BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs

Agustín Mista[1] and Alejandro Russo[1]

Chalmers University of Technology, Gothenburg, Sweden
`{mista,russo}@chalmers.se`

Abstract. Embedded Domain-Specific Languages (EDSLs) are an alternative to quickly implement specialized languages without the need to write compilers or interpreters from scratch. In this territory, Haskell is a prime choice as the host language. EDSLs in Haskell, however, are often incapable of reifying useful static information from the source code, namely variable binding names and source locations. Not having access to variable names directly affects EDSLs designed to generate low-level code, where the variables names in the generated code do not match those found in the source code—thus broadening the semantic gap among source and target code. Similarly, many existing EDSLs produce poor error messages due to the lack of knowledge of source locations where errors are generated.

In this work, we propose a simple technique for enhancing monadic EDSLs expressed using **do** notation. This technique employs *source-to-source plugins*, a relatively new feature of GHC, to annotate every **do** statement of our EDSLs with relevant information extracted from the source code at compile time. We show how these annotations can be incorporated into EDSL designs either directly inside values or as monadic effects. We provide *BinderAnn*, a GHC source plugin implementing our ideas, and evaluate it by enhancing existing real-world EDSLs with relatively minor modification efforts to contemplate the source-level static information related to variables names and source locations.

Keywords: Embedded domain-specific languages, Haskell

1 Introduction

Embedded Domain-Specific Languages (EDSLs) are ubiquitous in Haskell. Its powerful type system and extensible syntax are among the reasons making it a very suitable programming language for implementing EDSLs [14]. Especially, monads [25] and monadic **do** notation [17] are part of programmers' toolbox to implement all sorts of EDSLs. Monadic **do** notation enables users to write domain-specific code in a sequential-like manner that it is easy to adopt by programmers not familiar to Haskell's syntax or even to functional programming languages.

```
semaphore = do
  green  ← node
  yellow ← node
  red    ← node
  green  .->. yellow
  yellow .->. red
  red    .->. green
```

(a) EDSL code describing a semaphore color cycle.

```
digraph G
{
 n0; n1; n2;
 n0 -> n1;
 n1 -> n2;
 n2 -> n0;
}
```

(b) Generated code without source information.

```
digraph G
{
 green; yellow; red;
 green -> yellow;
 yellow -> red;
 red -> green;
}
```

(c) Generated code using the BinderAnn plugin.

Fig. 1: Enhancing the `dotgen` code generating EDSL with source information.

As a result of being embedded, Haskell EDSLs often lack the ability of reflecting some of the static source information that is intrinsic and available to the host language (Haskell) but not in guest (the embedded DSL), namely bound names and source locations. These limitations are especially known by designers of EDSLs which generate low-level code, e.g., FeldSpar [3], Ivory [8], or Copilot [22]. In these EDSLs, developers adopted, as the best-case scenario, ad-hoc measures to enforce that variables names in the generated code match those in the host language. In this paper, we instead propose a systematic solution to such problems as a *source-to-source* plugin [21] called *BinderAnn*. We will illustrate the aforementioned limitations of Haskell EDSLs using a series of real-world examples of code generation, while we will show in tandem how our approach can be used to overcome it.

1.1 Motivating examples

We consider as motivating example the monadic EDSL from the `dotgen` package for generating DOT code[1] from inside Haskell [10]. This EDSL creates new graph nodes and connects them using **do** notation. A simple example of this is shown in Fig. 1a, where we create a graph of the alternating colors of a street semaphore.

Internally, this EDSL sequentially creates a fresh node name for each invocation of the `node` combinator, i.e, `n0`, `n1`, and so on. Then, the corresponding DOT code is generated referring to these generated names, as it is shown in Fig. 1b. Sadly, the generated code does not quite reflect the nature of our particular graph: *sequential names are of little help for interpreting the semantics of the generated code.* To make things worse, this is not a just limitation of this particular EDSL. The variable names to the left of binds (←) do not belong to an EDSL itself, but to the host language in which it is embedded—thus, such EDSL cannot make use of this useful source information directly.

Common practices To address this recurrent limitation, some EDSLs resolve in using redundant strings to indicate variable names when synthesizing code [9,22,2]. For instance, consider the EDSL for synthesizing C programs via SMT

[1]DOT is a graph description language used by many open source applications.

```
1 genAddSub = do
2   x ← cgInput "x"
3   y ← cgInput "y"
4   cgOutput "diff" (x − y)
5   cgReturn (x + y)
```

(a) EDSL code with redundant string names for generating terms.

```
genAddSub = do
  x ← cgInput
  y ← cgInput
  diff ← cgOutput (x − y)
  cgReturn (x + y)
```

(b) Simplified EDSL where names are extracted automatically by BinderAnn.

Fig. 2: Avoiding redundant string names in the `sbv` EDSL via source annotations.

solvers in the `sbv` package [9]. This EDSL enables to express relationships between the inputs and outputs of a function, and based on that, it generates its C body accordingly. Fig. 2a presents a very simple example of this, where we use the `cgInput` combinator to bind the function inputs `"x"` and `"y"` to the Haskell variables `x` and `y`, respectively, and then we specify how the outputs are calculated based on them. In this example, the function will simply return the sum of both inputs (line 5), while storing their difference in the output pointer `"diff"` (line 4). Then, the EDSL will generate the following C code:

```
SInt32 AddSub(SInt32 x, SInt32 y, SInt32 *diff){
  ...
  *diff = x - y;
  return (x + y);
}
```

where `...` simply indicates the rest of the generated code which is not relevant to the point being made here. Notice how the EDSL expects the users to give strings denoting variable names to the expressions they already bind with the *same* variable name but using **do** notation. While this common technique works in practice, this added redundancy requires maintenance and might be hard to keep in sync with the concrete Haskell bind variable names they replicate.

