Post-Design Domain-Specific Language Embedding: a Case Study in the Software Engineering Domain

Anthony M. Sloane
Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
asloane@comp.mq.edu.au

Abstract

Experiences are presented from a new case study of embedding domain-specific languages in the lazy functional language Haskell. The domain languages come from the Odin software build system. Thus, in contrast to most previous embedding projects, a design and implementation of the domain languages existed when the project began. Consequently, the design could not be varied to suit the target language and it was possible to evaluate the success or otherwise of the embedding process in more detail than if the languages were designed from scratch. Experiences were mostly positive. The embedded implementation is significantly smaller than its Odin equivalent. Many benefits are obtained from having the full power of an expressive programming language available to the domain programmer. The project also demonstrates in a practical software engineering setting the utility of modern functional programming techniques such as lazy evaluation and monads for structuring programs. On the down side, the efficiency of the embedded version compares unfavourably to the original system.

1. Introduction

Embedding domain-specific languages in functional languages has received much recent attention [9, 10]. A number of successful domain-specific languages have been developed in this fashion, notably Fran for graphical animation [3, 5], Pan for image manipulation [4], Frob for robot control [16, 17], and languages in the financial modelling domain [18]. All of these projects have been based around the Haskell lazy functional programming language [11].

Generally speaking, previous projects in this area have had the freedom of designing the domain specific language at the same time as implementing it via embedding. While the projects have been successful, there is a sense in which they don't present the whole story about the suitability of languages like Haskell for embedding. Because the language design is being developed at the same time, alterations can be made to fit it to the target language. There is usually no existing implementation with which to compare the embedded implementation. In contrast, if the domain-specific language has already been designed before embedding is attempted and was previously implemented via an approach other than embedding, we can assume that the design reflects the designer's view of the domain without influence from any particular target language.

One domain where embedding has been applied using language designs that already exist is language processing. Quite a bit of work has been done to express lexical analysis and parsing problems as combinator programs in functional languages [7, 12, 13]. The aim is to write expressions using a syntax based on the well-known regular expression and context-free grammar notations for expressing lexical and syntactic properties of languages. Generally speaking this aim has been achieved fairly easily, but it has proven more difficult to emulate the behaviour of language processor generators. Recently, two projects have managed to combine the combinator approach with methods used by generators [1, 19].

This paper presents a case study from the software engineering domain whose aim is similar to the aim of these language processing projects. The idea is to take existing domain specific languages and to realise them as embedded languages in a functional language (in this case Haskell). The languages are from the *Odin* software build system [2]. One is used by Odin users to request the derivation of software objects. The other language, which has some aspects in common with the request language, is used to describe the legal object types and the possible derivations between object types.

We concentrate on the instantiation of Odin in the *Eli* language processing system [8]. The user supplies specifications of language processing tasks to Eli and Eli uses Odin to control the derivation of objects such as language processor executables or generated source code. Eli and Odin constitute a large complex system (over 13Mb in an

installed binary version) and hence reproducing their capabilities is a significant test of this approach to domain-specific language implementation.

The system that resulted from the case study is called *Nowra*. As a result of building Nowra we found that the modelling capabilities of functional languages offer significant benefits to the implementation of domain-specific languages even if the design was developed without this route in mind. Nowra's code is significantly smaller than Odin's because much of its functionality is performed by the underlying Haskell system. The remaining code is much easier to understand because of the high-level nature of Haskell programs compared to C ones. The Nowra encoding of the Eli system is somewhat bigger than the Odin version but the differences are either relatively cosmetic or are avoidable with non-functional changes to Eli itself. Unfortunately, but not surprisingly, we found that the run-time performance of Nowra compares badly to that of Odin, particularly with respect to memory consumption.

The structure of the rest of the paper is as follows. Section 2 describes Odin, the concepts of objects and object types and the language an Odin user uses to request the derivation of an object. Realisation of these concepts in Nowra is discussed in Section 3. Sections 4 and 5 consider Odin's derivation graph language and Nowra's version. Finally, Section 6 discusses the Nowra design with respect to the advantages and disadvantages of using an embedding approach in general, and embedding in Haskell in particular.

