Optimising Embedded DSLs using Template Haskell

Sean Seefried, Manuel Chakravarty, and Gabriele Keller

PLC Research Group
The University of New South Wales, Sydney
{sseefried, chak, keller}@cse.unsw.edu.au
National ICT Australia**, ERTOS.

Abstract. Embedded domain specific languages (EDSLs) provide a specialised language for a particular application area while harnessing the infrastructure of an existing general purpose programming language. The reduction in implementation costs that results from this approach comes at a price: the EDSL often compiles to inefficient code since the host language's compiler only optimises at the level of host language constructs. The paper presents an approach to solving this problem based on compile-time meta-programming which retains the simplicity of the embedded approach. We use PanTHeon, our implementation of an existing EDSL for image synthesis to demonstrate the benefits and drawbacks of this approach. Furthermore, we suggest potential improvements to Template Haskell, the meta-programming framework we are using, which would greatly improve its applicability to this kind of task.

1 Introduction

Domain Specific Languages (DSLs) reduce the cost of producing software by providing programming constructs tailored for a particular domain. This reduces the amount of repetitive code that would otherwise be written in a general purpose language and also means that people that have little programming experience, but are nevertheless conversant in the domain can use these languages. Yet in terms of implementation effort, constructing new languages is expensive [14].

Embedded domain specific languages (EDSLs) ([7], [8]) decrease the implementation burden since they are implemented in an existing, feature-rich, general purpose language. This allows the reuse of a substantial portion of the host language's programming environment, such as the lexical analyser, parser, type checker, optimisation phases and code generator of the compiler and the tools surrounding it such as debuggers and profilers. Some host languages are better choices than others: in this paper, we argue that a language with support for compile-time meta-programming is an ideal tool for the implementation of an EDSL due to their ability to express compiler-like optimisations, thus increasing the number of domain specific optimisations that can be written.

** National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

The standard approach to the construction of EDSLs involves implementing them as libraries of combinators in a language with support for higher order functions, a rich type system and good syntactic control mechanisms[1]. Unfortunately, EDSLs constructed in this manner often produce inefficient code.

The reason behind this is that domain data types are usually represented as algebraic data types and are *interpreted* by recursive traversal functions ([3], [9], [15], [18]). This interpretive overhead is present in the generated code as the host language compiler has no knowledge of code improvement techniques that may be applied to domain data types and expressions which use them. What is absent is the ability to declare compiler-like optimisations that operate on the syntactic structure of expressions: a capability that *is* offered by a language with compile-time meta-programming features.

An alternative is to *embed a compiler* [5] for the DSL, rather than the DSL itself, in the host language. This was precisely the approach taken by the implementors of Pan, a language for the synthesis of two-dimensional images and animations [4]. In this approach the primitives of the DSL are defined as functions over an abstract syntax tree (AST) representation. The ASTs generated by programs written in the host language are then optimised and fed to a code generator which produces efficient code in a (not necessarily different) target language. The effort involved is equivalent to writing a compiler back-end and although it is considerable, the cost of writing the components of a compiler front-end (such as a lexical analyser, parser, and type checker) is saved.

Approach	Inherit front-end	Inherit back-end	Optimise via
Embedded compiler	yes	no	traditional compiler opts.
Staged interpreter	no	yes	MP: delayed expressions
Extensional meta-programming	yes	yes	MP: transformation (requires code *inspection*)

MP = meta-programming

Fig. 1. A comparison of three approaches to implementing DSLs

The main disadvantage to embedding a compiler is that access to the (often extensive) general optimisations of the host language compiler are lost. Furthermore, if there is disparity between the host language and the target language, generated programs may not be able to use features of the host language. Finally, even though one ostensibly writes programs in the host language it may not be possible to use language constructs which require a base type of the language. (For instance, for *if-then-else* expressions to be valid they typically require an expression of boolean type.) Unfortunately, the representation of expressions

[1] This has the additional benefit of easing their distribution; it is much easier to distribute and build a module written in a well established language than the sources for an entire compiler.

as ASTs requires the use of synthetic types, which precludes the use of such language constructs.

These disadvantages are automatically avoided by the traditional approach to EDSL implementation (of not embedding a compiler back-end). We assert that the problem of inefficient code is adequately solved by extending this approach with the techniques provided by compile-time meta-programming. Our approach, which we have dubbed *extensional meta-programming* transforms user-written code according to its syntactic structure. The key idea is to represent code as a data structure (preferably an abstract syntax tree), manipulate this data so that it represents equivalent but faster code, and finally turn this data back into code.