1.2 BinderAnn

In this paper, we present a novel technique to enhance existing (and future) EDSLs with the static information that is missing to generate faithful code, and without relying on redundant string names. In essence, our approach consists of automatically transforming the syntactic representation of our Haskell code to make the static information related to bound names explicitly available to EDSLs. This is now possible due to the recent addition of *source-to-source* plugins [21] to the GHC Haskell compiler.

Recalling our `dotgen` example, our approach can be used to generate DOT code that accurately reflects the one written by the user of the EDSL—see Fig. 1c Furthermore, Fig 2b shows how our approach can simplify the `sbv` EDSL by not requiring string names to be passed around while generating the same C code.

1.3 Beyond bindings

In practice, bound names are not the only kind of useful static information that can be extracted from EDSL code. Many EDSLs lack descriptive error messages which could be improved by having access to the source locations. To illustrate this point, we consider the EDSL provided by the `shellmate` package for executing shell scripts from Haskell [7]. With this EDSL, we can create computations capturing the output of existing shell commands:

```
cpuinfo = capture (run "cat" ["/proc/cpuinfo"])
meminfo = capture (run "cat" ["/proc/meminfo"])
```

And use them to build complex shell-like scripts:

```
1 saveInfo = do
2   cpu ← cpuinfo
3   mem ← meminfo
4   output "info.txt" (cpu ++ mem)
```

Let us suppose that we mistype the `"/proc/meminfo"` path. If we run our `saveInfo` script, the mangled path given to the command `cat` will produce a runtime exception that will be captured by the EDSL and printed to the user simply as:

```
Command "cat" failed with error code 1
```

This error message is hardly helpful for debugging the problem of our shell script, especially considering that many functions may be defined in terms of capturing the output of the `cat` command.

By using BinderAnn, it is also possible to extract the exact position in the user code where the error is triggered. In this light, we can enhance this EDSL to support more precise and useful error messages. For instance, the error message above can be improved to:

```
Exception raised at src/MyScript.hs:(3,3):
The value "mem" produced the following error:
Command "cat" failed with error code 1
```

Note how this error message now includes not only the name bound to the problematic command (`mem`), but also its position in the code.

The examples presented so far have motivated the development of BinderAnn to improve the capabilities of monadic EDSLs considerably. To summarize, the contributions of this paper are:

- We propose a simple yet powerful syntactic transformation technique for annotating monadic computations expressed using **do** notation with useful source information (Section 2).
- We propose two different *annotation styles* depending on how EDSLs can consume the static information provided to them, i.e., binding names and source locations (Section 3).

- We extend our simple transformation technique with support for annotating monadic computations returning and pattern matching against tuples, as well as a mechanism for controlling the transformation scope (Section 4).
- We provide an implementation of our ideas, in the shape of a GHC source-to-source plugin called BinderAnn.[2] With our plugin in mind from the beginning, we develop a complete case study from scratch, demonstrating how the ability of reifying source information automatically might unlock attractive new features in future EDSLs (Section 5).
- We discuss other possible approaches to fill the static information gap between hosts and guests embedded languages and their implications. Additionally, we reflect on the limitations of BinderAnn, as well as possible extensions to make it applicable to a larger space of EDSL (Section 6).

2 Generating Source Annotations Using Source Plugins

This section briefly describes *source-to-source plugins* (or source plugins for short), a new mechanism included in the GHC compiler for inspecting and transforming the parsed representation of the compiled code before any other transformation is performed. Moreover, we show how it is possible to take advantage of this mechanism to transparently enhance monadic code written using **do** notation with useful source information.

Essentially, a GHC plugin is a Haskell function that can be inserted into the compilation pipeline to transform the output of the compiled code in different ways [21,20]. These transformations can alter the compiled code at different stages, where each stage defines a different interface for its corresponding kind of plugin, dependent on the representation of the code used by the compiler at that point. Historically, this mechanism only allowed plugins to be inserted during type-checking, and after the code was transformed to GHC's Core intermediate representation [15]. Recently, GHC 8.6.1 also added support for plugins to be inserted after parsing and after renaming, and this work focuses on the former kind.

In GHC, the plugin interface is condensed in a record data type `Plugin`, containing a field for each of the transformation stages available. In particular, source-to-source plugins are given by the record field `parsedResultAction` of this data type:

```
data Plugin = Plugin {
  parsedResultAction :: [CommandLineOption] → ModSummary
                                → HsParsedModule → Hsc HsParsedModule
  ...
}
```

This field exposes the interface of a transformation function over the abstract syntax tree of the module under compilation (of type `HsParsedModule`). This abstract syntax tree includes relevant static information not available to the

[2] Available at `https://github.com/OctopiChalmers/BinderAnn`

programmer, such as the variable name of every binding, as well as the source location of every syntactic object in the module—two valuable resources that one might want to have access to when implementing EDSLs in Haskell.

Using this interface, we can implement our source plugin by providing a module exporting a value `plugin :: Plugin`, which executes our code transformation:

```
module BinderAnn (plugin) where
import GhcPlugins
plugin :: Plugin
plugin = defaultPlugin {parsedResultAction = <our code here>}
```

Later, our plugin can be enabled by passing the name of its module as a flag to the GHC compiler (`-fplugin=BinderAnn`), or using a compiler-options pragma in the module we want our plugin to transform:

```
{-# OPTIONS_GHC -fplugin BinderAnn #-}
```

Next subsection introduces a simple syntactic transformation procedure based on source plugins for transposing useful static information from the source code representation into the internal state of our EDSLs automatically.