2. Odin objects and the request language

The names of objects are the heart of Odin's request language. Supplying the name of an object to Odin causes it to try to bring that object "up to date". Odin will run any tools necessary to obtain a current value for the object. It caches objects so it can avoid executing tools if their inputs have not changed since the last time the object was produced. It compares object values to further optimise the derivation process, rather than just relying on timestamps. Inclusion of parameters and their values in object names means that Odin can deal with many different build variants at once.

The examples in Figure 1 give a flavour of Odin's request language. Each request is based on a *source object* that is a user file, in this case `test.c`. A *derived object* is named by appending its type to the name of the object from which it is derived. Parameters are specified using the plus operator. The greater-than operator is used to display the contents of an object or to extract it into a file. As well as simple files, objects can be directories or lists of objects.

3. Nowra's request language

Odin's request language can be syntactically embedded in Haskell in a straight-forward fashion. Objects are represented by values of type `Object`, object types are constructors of `ObjectType`, and parameters are constructors of `Parameter` (Figure 2).

`File` is used to represent source file objects. Derived objects are specified by applying the infix `:<` constructor to an object and an object type. Parameterised objects are specified by `:+` and `:+=` applied to an object and a parameter; `:+=` also takes an object which is the value of the parameter. `Lit` is used to specify literal strings; it is most useful for specifying parameter values. With these definitions the Odin objects from Figure 1 are easy to represent as Nowra objects (Figure 3).

Our design is constrained somewhat by Haskell. For example, infix constructors must begin with a colon. We use `:<` for derived objects because the more appropriate `:` is the standard Haskell list constructor. Similarly, constructor names must begin with an uppercase letter and we can't use embedded periods.

Odin's output operator `>` is accommodated using the function `>:`.

```
(>:) :: Object -> FilePath -> IO ()
```

`>:` takes the object to output and the filename of the destination file, and returns a Haskell input/output action (`IO`) that outputs the value of the object and returns no result (indicated by the unit type `()`). The action can be performed by evaluating it at the top-level of the Haskell interpreter. Again we are constrained by Haskell in our choice of names for this function; non-constructors cannot begin with a colon, hence we cannot use the more consistent `:>` (although, admittedly, there is a nice symmetry between `:<` and `>:`). Haskell does not allow us to declare postfix functions so we can't define a version of `>:` that is equivalent to Odin's `>` with no output filename. Instead, we adopt the convention that if the file path is empty it means standard output, so `>:""` has the desired effect.

4. Odin's derivation graph

An Odin *derivation graph* is essentially a description of the object types and parameters that an instance of Odin understands, and how the object types relate to each other. Derivation graphs are divided into *packages* which contain related object types and relationships. Many users can get by with the default Odin derivation graph packages which cover tasks including compilation of C programs, creation of libraries, and formatting of documentation. However, power users (such as the Eli developers) need to write new packages to define object types and specify processing steps. In fact, Eli is just an instantiation of Odin with

```
test.c :exe                # Compiled executable
test.c :exe :symbols       # Symbols defined in executable
test.c :exe >test          # Executable to file "test"
test.c :exe :symbols>      # Display symbols
test.c +debug :exe         # Executable for debugging
test.c +opt=2 :exe         # Optimized executable
test.c :output             # Directory containing output from run
test.c :output / results   # Output file "results"
test.c :incl.all           # List of files included by "test.c"
test.c :incl.all :names>   # Display names of include files
test.c :output :list       # List of output files
```

Figure 1. Odin request language examples.

```
data Object = File FilePath                    -- source file
            | Object :< ObjectType             -- x:ot
            | Object :+ Parameter              -- x +foo
            | Object :+= (Parameter,Object)    -- x +foo=val
            | Lit String                       -- literal string
            | Object :/ String                 -- selection from a directory
            | ...                              -- see Figure 5

data ObjectType = Exe | InclAll | List | Names | Symbols | ...
data Parameter = Debug | Opt | ...
```

Figure 2. Haskell data types representing user-level Nowra objects, object types and parameters.

a collection of 43 packages dealing with object types and tools peculiar to the language processor generation domain. Examples in this section are taken from Eli's command-line processing (CLP) package. An excerpt from the CLP package derivation graph is shown in Figure 4.

