org/10.1145/3110271

Conal Elliott. 2018. The simple essence of automatic differentiation. In *Proceedings of the ACM on Programming Languages (ICFP)*. http://conal.net/papers/essence-of-ad/

Conal Elliott. 2019. Generalized convolution and efficient language recognition (extended version). *CoRR* abs/1903.10677 (2019). https://arxiv.org/abs/1903.10677

Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt free ivory. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. ACM, 189–200.

Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. 1999. Trampolined Style. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming* (Paris, France) *(ICFP '99)*. Association for Computing Machinery, New York, NY, USA, 18–27. https://doi.org/10.1145/317636.317779

Andy Gill and Graham Hutton. 2009. The worker/wrapper transformation. *Journal of Functional Programming* 19, 2 (March 2009), 227–251.

Mark Grebe, David Young, and Andy Gill. 2017. Rewriting a Shallow DSL Using a GHC Compiler Extension. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Vancouver, BC, Canada) *(GPCE 2017)*. ACM, New York, NY, USA, 246–258. https://doi.org/10.1145/3136040.3136048

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences* (Västerås, Sweden) *(GPCE 2014)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2658761.2658771

Anders Persson, Emil Axelsson, and Josef Svenningsson. 2012. Generic Monadic Constructs for Embedded Languages. In *Implementation and Application of Functional Languages*, Andy Gill and Jurriaan Hage (Eds.). Lecture Notes in Computer Science, Vol. 7257. Springer Berlin Heidelberg, 85–99.

Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. 2013. The Constrained-Monad Problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts). ACM, 287–298. http://dl.acm.org/citation.cfm?doid=2500365.2500602

Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25.

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, Indiana, USA) *(GPCE '13)*. ACM, New York, NY, USA, 145–154. https://doi.org/10.1145/2517208.2517220

Josef Svenningsson and Emil Axelsson. 2013. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*. Springer, 21–36.

Josef David Svenningsson and Bo Joel Svensson. 2013. Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th International Conference on Functional Programming*. ACM, 299–304.

A Additional Code for Standard Haskell Semantics Implementation

```
absEnv :: forall t. Env -> E t -> t
absEnv env (CaseExp x f) =
  sumMatchAbs f (absEnv env x)

absEnv env (TailRec f) = \ x ->
  case absEnv env f x of
    Step x' -> absEnv env (TailRec f) x'
    Done r  -> r

absEnv env (Var v) =
  case envLookup env v of
    Just x -> absEnv env
    Nothing ->
      error
        ("No␣binding␣for␣name␣"
          ++ show v)

absEnv env (Lam (name :: Name a)
                (body :: E b)) =
  \ (arg :: a) ->
    let go :: forall x. E x -> E x
        go expr@(Var name2 :: E a') =
          case namesEq name name2 of
            Just Refl -> rep arg
            _ ->
              case envLookup env name2 of
                Just v  -> v
                Nothing -> expr

        -- [... traverse rest of expression
        --      in go ...]
    in
    absEnv (extendEnv env
                      (name :=> rep arg))
           (go body)

absEnv env (LeftExp x) =
  Left (absEnv env x)

absEnv env (RightExp y) =
  Right (absEnv env y)

absEnv env (PairExp x y) =
  (absEnv env x, absEnv env y)

absEnv env (ConstructRep x) =
  unrep' (absEnv env x)

...
```

B A small EDSL for C code generation used internally by the C backend

```
type CCode = String
type CName = String

stmt :: CCode -> CCode
stmt = (<> ";")

cCall :: CName -> [CCode] -> CCode
cCall fnName args =
  fnName <> "(" <> intercalate ",␣" args <> ")"

derefPtr :: CCode -> CCode
derefPtr x = "*(" <> x <> ")"

cCast :: CCode -> CCode -> CCode
cCast toType x =
  "((" <> toType <> ")" <> x <> ")"

cIndex :: CCode -> Int -> CCode
cIndex x i = "(" <> x <> ")[" <> show i <> "]"

(!) :: CCode -> Int -> CCode
(!) = cIndex

(=:) :: CCode -> CCode -> CCode
x =: y = stmt (x <> "␣=␣" <> y)

cDecl :: CName -> CName -> CCode
cDecl cType v = stmt (cType <> "␣" <> v)

(#) :: CCode -> CName -> CCode
x # fieldName = x <> "." <> fieldName
```

```
macroDef :: CName -> [CName] -> [CCode] -> CCode
macroDef name args theLines =
  unlines
    [ "#define␣" <> name
              <> "(" <>
                intercalate ",␣" args
              <> ")\\"
    , "␣␣do␣{\\"
    , unlines
        . map ("␣␣␣␣"++)
        . map (++"\\")
        $ theLines
    , "␣␣}␣while␣(0);"
    ]
```