2.1 Enhancing EDSLs with Source Information

We have seen that it is possible to expose static source information from our code using source plugins. However, for our EDSLs to take advantage of this information, we need to transform the user code so that it explicitly communicates this information to the EDSL after our plugin runs at compile time.

In this work, we propose a simple transformation over **do** statements: we will *annotate* each statement with the static information that can be extracted from the parsed representation of the code, which we will simply refer to as a *source annotation*. To achieve this, the first step consists of defining a concrete representation for source annotations, which will be used both by our plugin and by the target EDSLs it annotates. For this purpose, we will rely on a new data type `SrcInfo` to hold the static information relative to a **do** statement:

```
data SrcInfo = Info (Maybe String) (Maybe Loc)
```

This data type stores the name bound to the statement (if any), and the location in the source code where it is defined, being the latter a conjunction of a file path, and a row and column within such file:

```
type Loc = (FilePath, Int, Int)
```

The option type used for the location information in the definition of `SrcInfo` is required to represent the fact that the GHC compiler might not know the specific source location of a statement. A situation that might occur, for instance, if such statement was automatically generated by another source plugin.

Later, our source plugin can easily populate a source annotation (of type `SrcInfo`) for each **do** statement it finds. However, we still need to insert each annotation into our EDSL in a predictable way. For this purpose, we will define a function `annotateM`, taking a monadic computation and a source annotation, and returning a new monadic computation which internalizes such annotation:

```
annotateM :: Monad m ⇒ m a → SrcInfo → m a
```

Note how this function refers neither to a specific monadic type (`m`) nor to a specific return type of the monadic computation (`a`). This generality lets our plugin blindly transform every **do** statement it finds in the user code in a type-safe manner. To do so, it simply wraps every statement with its static information using our generic annotation function. For instance, our plugin will transform the semaphore example from Section 1 to the following concrete code:

```
1 semaphore = do
2   green  ← node     `annotateM` Info (Just "green")  (Just ("Main.hs", 2, 3))
3   yellow ← node     `annotateM` Info (Just "yellow") (Just ("Main.hs", 3, 3))
4   red    ← node     `annotateM` Info (Just "red")    (Just ("Main.hs", 4, 3))
5   green  .->. yellow `annotateM` Info Nothing         (Just ("Main.hs", 5, 3))
6   yellow .->. red    `annotateM` Info Nothing         (Just ("Main.hs", 6, 3))
7   red    .->. green  `annotateM` Info Nothing         (Just ("Main.hs", 7, 3))
```

Notice, for instance, how the bound name `red` is reflected in the source annotation for the `red ← node` statement with the value `Just "red"`, whereas the `green .->. yellow` statement in the next line is not given any name, which gets represented by the `Nothing` constructor on its corresponding source annotation.

Additionally, each annotation carries the source location within the user code of its corresponding statement—assuming here that the first **do** statement is defined in line number 2 of the file *Main.hs*.

After this transformation is automatically applied, the user will be able to make use of this useful source information, which is now explicit in the source code—and without the burden of maintaining manually written annotations.

Even though this transformation is rather mechanical, the behavior of the annotating function `annotateM` is not trivial, and is subject to *which* types of our EDSLs are expected to be annotated, and *how* the source annotations should be consumed by them. The next section addresses the challenges of implementing this function in depth.

3 Consuming Source Annotations

In the previous section, we demonstrated how it is possible to annotate expressions written using **do** notation with source information via source plugins. Such annotations rely on a generic function `annotateM` to produce the annotation effect. This section explores the details of this function in two possible variants.

Haskell gives the programmer the freedom to implement EDSLs in many ways, depending on the nature of the embedded language. As a consequence, a concrete solution for annotating EDSLs would likely not fit many use cases. In this light, our approach supports two different *annotation styles* that the programmer can use depending on the particular implementations of their EDSLs:

- *Effect-free annotations:* the annotations are stored directly on the values they refer to, e.g, using a specialized data constructor, or an option type.
- *Effect-full annotations:* the annotations are kept in a monadic context as a side effect, e.g., using a mapping from values to annotations inside a state monad.

On one hand, the effect-free style lets us annotate values in place, regardless of the monadic context producing them, which might come in handy if our EDSL defines several monadic types to be used by the end-user. On the other hand, the effect-full style lets us insert the source annotations in the monadic context without having to modify the return value of each computation. This style might be useful if our EDSL already carries an internal monadic state, or if the source annotations should not be available to the end-user.

Both annotation styles are *independent* of each other and provide different interfaces to interact with BinderAnn. Programmers will then have to choose the most suitable one depending on the nature of their EDSLs, and adapt their code to be able to consume the annotations generated by our plugin.

The rest of this section addresses each annotation style in detail.

3.1 Effect-Free Annotations

The simplest way to annotate a value with source information is given when its type already supports annotations. For instance, suppose that the graph-building EDSL from Section 1 defines graph nodes as having an identifier, and an associative list of attributes as payload:

```
data Node = Node Id [(Attr, Value)]
```

With this in place, the rest of the EDSL combinators can be implemented in terms of nodes as inputs and outputs:

```
node   :: Dot Node
(.->.) :: Node → Node → Dot ()
```

where `Dot` is the main monad defined by this EDSL, whose details are not very relevant for this annotation style. To support generating faithful code, we can extend the definition of the `Node` data type to also carry an optional field representing the name of each node:

```
data Node = Node Id (Maybe String) [(Attr, Value)]
```

Then, we need to somehow specify that every monadic computation returning a `Node` should (potentially) be annotated with its bound name. To encode this, we can define a new *type class* [13] `Annotated`, representing types (of type `a`) that can be annotated directly:

```
class Annotated a where
  annotate :: a → SrcInfo → a
```

The function `annotate` simply takes a value and an annotation and returns an annotated value of the same type. Then, we can specify how the source bound names can be inserted into nodes by giving an appropriate `Annotated` instance:

```
instance Annotated Node where
  annotate (Node id _ attrs) (SrcInfo name loc) = Node id name attrs
```

where we simply extract the bind name from the source annotation and use it as the node name—for simplicity, we discard the location information here.