Extensional meta-programming differs quite markedly from another popular meta-programming approach: staged interpreters [2]. Staged interpreters use meta-programming annotations to traverse the representation of the interpreted program before the essence of the program is executed at run-time and amounts to a form of domain specific partial evaluation. This approach also inherits the optimisations of the host language (as they will be applied to the generated code) but introduces the expense of having to implement the front-end of a compiler. If this was merely restricted the implementation of a simple lexer and parser this expense would be acceptable. Unfortunately real languages often require significant front-end infrastructure such as a symbol table, a complicated abstract syntax tree representation and analysis phases.

One advantage of the staged interpreter approach is that the meta-language does not require the ability to inspect the structure of code. MetaML and MetaOCaml both lack this ability. We have summarised the differences between embedded compilers, staged interpreters and extensional meta-programming in Figure 1.

Since our primary motivation is to increase reuse in order to reduce the effort required to create new languages we have chosen extensional meta-programming. To demonstrate the feasibility of our approach, we use Template Haskell[16], a compile-time meta-programming extension to the Glasgow Haskell Compiler [1] to provide an alternative implementation of Pan [4], which we have dubbed PanTHeon. Pan was implemented as an embedded compiler in Haskell, its target language being C. As alluded to earlier, the disparity between these two languages means that the generated programs cannot use features of the host language: in this case laziness and higher-order functions.

To summarise, the main contributions of this paper are:

1. We introduce an approach, extensional meta-programming, to implementing EDSLs offering the following benefits over an embedded compiler:
   – Reduced implementation effort through the sharing of host language's programming environment and the *extension* of its suite of optimisations.
   – Retention of host language features.
   – Inheritance of host language constructs.
2. We investigate the feasibility of extensional meta-programming using PanTHeon as an example.

3. Template Haskell is evaluated on an example of significant size, and suggested extensions to Template Haskell that will make a wider range of EDSL implementation tasks easier.

The rest of the paper is organised as follows. First, we provide an introduction to meta-programming through Template Haskell. We then give a brief overview of PanTHeon. This comprises a brief introduction to the Pan language and then a description of the transformations we apply to it. Each description is divided into a language independent description (lest readers come to the conclusion the solution is specific to Template Haskell) and one that focuses on the implementation details. Next we present benchmarks that provide evidence of the efficacy of our optimisations followed by an analysis of recurrent problems we have had with Template Haskell and any solutions we devised.

2 An introduction to compile-time meta-programming via Template Haskell

This section shows how a useful algebraic transformation can be implemented in Template Haskell. Since the optimisations of PanTHeon are treated more than adequately in the rest of the paper we focus on a useful transformation in a well-known domain: linear algebra.

A basic result of linear algebra is that an $n \times n$ matrix, M, multiplied by its inverse, M^{-1}, is equal to the identity matrix, I. This is just the sort of property that we cannot expect most compilers to optimise away, due to the domain-specific knowledge that is required to perform such an optimisation. Consider an expression `m * inverse m` where `m` and `n` are matrices. (The precise details of how matrices are implemented is immaterial.) In order that this expression may be simplified it must first be converted from code into a data structure via a process known as *reification* [6].

Once we have verified that this data structure matches the pattern *m* `* inverse` *m* we can replace it with the data structure that represents `identity`. We then need to convert the data structure back into code, via a process known as *reflection* [17]. This is also known, particularly in Template Haskell, as *splicing.*

In most meta-programming languages, the reification of `m * inverse m * n` will take the form of an abstract syntax tree. The transformation of this expression is then simple. We create a new data structure which represents `identity * n` and splice it. We now show how this is achieved using Template Haskell on a (rather contrived) lambda expression with body equal to `m * inverse m * n`.

```
exp_mat = [| \m n -> m * inverse m  * n |]
```

`exp_mat` makes use of the quasi-quote notation of Template Haskell, denoted by the `[|` and `|]` brackets. These brackets reify code within.

Figure 2 presents the function `rmMatByInverse` which removes the redundancy in the reified expression. Unfortunately, without familiarity with the data structure used for representing expressions the code can be difficult to understand.

The first case does the real work; it matches against infix expressions of the form $m *$ `inverse` m and returns `identity`, while the second and third (after matching against expressions of the form $\lambda p.e$ and $f a$ respectively) recursively call upon sub-expressions. (Note that we have only presented the cases necessary to transform `exp_mat`.)

```
rmMatByInverse (InfixE (Just 'm) 'GHC.Num.* (Just (AppE 'inverse 'm))) =
  VarE (mkName "identity")
rmMatByInverse (LamE pats exp) = LamE pats (rmMatByInverse exp)
rmMatByInverse (AppE exp exp') =
  AppE (rmMatByInverse exp) (rmMatByInverse exp')
rmMatByInverse exp = exp
```

Fig. 2. An example of an arithmetic transformation.