The first line of the excerpt declares that files whose names match the pattern "*.clp" are to be regarded as source objects of type `clp`.

An object type is declared by specifying its name, a documentation string and its direct supertypes. The types `FILE`, `LIST`, and `DERIVED-DIRECTORY` are predefined Odin types. Hence, `clp` objects are files, `clp.cmpd` objects are lists, and `clp_gen` objects are directories. Both `one.clp.cmpd` and `ext.clp.cmpd` are declared to be subtypes of `clp.cmpd`, so objects of those types are lists too.

The derivation graph describes how objects can be produced from other objects by *tools*. There are *internal tools* (ones that are predefined by Odin) and *external tools* (programs or scripts that exist outside Odin). The most commonly used internal tool is `COLLECT` that can be applied to any number of input objects and produces a list of those objects. For example, an object of type `one.clp.cmpd` can be made from an object of type `clp` by collecting the input object into a (singleton) list. Similarly, an `ext.clp.cmpd` can be made from a list by extracting the list elements that are subtypes of `clp`. The derivation `:extract=:clp` applies the *second-order object type* `extract` to the input list with the type `clp` as an argument. Among Odin's other second order object types are ones for deleting elements from a list (`delete`), applying an object type to each element of a list (`(map)`, and recursively applying an object type to a list (`recurse`).

External tools are specified using the internal `EXEC` tool. The name of the external tool to invoke is given, along with any command-line arguments. Odin arranges for the tool to be invoked in an empty working directory where the outputs can be created. For example, a `clp_gen` object can be derived by running the script `clp_gen.sh` with command-line arguments "." (the package directory) and the result of deriving `clp.cmpd :cpp` from the input object of the `clp_gen` derivation. (`cpp` applies the C pre-processor to each file in its input list and concatenates the result.) The script is expected to produce a directory called `clp_gen` in the working directory because `clp_gen` is a subtype of `DERIVED-DIRECTORY`.

The derivation graph also contains declarations of parameters. For example, a package `cc` for C program compilation might include parameters for specifying a debugging compilation and for listing pre-processor symbols that should be defined.

```
+debug 'Debugging flag' => :first;
```

```
File "test.c" :<Exe
File "test.c" :<Exe :<Symbols
File "test.c" :<Exe >:"test"
File "test.c" :<Exe :<Symbols >:""
File "test.c" :+Debug :<Exe
File "test.c" :+=(Opt,Lit "2") :<Exe
```

```
File "test.c" :<Output
File "test.c" :<Output :/ "results"
File "test.c" :<InclAll
File "test.c" :<InclAll <:Names >:""
File "test.c" :<Output <:List
```

Figure 3. Examples from Figure 1 in Nowra syntax.

```
*.clp => :clp;

:clp 'CLP specification' => :FILE;
:clp.cmpd 'Set of CLPs' => :LIST;
:one.clp.cmpd 'Singleton CLP' => :clp.cmpd;
:ext.clp.cmpd 'Extracted CLPs' => :clp.cmpd;
:clp_gen 'Objects generated from CLPs' => :DERIVED-DIRECTORY;

COLLECT (:clp) => (:one.clp.cmpd);
COLLECT (:LIST :extract=:clp) => (:ext.clp.cmpd);
EXEC (clp_gen.sh) (.) (:clp.cmpd :cpp) => (:clp_gen);
```

Figure 4. Part of the CLP package derivation graph from Eli.

```
+define 'Defined symbol' => :cat;
```

When processing a derivation involving parameters, Odin collects the parameter values as specified by the user into a list, applies the object type from the parameter declaration, and passes the resulting object to the parameterised derivation. The predefined types `first` and `cat` return the first element of a list and concatenate a list, respectively.

The declaration of an external tool can specify parameter values to be inputs. For example, we might use the following declaration to specify how to produce an object file (type `c.o`) from a C file.

```
EXEC (c.o.sh) (:c) (+debug) (+define)
  => (:c.o);
```

Three arguments will be passed to the script `c.o.sh`: the name of the C file, the value of the `debug` parameter, and the value of the `define` parameter.