Using this type class, we can finally implement our desired `annotateM` function which transforms **do** statements by unwrapping the return value from the monadic computation and returning the corresponding annotated one:

```
annotateM :: (Monad m, Annotated a) ⇒ m a → SrcInfo → m a
annotateM ma ann = do
  a ← ma
  return (annotate a ann)
```

This is an extensible mechanism that lets us support automatic annotations over the return types of our interest. We simply need to provide an instance of the `Annotated` type class for every return type of a **do** statement we want to annotate using our plugin.

While simple, this transformation is not safe (yet). Recalling from Section 2, our plugin knows nothing about the return type of a **do** statement. Hence, it transforms every statement it finds under the assumption that this transformation will not produce a type error—as `annotateM` universally quantifies over any possible return type of the monadic computation it transforms. However, our `annotateM` function now carries an additional `Annotated` constraint! In practice, this means that our plugin will break the well-typedness of our EDSL if it happens to find a monadic computation returning a value of a type without an `Annotated` instance. And even though we could potentially provide an `Annotated` instance for every type used by our EDSL, a user could always write a statement returning a value of a type not known by our EDSL:

```
x ← return False
```

Here, the lack of an instance for `Annotated Bool` will break the module during type checking.

To attenuate this problem, we can make every type trivially annotatable by simply discarding the annotation altogether:

```
instance {-# INCOHERENT #-} Annotated a where
  annotate a _ = a
```

This generic instance works as a default trivial annotation method, where any concrete `Annotated` instance written by the programmer will take precedence against this one [16]. Furthermore, note how this default instance requires to be declared as *incoherent*. This ensures that the type checker will pick a concrete instance written by the user whenever possible, but it will conservatively use the default one in case of an overlapping arising from annotating fully-polymorphic functions—we discuss this in detail in Section 6.4.

3.2 Effect-Full Annotations

EDSLs might also be implemented in a fully stateful manner, where the important data is kept in the monadic context, and the user only gets a reference to handle it. For instance, suppose that our graph-building EDSL from Section 1 does not return nodes directly, but references to them instead:

```
data NodeRef = NodeRef Id

node :: Dot NodeRef
(.->.) :: NodeRef → NodeRef → Dot ()
```

Here, the node payload will be kept in an internal state of the `Dot` monad defined by the EDSL, which could be defined on terms of a state monad:

```
newtype Dot a = Dot (State DotState a)
data DotState = DotState {
  node_attrs :: Map NodeRef [(Attr, Value)]
}
```

In this case, we might as well want our annotation mechanism to follow the same pattern, inserting the annotations in the monadic context instead of directly in the value they refer to. For this purpose, we can extend our `DotState` type to also carry the source names given to the bound nodes (if any) using a partial mapping:

```
data DotState = DotState {
  node_attrs :: Map NodeRef [(Attr, Value)],
  node_names :: Map NodeRef String
}
```

Similarly as before, we can define a new type class to specify how to annotate values of different types, except that this time we also need to quantify over the specific monadic context in which the annotation takes place:

```
class Monad m ⇒ AnnotatedM m a where
  annotateM :: m a → SrcInfo → m a
```

Notice that this new type class defines our desired `annotateM` function directly. In contrast to the previously seen `Annotated` type class from the previous subsection, this type class let us specify how **do** statements can be annotated depending not only on their result type but also on their specific monadic type. In this light, computations returning new node references can be annotated within the `Dot` monad by inserting the bound names in the extended internal state:

```
instance AnnotatedM Dot NodeRef where
  annotateM mref (Info name loc) = do
    ref ← mref
    when (isJust name) $ modify $ λs →
      s {node_names = Map.insert ref (fromJust name) (node_names s)}
    return ref
```

As before, we also need to provide a default instance for our new type class, to ensure that our plugin will not break the well-typedness of the user code:

```
instance {-# INCOHERENT #-} Monad m ⇒ AnnotatedM m a where
  annotateM ma _ = ma
```

All in all, the two annotation styles presented in this section cover a wide variety of EDSL implementation patterns.

4 Extensions

This section describes two useful extensions to our annotation approach that are currently supported by our plugin.

4.1 Annotating Computations Returning Tuples

The syntactic transformation described so far contemplates monadic computations with and without bound names. However, in principle we could only use it to extract the names bound to complete resulting values, i.e, when the pattern at the left hand side of (←) is a plain variable pattern. In practice, a computation could produce multiple values and return them in a tuple. For instance, suppose that our graph-building EDSL example from Section 1 provides a combinator `nodes` returning multiple new nodes at once:

```
(green, yellow, red) ← nodes
```

For this common programming practice, we would want to insert an annotation for each element of this tuple, following the same pattern as we did before. However, our source annotations can only associate a single name bound to a complete result value of a monadic computation.