Template Haskell's splicing operator, `$(...)`, *runs* meta-programs and converts the resulting data structure back to code. In our case the expression `$(rmMatByInverse exp_mat)` evaluates to the *code* `\m n -> n` at compile-time. This is a key aspect of our approach; by using a language which is restricted to compile-time meta-computation we guarantee that there is no run-time overhead in the code generated.

3 PanTHeon

PanTHeon is a direct implementation of the image primitives presented in Elliott's paper [4]. There are three main classes of optimisation: the unboxing of arithmetic expressions, aggressive inlining and algebraic transformations. In the subsections below we describe why each is particularly applicable to our domain, and its realisation in general meta-programming terms, without recourse to Template Haskell specifics. Any language of similar functionality could be used in its place (although we know of no such language yet.) However, Template Haskell is a relatively new extension to Haskell. It is common wisdom that languages change rapidly and substantially early in their lives. This is at least partly motivated by their use in novel situations where it is discovered that additional features would simplify things. As such, we follow the general description of each optimisation with the solution we devised in Template Haskell, highlighting problems we encountered.

But first, to put all this in context, we describe Pan in more detail.

3.1 A Pan example

Pan is a domain specific language founded upon the concept of modelling an *image* as a function from continuous Cartesian coordinates to colour values. The *animation* extends the image concept; it is simply a function from continuous time to an image.

Figure 3 presents a simple Pan effect that will be used as a running example throughout the rest of the paper and is self contained with respect to Appendix A.

```
checker (x,y) = if even e then blackH else whiteH
  where e = floor x + floor y
stripes (x,y) =
  | even (floor x) = blue
  | otherwise = red
checker_on_stripes = checker `over` (empty `over` swirl stripes)
```

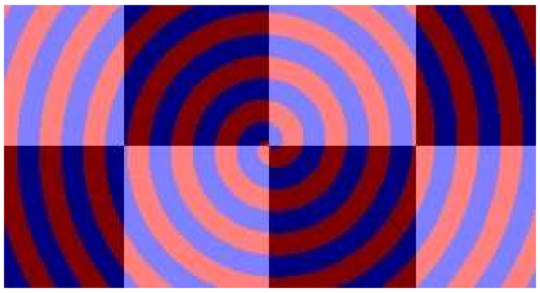

Fig. 3. Checker board imposed over swirled vertical blue and red stripes

Colours are represented as four-tuples containing red, green, blue and alpha (transparency) components in the range $[0, 1]$. `whiteH` and `blackH` are 50% transparent. The checker board (`checker`) is defined as a function which takes a coordinate (x, y) and returns `blackH` if $\lfloor x \rfloor + \lfloor y \rfloor$ is even and `whiteH` otherwise. The `stripes` function is even easier to define. Here we simply check that that $\lfloor x \rfloor$ is even and colour it blue if so, red if not.

In `checker_on_stripes` we see the use of the *image overlay* combinator, `over`. This function combines two images pointwise. Depending on the transparency of the top image a portion of the underlying image will show through. `swirl` is an interesting Pan primitive that warps an image by rotating points a distance proportional to their distance from the origin. The `empty` image is completely transparent. See Appendix A for their implementation.

3.2 Architecture of PanTHeon

PanTHeon consists of three main parts - the language implementation, the optimisation modules and a client for displaying effects. As mentioned above, the language implementation is a direct implementation of the combinators in [4]. Users write effects in Pan (which is really just Haskell) which can then be loaded directly into the client via conventional file menu widgets. The user written code is imported into an automatically generated module which transforms their code via functions present in the optimisations modules. This file is then compiled in GHC and dynamically loaded using Don Stewart's *hs-plugins* library[13].

3.3 Unboxing Arithmetic

Motivation and abstract approach PanTHeon is a numerically intensive application, almost exclusively using floating-point arithmetic. Hence unboxing can yield significant improvements in speed[2]. Unboxed code also yields better memory locality as the arguments and results do not require an indirection to a heap allocated object. In fact, it may be possible that the arguments are placed directly into registers.