5. Nowra's derivation graph

In Nowra the main part of the derivation graph is encoded in an evaluation function `eval`.

```
eval :: Object -> Eval Object
```

A value of type `Eval Object` is a computation that evaluates an object given the current state of the system and returns the result of the evaluation (either an evaluated object or a failure message) with the new state of the system.

`eval` performs evaluation of atomic objects such as files, lists and strings. The `Object` type is extended with constructors for evaluated objects as shown in Figure 5. Evaluated file, directory, and list objects are represented by `CachedFile`, `CachedDir`, and `Objs` objects, respectively. They also carry their parameter values.

`eval` also handles selection of files from directories, second-order derivations (expressed with the `:=` constructor) and derivations based on the sub-typing relation defined by the derivation graph. For the latter it uses a function `subtypes` that encodes the relation. For example,

```
subtypes ClpCmpd => [OneClpCmpd,
                     ExtClpCmpd]
```

5.1. Evaluating derived objects

Derived objects are evaluated by dispatching to an evaluation function defined by the appropriate derivation graph package. For example, the CLP package defines the function `clp` to handle derivations to the CLP object types.

```
clp :: Object -> ObjectType ->
         Eval Object
```

`clp` has one clause per object type that it can handle. For example, the clause for `OneClpCmpd` is

```
clp x OneClpCmpd = collect [x :<Clp]
```

where `collect` is a function that implements the functionality of Odin's `COLLECT` internal tool. This definition closely matches the corresponding Odin derivation graph declaration from Figure 4; the main difference is the inclusion of the object `x` which is implicit in the Odin version. Evaluation of `ExtClpCmpd` can be defined similarly.

```
data Object = ...                                  -- see Figure 2
            | CachedFile FilePath Parameters       -- a file in the cache
            | CachedDir FilePath Parameters        -- a directory in the cache
            | Objs [Object] Parameters             -- a list of objects
            | Object := (ObjectType2,ObjectType)   -- second order x ot2=ot
            | Object :? Parameter                  -- a parameter value

data ObjectType2 = Delete | Extract | Map | Recurse | ...
```

Figure 5. Nowra objects representing values, parameter access and second-order derivations.

```
clp x ExtClpCmpd =
  collect [x :<LIST :=(Extract,Clp)]
```

`ClpCmpd` is harder because we must take into account the subtyping relationships. We must evaluate the input object and see whether it's a file or a list. If it's a file then `OneClpCmpd` is applicable; if it's a list then `ExtClpCmpd` is used. (Readers unfamiliar with Haskell's `do` syntax should just interpret this code in an imperative fashion.) `dfail` is a Nowra utility function that returns an evaluation failure indicating that the requested object cannot be derived.

```
clp x ClpCmpd =
  do sx <- eval x
     case sx of
       x'@(CachedFile _ _) ->
         clp x' OneClpCmpd
       x'@(Objs _ _) ->
         clp x' ExtClpCmpd
       otherwise ->
         dfail x ClpCmpd
```

Many Eli packages define a set of "compound" object types analogous to those in the CLP package. Instead of specifying similar subtyping rules in each package, we can take advantage of this similarity of structure by abstracting the previous three definitions into Haskell functions `onecmpd`, `extcmpd`, and `compound`. For example,

```
extcmpd x ot =
  collect [x :<LIST :=(Extract,ot)]
```

We can now rewrite our definitions in a more abstract way using these functions.

```
clp x OneClpCmpd = onecmpd x Clp
clp x ExtClpCmpd = extcmpd x Clp
clp x ClpCmpd =
  compound x ClpCmpd OneClpCmpd
    ExtClpCmpd
```

Finally, we must define how to evaluate the `Clp_Gen` object type. We use a function `exec` which abstracts the functionality of Odin's `EXEC` internal tool.

```
clp x Clp_Gen =
  exec "clp/clp_gen.sh"
    [pkg "clp", x :<ClpCmpd :<Cpp]
    "clp_gen" Clp_Gen
```

The arguments to `exec` correspond fairly closely to those in the corresponding `EXEC` declaration in Figure 4. There is the external tool to run and a list of arguments. The Nowra utility function `pkg` is used to produce a directory object for the location of the CLP package. The last two parameters are the name of the output file created by the external tool and the Nowra object type. The latter is used with the subtyping relationships to determine whether the output of the tool is a file or a directory so that the appropriate object can be returned.