Fortunately, we can extend our plugin to support tuple results by inserting a function that lifts our annotation mechanism to each element of the resulting tuple:

```
(green, yellow, red) ← nodes
  `annotateM3`
    (Info (Just "green")  (Just ("Main.hs", 2, 4)),
     Info (Just "yellow") (Just ("Main.hs", 2, 10)),
     Info (Just "red")    (Just ("Main.hs", 2, 18))
```

where annotateM3 simply extracts each tuple element from the monadic computation, annotates it using the ordinary annotation function, and returns a new tuple containing each annotated value:

```
annotateM3 :: Monad m ⇒ m (a, b, c) → (SrcInfo, SrcInfo, SrcInfo) → m (a, b, c)
annotateM3 mabc (ia, ib, ic) = do
  (a, b, c) ← mabc
  a' ← return a `annotateM` ia
  b' ← return b `annotateM` ib
  c' ← return c `annotateM` ic
  return (a', b', c')
```

It is easy to see how this lifting primitive can be trivially generalized to tuples of any fixed size.

4.2 Specifying the Annotation Scope

By default, our annotation plugin will transform *every* **do** expression present on the module it runs over. Even though a module can contain **do** expressions of different monadic types, we have shown in Section 3 how this transformation can effectively affect only those expressions of the types the user is interested in.

Nonetheless, for a given type to be annotated with source information, a user might still want to limit the scope of the annotations to a certain subset of **do** expressions. To support this, our plugin can also be set to work in a selective mode, where the user specifies which **do** expressions should be transformed.

On one hand, if the target expression is bound to a top-level name, we can use a GHC *annotation pragma* to specify that we are interested in annotating it:

```
{-# ANN semaphore SrcInfo #-}
semaphore = do
  <annotated do statements>
```

This way, BinderAnn will begin by reifying all the annotation pragmas defined in the module, and will proceed to transform only those **do** expression for which a corresponding annotation pragma exists.

However, annotation pragmas can only refer to top-level bindings, limiting the applicability of this technique. In practice, writing **do** expressions at the right hand side of the ($) infix function application operator is quite common. For instance, a user might define a graph and render its DOT code right away:

```
semaphoreCode = showDot $ do
  <do statements>
```

There is no top-level name we can use to specify our plugin to annotate this nested **do** expression. To solve this, we can introduce an *infix annotation operator*. This is, we can replace the infix function application operator ($) with a new syntactic operator, e.g., (|$|), that can be sought within the user's code in order to transform nested **do** expressions:

```
semaphoreCode = showDot |$| do
  <annotated do statements>
```

Then, our plugin will transform every **do** expression at the right hand side of a (|$|) operator to include the appropriate source annotations, replacing it with normal function application in the process. In practice, the programmer can specify the annotation operator to be any valid infix operator name using a plugin option in BinderAnn (`-fplugin-opt BinderAnn:infix=|$|`).

This gives us the freedom to choose the most appropriate operator according to the nature of the embedded language. Additionally, the infix annotation operator can be defined as a synonym to the actual function application operator:

```
(|$|) :: (a → b) → a → b
(|$|) = ($)
```

This way, the behavior of our code does not change when the plugin is disabled.

Next section develops a complete case study, exploring some interesting features that our plugin enables and can aid in implementing in future EDSLs.

5 Case Study: Theorem Proving EDSL

So far we have seen how source annotations can be automatically extracted from the source code using a GHC source plugin (Section 2), as well as consumed to our EDSLs in different ways depending on how they are implemented (Section 3).

Using this approach, we enhanced several existing EDSLs [10,9,7,1] (including the ones presented in Section 1) to support source annotations, obtaining attractive results[3]with relatively small effort.

To demonstrate the full potential of our automated transformation technique, this section introduces a novel case study we designed from scratch having source annotations in mind. In this light, we implemented a simple proof assistant EDSL for propositional logic formulas,[4] based on Coq's [4] tactic style, i.e., our proofs will consist of a series of monadic commands (the tactics) which will manipulate our goals and hypotheses to construct a proof for a given target formula.

Despite not being academically enlightening, this EDSL uses the effect-full annotation style to take advantage of the source information present in the user code, in order to provide useful interactive (modulo recompilation time) proof-state reports—an attractive feature that was not possible to achieve before using monadic EDSLs. To give an example of this, Fig. 3a shows a proof of *Modus ponens* discharged using our EDSL. Firstly, we use the combinator `variables` to create two new propositional variables `p` and `q` (line 3). These variables are used immediately in line 4, where the `proof` combinator establishes the current proof goal ($p \wedge (p \Rightarrow q) \Rightarrow q$) and we can proceed to prove it using the **do** expression starting after the ($) operator.

[3]Available at `http://github.com/OctopiChalmers/BinderAnn-examples`
[4]Available at `http://github.com/OctopiChalmers/PropProver`

```
1 modus_ponens :: Proof Prop
2 modus_ponens = do
3   (p, q) ← variables
4   proof (p ∧ (p ⇒ q) ⇒ q) $ do
5     hand ← intro
6     (hp, hpq) ← destruct hand
7     hq ← apply hp hpq
8     exact hq
9     qed
```

(a) A proof of Modus Ponens using **do** notation in our EDSL.

```
At Proofs.hs:(7,5):
1 subgoal left
p, q: Prop
hand: p ∧ (p ⇒ q)
hp: p
hpq: p ⇒ q
hq: q
===================
q
```

(b) Proof state using source annotations.

```
Incomplete proof:
1 subgoal left
V0, V1: Prop
H0: V0 ∧ (V0 ⇒ V1)
H1: V0
H2: V0 ⇒ V1
H3: V1
===================
V1
```

(c) Proof state using internal names.

Fig. 3: User interface of our Coq-like, tactics-based proof assistant EDSL.