Most compilers optimise away as much unnecessary boxing as is feasible, but as implementors of an EDSL we have more knowledge than the compiler does and can consequently do better. We can be certain of the validity of unboxing assuming that every function in PanTHeon is also monomorphic. Although it is quite possible to define functions this way (and we have done so) a much nicer solution would be to specialise each invocation of a polymorphic function based on the type information gleaned from the context in which it is invoked. We discuss this further in the next subsection.

This begs the question, why do we not simply define all the functions in terms of unboxed arithmetic in the first place? Apart from the fact that the syntax of unboxed arithmetic is ugly and cumbersome to use, there is a more important issue: abstraction. When a colour is displayed, each of its component values is converted to an integral value between 0 and 255 and combined into a single 32-bit integer that is placed into video memory. Efficiency can be gained by converting the functions that operate on colours to their integer arithmetic equivalents behind the scenes, while the user retains their view of the current abstraction (i.e. floats of [0,1])[3].

In general terms this optimisation requires that we traverse the representation of each top level function replacing all boxed arithmetic operators and constants with their unboxed equivalents. The unboxing of arithmetic is an interesting transformation as it changes the semantics of the program. Each type in the resulting program corresponds exactly to a type in the original, but it is clear that the validity of this correspondence relies upon our knowledge of the domain.

Implementation in Template Haskell The process of replacing all boxed operators and constants with their unboxed equivalents is generally a straightforward process in Template Haskell, although we run into difficulty in the context

[2] Without unboxing, each arithmetic function must first unbox its arguments, perform a primitive arithmetic operation upon these values, and re-box the result.

[3] Although early experiments indicated that this arithmetic conversion measurably improved performance, there were technical reasons that prevented it. We discuss the reasons in the implementation details.

of polymorphic data structures. Most cases written for the family of unboxing functions merely call `unbox` recursively on sub-objects (be they declarations, types, bodies, expressions, etc). There are only a few interesting cases:

1. Transforming type signatures. It is clear that any type signatures or type annotations that existed in the original declarations will no longer be valid. For each type synonym and data type declared for the boxed declarations we declare an unboxed version. For ease of recognition the name of such types have a `_UB` suffix appended.
2. Replacing arithmetic operators with unboxed equivalents. This code assumes that all operators will be changed to their unboxed floating point equivalents. We recognise this as a flawed assumption and we discuss this further in this paper.
3. Replacing tuples with stricter versions. We declared two new data types to express points and colours to increase strictness. Unlike the situation with tuples, one can add strictness annotations to the arguments of the constructor.

We now present an example of all three of these cases in action.

```
checker :: ImageC
checker (x,y) = if even e then blackH else whiteH
  where e = floor x + floor y
```

becomes

```
checker :: ImageC_UB
checker (Point_UB x y) =  if evenInt# e then blackH else whiteH
  where e = float2Int# x + float2Int# y
```

Our main problem with the implementation of the unboxing pass has been the lack of easily accessible typing information. It is problematic in three ways.

– It is impossible to know what the type of a literal is. This was first identified by Lynagh [12]. Fortunately, nearly all literals in the definition of Pan functions are instances of `Fractional`. However there were a few instances where this was not true and special cases had to be written for them.
– In the previous section we stated that we had also considered converting the components of colours to the range $[0, 255]$. This would have necessitated a relatively complex transformation on all functions which manipulated colours.
  For instance consider the definition of `cOver` (a key component of the definition of image overlay.)

```
cOver (r1,g1,b1,a1) (r2,g2,b2,a2) =
  (h r1 r2, h g1 g2, h b1 b2, h a1 a2)
  where h x1 x2 = a1* x1 + (1 - a1) * x2
```

  Under our proposed transformation it would become

```
cOver (Colour_UB r1 g1 b1 a1) (Colour_UB r2 g2 b2 a2) =
  (Colour_UB (h r1 r2) (h g1 g2) (h b1 b2) (h a1 a2))
  where h x1 x2 = (a1 *# x1 +# (255# -# a1) *# x2) `divInt#` 255#
```

Such a transformation is only feasible when one has knowledge of the type of each variable. For instance, in the example above it is necessary to know that `a1` is of type `ColourBit_UB` (i.e. in range $[0, 255]$).

– We have had to define all PanTHeon functions that contain arithmetic operations monomorphically. A restriction that GHC imposes is that a function containing unboxed operations cannot operate on polymorphic data types. With type information we could specialise such polymorphic functions at each call site.

Without the ability to reify the type of a fragment of an expression, some transformations simply cannot be written for the general case, and until a satisfactory solution has been found, we regard this as one of the principle shortcomings of our implementation. We discuss this issue further in Section 5.2.