5.2. Parameter access

Parameter access is specified using the `:?` constructor. Hence Odin's `(+debug)` is expressed as `:?Debug`. Note that Odin overloads the `+debug` notation to mean both parameter specification and parameter access since the former is only used in user requests and the latter is only used in the derivation graph. We prefer to distinguish between the different operations so we can use them together. The `c.o` derivation step from the last section is defined as follows.

```
cc x C_O =
  exec "cc/c.o.sh"
    [x :<C, x :?Debug, x :?Define]
    "c.o" C_O
```

eval uses the function `paramtype` to determine how to obtain parameter values. It returns the object type to which the list of parameter values should be derived. For example,

```
paramtype Debug  = First
paramtype Define = Cat
```

5.3. Multiple tool outputs

Odin's `EXEC` tool can produce more than one output from a single external tool evaluation. For example, the step that runs the frontend of Eli's lexical analyser generator GLA is defined by the following derivation graph fragment.

```
EXEC (gla_fe.sh) (.)
    (:scan_spec :list :cpp)
  => (:flex_spec) (:back_data);
```

Two outputs are produced: `flex_spec` and `back_data`.

Nowra's evaluation model assumes that an evaluation yields a single object. We accommodate multiple object outputs by introducing a new built-in type called `COLLECTION`, that is a subtype of `DERIVED-DIRECTORY`. If a tool returns an object that is a subtype of `COLLECTION`, Nowra returns the working directory in which the tool was executed. The outputs of the tool can then be selected from that directory. In Nowra the relevant GLA derivation graph definitions are

```
gla x Gla_FeDir =
  exec "gla/gla_fe.sh"
    [pkg "gla",
     x :<Scan_Spec :<List :<Cpp]
    "gla_fe" Gla_FeDir
gla x Flex_Spec =
  eval (x :<Gla_FeDir :/ "flex_spec")
gla x Back_Data =
  eval (x :<Gla_FeDir :/ "back_data")
```

where `Gla_FeDir` is a subtype of `COLLECTION`.

6. Discussion

The Nowra prototype as it currently stands does not implement all of the functionality of Odin. In particular, Odin is able to run multiple derivation steps at once and distribute derivations to remote machines. Nowra can only run a single derivation at a time on a single machine. Nowra also does not implement value-based caching, features for controlling which objects are brought up to date before tool executions, or the ability to ignore error status on input objects.

Nevertheless, Nowra is functional enough to specify the Eli derivation graph packages and can execute normal Eli requests. In this mode Nowra just replaces Odin and uses the external tools, scripts and files of Eli almost unchanged. The only changes were to Odin derivation expressions that occur in Eli files; these were changed to use Nowra's syntax. We have used the system to successfully derive executables and generate source code for the specifications normally used to test an Eli installation. In each case the Nowra-based Eli produces the same code as the Odin-based Eli.

The rest of this section discusses Nowra with a view to the pros and cons of using Haskell as an embedding target for this project.

6.1. Syntax and specification

We were able to get pretty close to Odin's request language and derivation graph syntax in Nowra. Haskell's ability to define infix data constructors was important. Restrictions on the form of constructor and function names bit somewhat, but not greatly enough to obscure the similarities. (Actually, it was somewhat fortunate that colons feature prominently in Odin syntax and are required to begin Haskell constructor names.)

It's important to remember that Odin has a fixed syntax whereas Nowra's is easily extensible at the Haskell level. This difference impacts in a number of specific ways. For example, Odin tools like `COLLECT` and `EXEC` are built into the system, whereas their Nowra equivalents are just Haskell functions. Hence they can be used at the interactive prompt, instead of just in derivation graph fragments. Having the whole language available interactively makes testing easier. In Odin, uses of `COLLECT`, `EXEC`, and so on must be compiled by the derivation graph compiler and Odin must be restarted before they can be tested. Similarly, in Nowra we can write requests that access parameter values or perform second-order derivations; in Odin these features can only be used in the derivation graph.