The proof itself uses a series of tactic combinators to progressively manipulate our goal and hypotheses in order to prove our goal. In the first place, we introduce the left hand side of the top-level implication goal as a new hypothesis named `hand` using the `intro` combinator (line 5), leaving us with the responsibility of proving its consequence, i.e., `q`. From here, we use the `destruct` combinator to split our conjunction hypothesis `hand` into two new hypotheses named `hp` and `hpq`, representing each side of the conjunction (line 6). Having the hypotheses `p` and `p ⇒ q` now in scope, we use the `apply` tactic to eliminate the latter applying it the former, obtaining a new hypothesis `hq` which represents our goal (line 7). Our proof concludes in line 8 by telling the EDSL to use the specific hypothesis `hq` as a proof of our goal, using the `exact` combinator. The final `qed` command at line 9 simply asserts that the proof given matches the current goal, and returns the proven proposition.

While writing this proof, our EDSL assists the user with a report of the current proof state on each step. For instance, by removing the last tactic we apply (line 8), the corresponding proof state given to the user is the one shown in Fig. 3b. Notice how this report reflects the same variable and hypothesis names introduced by the user in the proof code, i.e., `p`, `q`, `hand`, and so on. Additionally, it indicates the current proof position within our file, which is also used to emit a precise error message whenever some tactic is applied incorrectly—all these features being now possible thanks to our plugin.

To illustrate how helpful this information is for our EDSL, Fig. 3c illustrates the same proof state report we would obtain without reified source annotations (by disabling our plugin for instance). There, both variable and hypothesis names are just printed out using their internal names. Moreover, the current proof-state source position is not available either. Together, these two compromises limit the attractiveness of implementing elegant embedded proof assistants in Haskell.

Implementation To implement our EDSL, we will start by defining our main monadic data type `Proof` by stacking two monads: a `StateT` transformer to keep an implicit proof state, on top of an `Except` monad to raise and catch proof-related errors:

```
newtype Proof a = Proof (StateT ProofState (Except ProofError a))
```

The most interesting bit here is how we define our proof state. In essence, we will keep a set of propositional variables in scope, along with a stack of subgoals (propositional formulas to construct) and their corresponding context:

```
data ProofState = ProofState {
  ps_vars     :: Set Var,
  ps_subgoals :: [(Prop, Context)]
}
```

Here, variables are represented simply as numbers, whereas contexts are mappings from hypotheses (also represented as numbers) to propositions:

```
newtype Var = Var Int
newtype Hyp = Hyp Int
type Context = Map Hyp Prop
```

Finally, propositions are represented using a simple recursive data type encoding each logical connective:

```
data Prop = Var Var | Prop ∧ Prop | Prop ⇒ Prop | ···
```

The machinery introduced so far is enough to implement the core logic of our EDSL and its proof tactics. However, to take advantage of the source information extracted by our plugin using the effect-full annotation style, we will further extend our proof state with three additional fields to keep track of the source information relevant to our proofs:

```
data ProofState = ProofState {
  ···
  ps_var_names :: Map Var String,
  ps_hyp_names :: Map Hyp String,
  ps_curr_pos  :: Maybe Loc
}
```

These new fields will help us keeping track of: the source name given to each propositional variable (introduced by the `variables` combinator); the source name given to each new hypothesis (introduced by our different tactics); and the location in the source code of the last command evaluated by the EDSL (if any).

Then, to connect this internal state to the source annotations generated by our plugin, we need to consider the different result types that each combinator of our EDSL produces. In first place, our `variables` combinator is used to instantiate new propositional variables (of type `Var`). In this light, we can create an annotation rule (using an `AnnotatedM` instance) to store the source name each variable is given by the user (if any) into the internal names mapping of our proof state:

```
 1 instance AnnotatedM Proof Var where
 2   annotateM mvar (Info name loc) = do
 3     updateCurrentPosition loc
 4     var ← mvar
 5     when (isJust name) (recordVarName var (fromJust name))
 6     return var
```

where `recordVarName` (line 5) inserts the bind name (if any) coming from the source annotation into the internal variable names mapping:

```
recordVarName :: Var → String → Proof ()
recordVarName var name = modify $ λs →
  s {ps_var_names = Map.insert var name (ps_var_names s)}
```

Additionally, the function `updateCurrentPosition` (line 3) simply updates the location in the code of the last command executed by the EDSL (if any):

```
updateCurrentPosition :: Maybe Loc → Proof ()
updateCurrentPosition loc = modify $ λs → s {ps_curr_pos = loc}
```

The next thing we need to consider is how the result of each tactic affects the source information collected in the internal proof state. In principle, proof tactics can return either a new hypothesis (or a tuple of them), when they cause new hypotheses to appear in the proof state, e.g., `intro` or `apply` tactics; or a unit value, when they transform the proof state without introducing any new hypothesis, e.g., the `exact` tactic. With this in mind, we will provide two additional annotation rules to be executed whenever a proof tactic returns either a new hypothesis (of type `Hyp`) or nothing (of type ()):

```
 1 instance AnnotatedM Proof Hyp where
 2   annotateM mhyp (Info name loc) = do
 3     updateCurrentPosition loc
 4     hyp ← mhyp
 5     when (isJust name) (recordHypName hyp (fromJust name))
 6     return hyp
 7 instance AnnotatedM Proof () where
 8   annotateM munit (Info name loc) = do
 9     updateCurrentPosition loc
10     munit
```

The first `AnnotatedM` instance (line 1) will store the source name each hypothesis is given by the user (if any) into the internal proof state—the function `recordHypName` from line 5 works analogously as `recordVarName`. As before, we keep track of the last command evaluated by the EDSL in case of a proof error.

For the case of the second `AnnotatedM` instance (line 7), tactics not producing new hypotheses will not bring new source names to store into the internal proof state. However, this instance makes sure that if such a tactic fails, we have its position logged into our internal proof state in order to report a precise error message (line 9).

With these `AnnotatedM` instances in place, our plugin will seamlessly interact with them, keeping track automatically of source names introduced by the users in their code, as well as the location of each tactic invocation in case of having to report a proof-related error.

6 Discussion

We have presented a simple mechanism based on source plugins for enhancing Haskell EDSLs with source information. This section reflects on other approaches for supporting the extraction of source information without relying on source plugins. Moreover, we discuss limitations and possible extensions to our approach.

6.1 Preprocessing Haskell Code

Our approach is based on transforming the user code adding explicitly some of the useful information that gets lost during compilation. The main advantage of source plugins is that they provide a simple way of doing so without relying on external machinery. Before their existence, achieving the same kind of functionality would have required a substantial amount of effort.

For an overview of other possible (and arguably less pleasant) solutions of this problem, we refer the reader to the work of Dévai et al. [5]. There, the authors propose different indirect techniques for enhancing Haskell EDSLs with static information, e.g., using *cpphs*, the Haskell implementation of the C preprocessor; as well as transforming the Haskell source AST using existing parsers and pretty printers before feeeding it to the actual compiler.

6.2 Implementing EDSLs Using QuasiQuotation

In contrast to preprocessing our Haskell code to include static information, it is also somewhat possible achieve the same goal using meta-programming.

Template Haskell [23] is the Haskell meta-programming framework bundled in the GHC compiler. This tool can be used to inspect the typing information present in the user's codebase and synthesize new code depending on it but, for technical reasons, inspecting term definitions or modifying existing Haskell code is not possible, making this framework unsuitable for implementing a transformation-based approach. Nonetheless, a useful feature of Template Haskell used by many existing EDSLs [24,8,12,19,11] is the support for *quasiquotation* [18]. Essentially, quasiquotation allows to embed code written using arbitrary, domain-specific syntax into our Haskell code. To do so, this approach relies on implementing *quasi quoters*, i.e., interpretations from arbitrary strings to their corresponding Haskell expressions:

```
data QuasiQuoter = QuasiQuoter {
  quoteExp :: String → Q Exp,
  ...
}
```

where `Q` is the quasiquotation monad defined by Template Haskell.

Using this approach, it would be possible to implement our Coq-like EDSL from Section 5 as a quasi quoter `coq` :: `QuasiQuoter` accepting concrete Coq syntax. Then, we could use it to embed Coq proofs into our Haskell EDSL using quasiquotation brackets syntax (`[|` ··· `|]`):

```
 1 modus_ponens :: Proof Prop
 2 modus_ponens = [coq|
 3   Variables P Q.
 4   Theorem (P ∧ (P → Q) → Q).
 5     Proof.
 6     intro hand.
 7     destruct hand as [hp hpq].
 8     apply hp hpq as hq.
 9     exact hq.
10   Qed.
11 |]
```

An advantage of this approach is that the arbitrary code written inside of the quasiquotation brackets has (almost) no syntactic restrictions. Hence, it can be used to embed domain-specific code written using the syntax that fits best the nature of a given EDSL, as opposed to the syntactic restrictions imposed by the use of Haskell syntax and **do** notation—which are exploited by BinderAnn.

However, all this flexibility does not come for free. Implementing a quasi quoter for a language with a novel syntax implies writing a lexer and a parser from a plain string to a Haskell expression—a task that might overcome the benefits of having a new specialized syntax. Moreover, the interaction between quasiquoters and native Haskell code tends to be intricate. In particular, enabling quasiquoters to support embedding native Haskell code inside quasiquotation brackets (something known as *antiquotation*) requires a considerable amount of work and knowledge [18]—without this feature, our quasiquoters can only accept constant EDSL expressions inside the quasiquotation brackets.

Extracting bound names becomes possible using quasi quoters, since, as we mention above, we have access to the literal string written by the user. Source locations, on the other hand, are more tricky to infer. By default, quasi quoters will only be able to recognize source locations relative to where the quasiquotation brackets are interpolated in our Haskell code (line 2 in our example above), difficulting the task of giving the end-user error messages referring to absolute locations within their code.

6.3 Source Annotations for Non-Monadic EDSLs

In this work, we decided to focus only on automatically annotating monadic EDSLs expressed using **do** notation. Although it may seem arbitrary, the reason behind this decision is simple: **do** notation gives us a good level of granularity. Our plugin perform statement-wise transformations, matching the natural notion of having one domain-specific command or instruction per **do** statement. This symmetry lets us annotate EDSL very transparently for the end-user.

On the other hand, there exists many remarkable non-monadic EDSLs written in Haskell and not supporting them by default constitutes a noticeable limitation of our current approach. In principle, we could use the pure annotation style introduced in Section 3 to insert annotations into pure values. However, it is the lack of a well-defined statement structure what complicates deciding *where* to insert source annotations. On one hand, annotating only top-level bindings might be too sporadic for practical purposes, while doing so for every subexpression within a value might blow up the size of our transformed code exponentially, so an acceptable annotation granularity would seem to lay somewhere in between of these two extremes—an intriguing problem to drive our future work.

6.4 Use of Incoherent Instances

As mentioned in Section 3, our approach let us inject source annotations into the values of certain types of interest, and relies on default instances to provide trivial implementations of the annotation functions for any other possible value.