A Nowra user can write their own functions to implement new tools if they have the need. All they have to do is fit in with the interface assumed by `eval`. Extension via this method is much easier than understanding and modifying the Odin implementation to add new functionality. Similarly, while Odin allows user-defined first-order object types, it does not allow definition of second-order object types; we are stuck with the ones provided. In Nowra it is easy to add new second-order types and write an evaluation clause to handle them. We also have more flexibility when it comes to defining first-order object types, because Nowra makes the representation of objects available. For example, Odin supplies `First` to return the first object in a list, but no way to return a list of all but the first object. It is easy to define such operations in Nowra by lifting Haskell list operations to Nowra lists.

We can also use functions to define shorthands for commonly occurring situations in a specific use of Nowra. For example, in Section 4 we saw how the Eli CLP package uses `Clp`, `OneClpCmpd`, `ExtClpCmpd` and `ClpCmpd` to assemble "compound" CLP specifications from either a single file or from a list of files. This is one instance of an idiom that is repeated more than a dozen times in other Eli packages. We showed how using user-defined functions can simplify these declarations, but we can go one step further. Observe that the object types `OneClpCmpd` and `ExtClpCmpd` really only exist because of the particular design of Odin's derivation graph language. Specifically, there is no simple way to say that we want `ClpCmpd` to form a singleton list if the input is a single `Clp` object, or to extract the `Clp` objects from an input list. We are forced to invent the somewhat artificial `OneClpCmpd` and `ExtClpCmpd` types and to use subtyping to express this processing. Because Nowra makes the subtyping relation available programmatically we could define a single

primitive for handling compound objects and do away with `OneClpCmpd` and `ExtClpCmpd`. Similar savings can be achieved with other idioms in the Eli derivation graph. A more abstract derivation graph description results.

Having a full language available also means that we can easily implement features in Odin's equivalent of Makefiles [6]. For example, Odin allows definitions like

```
test == test.c +debug :exe
```

to be placed in an Odinfile. The request `test` is then equivalent to `test.c +debug :exe >test`. In Nowra since requests are just values we can use Haskell's normal binding mechanisms to create shorthands of this kind.

In summary, because we have the full definitional power of a programming language *and* we can use the domain notations to express programs in that language, we have a fundamental advantage over an alternative where we can only extend the system by pushing down to a lower-level implementation language.

6.2. Implementation issues

The most obvious effect of using Haskell for the Nowra implementation is that we can use the Haskell interpreter instead of having to develop a language implementation from scratch. Less obviously, another advantage of embedding is that the domain-specific language can be used in the implementation of the system. For example, in the evaluation of parameter accesses we need to apply the parameter type to the list of parameter values to obtain a single object. We can express this in the evaluation function using the `:<` constructor. Thus the implementation is much more abstract than one where operations must be expressed in a lower-level implementation language.

Some specific features of Haskell play an important role in Nowra. For example, the module system gives us proper control over information hiding compared to Odin's package mechanism where derivation graph fragments are physically separated but logically declare object types in a global name space. On the downside, our approach to modularising the evaluation function by dispatching to individual package functions is not ideal since we are required to dispatch from a central location. Similarly, we must declare all of the object types and parameters in one place. The system would be more modular if there was a way to extend the definitions of functions and data types in different modules.

Lazy evaluation also plays a useful role in the Nowra implementation. In Haskell we automatically get the benefit of only having things evaluated when they are needed. For a build system that is trying to minimise work, this is a significant advantage. We can generally code the implementation in the most convenient way and leave the minimisation up to the language implementation.

Apart from the input/output involved with tool invocation, Nowra's other major input/output requirement is to be able to read and write the cache, and to read and write objects by name. Because all of the data types involved in the definition of the cache are instances of the Haskell class `Show`, the Haskell implementation is able to automatically produce input and output functions for the `Cache` data type. Thus reading and writing the cache is coded in a few lines. Derived input and output is also very useful for processing object names. Odin is able to display the name of any object and its `list` object type reads names from files to create the objects in a list. In Nowra we can display the name of an object using the derived output function for the `Object` type so no coding is required. The hitch is that we cannot use the derived input function for the `Object` type because some of the constructors are left recursive which makes the derived input function go into an infinite loop. To get around this problem we wrote a simple reader for objects, effectively making the same transformation that is performed on left recursive productions when defining top-down parsers [22].