Instead of having to provide concrete annotation instances for each possible type present in the user code, these default instances are a convenient feature that allows doing so on a per-case basis while preserving the type-correctness of the user code after it is transformed by our plugin. Sadly, this convenience has as a limitation that *annotations inserted into fully-polymorphic functions will be systematically discarded.* To illustrate this, consider for example the following function that duplicates the output of a monadic computation:

```
twice :: Monad m ⇒ m a → m (a, a)
twice ma = do
  x ← ma
  return (x, x)
```

If written by the user of the EDSL, and then annotated by our plugin, this function will trigger a type error when there exists at least a single more concrete `Annotated` or `AnnotatedM` instance. The reason behind this is simple: while type checking the annotated statement `x ← ma`, only the default annotation instance is polymorphic enough to match the type of `ma`, however, it cannot be chosen directly, as the existence of other more concrete ones would make this choice inconsistent, e.g, using the default instance even when `twice` is instantiated in the user code with a type that has a more concrete one. Then, declaring our default instances as incoherent loosens this constraint, allowing the compiler to choose the default instance whenever it has to solve an overlap while compiling fully-polymorphic functions like `twice`, but leaving us with the aforementioned limitation as a result of this conservative behavior.

The complexity around the use of overlapping instances is well known by the Haskell community. In this light, this problem has been solved using more sophisticated approaches relying on type-level programming, e.g., using *closed type families* [6]. Adopting them in our plugin without sacrificing its transparency and ease of use is an ambitious problem that we keep as future work.

7 Conclusions

We developed a simple mechanism to facilitate the automatic extraction of useful source code information that is otherwise lost during compilation. Having access to such information when implementing embedded domain-specific languages is extremely valuable, making possible to implement attractive features such as faithful code generation and precise error messages. In the past, such features were more complicated if not impossible to achieve without involving undesirable trade-offs like repeated code or quasiquotations.

In the future, we aim to investigate how to extend our approach to a wider set of EDSL programming patterns, especially to those implemented using non-monadic combinators, and for which the use of **do** notation is not available. Additionally, we intend to evaluate how our annotation framework could be extended using generic programming techniques, so programmers should not need to adapt their existing EDSL data type definitions to work with it.

Acknowledgments. We want to thank Koen Claessen for the useful feedback given throughout the development of this work. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet.

References

1. Algehed, M., Jansson, P., Einarsdóttir, S.H., Gerdes, A.: Saint: An API-generic type-safe interpreter. In: Pałka, M., Myreen, M. (eds.) Trends in Functional Programming. pp. 94–113. Springer International Publishing (2019)
2. Axelsson, E.: Compilation as a typed EDSL-to-EDSL transformation. arXiv preprint arXiv:1603.08865 (2016)
3. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). pp. 169–178. IEEE (2010)
4. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant reference manual: Version 6.1 (1997)
5. Dévai, G., Leskó, D., Tejfel, M.: The EDSL's struggle for their sources. In: Central European Functional Programming School. pp. 300–335. Springer (2013)
6. Eisenberg, R.A., Vytiniotis, D., Peyton Jones, S., Weirich, S.: Closed type families with overlapping equations. ACM SIGPLAN Notices **49**(1), 671–683 (2014)
7. Ekblad, A.: shellmate: Simple interface for shell scripting in Haskell. (2014), `https://hackage.haskell.org/package/shellmate`
8. Elliott, T., Pike, L., Winwood, S., Hickey, P., Bielman, J., Sharp, J., Seidel, E., Launchbury, J.: Guilt free ivory. In: ACM SIGPLAN Notices. No. 12, ACM (2015)
9. Erkok, L.: sbv: SMT based verification: Symbolic Haskell theorem prover using SMT solving. (2010), `https://hackage.haskell.org/package/sbv`

10. Gill, A.: dotgen: A simple interface for building .dot graph files. (2008), `https://hackage.haskell.org/package/dotgen`
11. Giorgidze, G., Grust, T., Schreiber, T., Weijers, J.: Haskell boards the ferry. In: Symposium on Implementation and Application of Functional Languages. pp. 1–18. Springer (2010)
12. Giorgidze, G., Nilsson, H.: Embedding a functional hybrid modelling language in Haskell. In: Symposium on Implementation and Application of Functional Languages. pp. 138–155. Springer (2008)
13. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems (TOPLAS) **18**(2), 109–138 (1996)
14. Hudak, P., et al.: Building domain-specific embedded languages. ACM Comput. Surv. **28**(4es), 196 (1996)
15. Jones, S.L.P., Santos, A.M.: A transformation-based optimiser for Haskell. Science of computer programming **32**(1-3), 3–47 (1998)
16. Jones, S.P., Jones, M., Meijer, E.: Type classes: an exploration of the design space. In: Haskell workshop. pp. 1–16 (1997)
17. Launchbury, J.: Lazy imperative programming. In: Workshop on State in Programming Languages, Copenhagen, Denmark, ACM (1993)
18. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. pp. 73–82. ACM (2007)
19. Mainland, G., Morrisett, G.: Nikola: embedding compiled GPU functions in Haskell. In: ACM Sigplan Notices. vol. 45, pp. 67–78. ACM (2010)
20. Marlow, S., Jones, S.P., et al.: The Glasgow Haskell compiler (2004)
21. Pickering, M., Wu, N., Németh, B.: Working with source plugins. In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell. ACM (2019)
22. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: International Conference on Runtime Verification. pp. 345–359. Springer (2010)
23. Sheard, T., Jones, S.L.P.: Template meta-programming for Haskell. SIGPLAN Notices **37**(12), 60–75 (2002)
24. Snoyman, M.: Developing web applications with Haskell and Yesod. O'Reilly Media, Inc. (2012)
25. Wadler, P.: Monads for functional programming. In: International School on Advanced Functional Programming. pp. 24–52. Springer (1995)