The requirement that we be able to read and write objects by name rules out some alternative implementation approaches. For example, it is tempting to consider using functions to represent object types and implementing derivation by function application. A significant problem with this approach is that Haskell is not able to treat programs as data in the way that languages like Lisp can. Hence we can't read or write an expression involving a function. An exception in Haskell is applications of constructor functions which can be read and written and leads to the approach we have adopted.

We use monads to structure Nowra's implementation. Monads are a structuring mechanism for functional programs based on concepts from category theory [20, 21]. They have been shown to be useful for expressing side-effecting computations in a functional setting and structuring programming language implementations [14, 15]. Nowra's `Eval` type uses a combination of state and error monad techniques to thread the cache state and error messages through evaluations without having to pass explicit arguments. We also use the standard Haskell `IO` monad to encapsulate computations that have side-effects such as reading files or running external tools. The resulting implementation is significantly less complex than it would have been without monads. We were able to use a highly expressive functional style in much of Nowra but precisely control evaluation in an imperative style in core areas. Given that we had very little experience with monads before this project began, we were pleased with how easy they were to apply in quite a practical setting.

6.3. Empirical comparison

Even though the Nowra version of Eli is not functionally equivalent to the Odin version, it implements a large fraction of the latter's capabilities. Hence an empirical comparison of their implementations, while not completely valid, has some legitimate basis.

The Odin implementation contains much more code than the Nowra implementation. The Odin interpreter that evaluates requests, manages the cache and so on, contains more than 20,000 lines of C code and there is a derivation graph compiler that is another 6,000 lines. This code is the portion of the system that is fixed for all domains to which Odin is applied. The equivalent code in Nowra is just under 1,000 lines of Haskell. We haven't conducted a detailed comparison, but the most significant savings appear to be due to the lack of an interpreter, and in the code for reading and writing expressions and the cache. Other savings are due to having lists as a basic data type, not representing the derivation graph as a data structure, and the generally more abstract nature of Haskell programs compared to C programs.

The derivation graph constitutes the portion of the system that changes from domain to domain. For the language processor generation domain targetted by Eli the derivation graph in Odin notation totals around 1,000 lines. The same derivation graph in Nowra's notations is about 1,700 lines of Haskell code. While it would seem that the Nowra version is significantly more verbose, the two versions are actually very similar. Most of the differences are due to minor formatting issues and the fact that the Nowra derivation graph is structured as modules so import declarations are required to make cross-package dependences explicit. Other differences include the introduction of extra clauses to specify multiple outputs from a tool (as described in Section 5.3), explicit object references in the Nowra derivation graph as opposed to Odin's implicit objects, and Nowra's `EXEC` tool that has more arguments than it strictly needs because we didn't want to alter the Eli tools to accommodate the differences between Odin and Haskell naming conventions.

Runtime measurements were conducted with a version of Nowra compiled by the Glasgow Haskell Compiler (version 5.00.1) using the `-O` option. The machine is a IBM compatible PC with 128Mb of memory running Linux Redhat 7.0 and kernel version 2.2.19. The compiled Nowra modules were loaded into the interactive version of the Glasgow compiler. We used Nowra and Odin to drive the Eli packages working on an Eli compiler specification containing lexical analysis, parsing and attribute grammar components. The specification is written in a literate programming style and makes moderate use of Eli library modules. Hence the specification exercises most of the packages in the Eli system.

Odin takes about two minutes to generate an executable for the compiler on a lightly loaded machine. In contrast, Nowra takes more than five minutes. An even bigger difference is apparent when memory consumption is examined. Odin starts out with a process size of less than a megabyte and rises to around three megabytes. The Haskell interpreter process with the Nowra modules loaded is around 70Mb in size. During derivation of the compiler its size rises to over 110Mb. The Odin cache consumes about 3Mb after the compiler has been generated and Nowra's ends up at a similar size.

The cache size performance of Nowra is good, particularly considering that the cache is output in a text form rather than in binary. Nowra doesn't currently implement value-based caching, however; we can expect the cache size to increase when it does.

Nowra's overall runtime performance leaves something to be desired, particularly as far as memory consumption goes. The measurements reported here constitute a worst case since we have made no effort to optimise Nowra. It is likely that significant improvements can be made with relatively little effort. Of course we will probably never approach the performance of the C version.

7. Conclusion

The Nowra design constitutes just one data point in the space of possible Odin re-implementations. Undoubtedly, other developers would come up with different designs. Similarly, Odin's domain-specific languages are just two among many. Experiences applying Haskell to other languages will obviously be different. Nevertheless, some general conclusions seem to be warranted since this case study was able to consider questions not possible in most previous work in embedding domain-specific languages in functional languages, because the language design implementation already existed.

Specifically, since the design of Odin predates Nowra by many years, we were able to exercise the modelling capabilities of the functional language without being tempted to change the domain-specific language. Of course, we had to react to some constraints placed on us by Haskell but the general flavour and expressiveness of Odin's languages have been retained. Because the Odin implementation already existed, we were able to evaluate our new design to demonstrate the benefits and drawbacks of the embedding approach. We saw that we were able to implement Nowra in a relatively small amount of code compared to the Odin implementation, making good use of many features not present in C. On the other hand, we saw that the runtime performance of Haskell for this application is significantly worse than the C implementation. We also would have liked the ability to extend function and data type definitions in different modules to achieve proper modularity.

Acknowledgements Thanks to Paul Hudak and the Department of Computer Science at Yale University who hosted the author's sabbatical leave during which this work was performed. Some of the derivations in Section 2 are based on examples in the Odin manual by Geoff Clemm. The anonymous referees made many useful suggestions that improved the presentation of this work.

References

[1] M. M. T. Chakravarty. Lazy lexing is fast. In *Proceedings of the Fourth Fuji International Symposium on Functional and Logic Programming*, number 1722 in Lecture Notes in Computer Science, pages 68–84. Springer-Verlag, 1999.

[2] G. Clemm and L. Osterweil. A mechanism for environment integration. *ACM Trans. on Programming Languages and Systems*, 12(1):1–25, Jan. 1990.

[3] C. Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Trans. on Software Engineering*, 25(3):291–308, May/June 1999.

[4] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *Proceedings of the Semantics, Applications and Implementation of Program Generation Workshop*, number 1924 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the International Conference on Functional Programming*, pages 163–173. ACM Press, June 1997.

[6] S. I. Feldman. Make—a program for maintaining computer programs. *Software — Practice and Experience*, 9:255–265, 1979.

[7] J. Fokker. Functional parsers. In *Proceedings of the First International Spring School on Advanced Functional Programming Techniques*, number 925 in Lecture Notes in Computer Science, pages 1–23. Springer-Verlag, 1995.

[8] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, Feb. 1992.

[9] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, June 1998.

[10] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[11] P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[12] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[13] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[14] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Proceedings of the European Symposium on Programming*, number 1058 in Lecture Notes in Computer Science. Springer-Verlag, Apr. 1996.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 333–343. ACM Press, 1995.

[16] J. Peterson, G. D. Hager, and P. Hudak. A language for declarative robot programming. In *Proceedings of the International Conference on Robotics and Automation*. IEEE, May 1999.

[17] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: controlling robots with Haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, number 1551 in Lecture Notes in Computer Science, Jan. 1999.

[18] S. Peyton-Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering. In *Proceedings of the Fifth International Conference on Functional Programming*, pages 280–292. ACM Press, Sept. 2000.

[19] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In *Proceedings of the Second International School on Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science, pages 184–207. Springer-Verlag, 1996.

[20] P. Wadler. The essence of functional programming. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1992.

[21] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, Proceedings of the Bøastad Spring School*, number 925 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[22] W. M. Waite and L. R. Carter. *An Introduction to Compiler Construction*. HarperCollins, New York, 1993.