3.4 Inlining

Motivation and abstract approach The style of embedding used in the original implementation of Pan has the effect of inlining all definitions and β-reducing the resulting function applications before any further simplification occurrs. This greatly increases the opportunities for algebraic transformation but has the drawback of introducing the possibility of code replication. Fortunately, the effect of code replication can be mitigated by applying a common subexpression elimination (CSE) pass following this one. Based on the success Elliott, Finne, and de Moore [5] had with it we investigated this approach to code improvement.

However, since GHC has its own passes for performing beta-reduction and CSE, we decided to leave these passes unimplemented and see how well the compiler performed. The results of our experiment are encouraging and we provide a concrete example in next section.

In general terms the ability to inline code relies upon two meta-programming facilities: the ability to reify, transform and splice code, and the ability to look up the definition of a top-level function declaration. Unfortunately, Template Haskell does not (yet) support this second facility. In Section 5.1 we explain our solution to this problem which involves the manual creation of a look-up table.

With this infrastructure in place, the inlining process is relatively straightforward. We take as input the final animation or image function that PanTHeon will display and traverse its definition. Each time we encounter the use of a function that has been defined in PanTHeon[4] we look up its definition, create an equivalent lambda expression and substitute it at that location. We do this recursively.

[4] We do not inline functions that are part of other Haskell libraries.

Clearly, this leads to non-termination in the context of recursion. While we could refuse to inline recursive function definitions, determining whether a function is recursive is an involved process requiring the construction of a call graph and the determination of strongly connected components, and in any case GHC already does this. Unfortunately we do not have access to this information. (Perhaps Template Haskell should provide it.) Instead we have chosen to limit the inlining process to a fixed depth which roughly corresponds to loop unrolling.

Implementation in Template Haskell Most definitions in the inlining transformation are concerned with traversing the components of a declaration. The function that actually does the real work is `mkInlineExp`. Its implementation is quite cluttered with Template Haskell specifics so we have chosen to present a stepwise example of its effect on our running example (introduced in Section 3.1).

The inlining pass traverses the declaration of `checker_on_stripes` until it comes to the *variable* sub-expression `checker`. At this point a look-up is performed upon its name and the declaration for `checker` is retrieved. We then convert this definition to an equivalent lambda expression. Note that *where* declarations are converted to *let* declarations. Note that without typing information we cannot inline functions that have been overloaded using Haskell's type class mechanism.

```
\(x,y) -> let e = floor x + floor y
          in  if even e then blackH else whiteH
```

This expression is then substituted in place of the variable. Function definitions that contain guards are also handled. This occurs during the inlining of `stripes`. It is replaced with

```
\(x,y) -> if even (floor x) then red else blue
```

In the previous section we promised a concrete example of the effect of GHC's common subexpression elimination on inlined code. A fitting example to consider comes from the original paper on the implementation of Pan [5]):

```
swirlP r = \p -> rotate (dist0 p * (2*pi/r)) p
```

The result of inlining clearly contains much redundancy:

```
(\(x,y) -> (x * cos (sqrt (x*x + y*y) * (2*pi/r))
            - y * sin (sqrt (x*x + y*y) * (2*pi/r)),
            y * cos (sqrt (x*x + y*y) * (2*pi/r))
            + x * sin (sqrt (x*x + y*y) * (2*pi/r)))
```

The following dump of the Core[5] code produced shows that it is capable of removing much of the redundancy.

[5] An intermediate representation used by GHC. Adding the flag -ddump-core to the command line will dump the code to standard output.

```
\w_se6i ww_se6l ww_se6m ->
let { a'334 = <core equivalent of x*x + y*y * (2*pi/r)>
} in
 (# (GHC.Prim.minusFloat#
   (GHC.Prim.timesFloat# ww_se6l (GHC.Prim.cosFloat# a'334))
   (GHC.Prim.timesFloat# ww_se6m (GHC.Prim.sinFloat# a'334))),
  (GHC.Prim.plusFloat#
   (GHC.Prim.timesFloat# ww_se6m (GHC.Prim.cosFloat# a'334))
   (GHC.Prim.timesFloat# ww_se6l (GHC.Prim.sinFloat# a'334)))
 #)
```

GHC also performs β-reduction and constant folding (e.g. 2π is replaced with the constant $6.283\ldots$) which saves us yet more implementation effort.

3.5 Algebraic Transformation

Motivation and abstract approach The principle behind algebraic transformation as an optimisation technique is simple: expressions are substituted for semantically equivalent expressions which compile to faster code, be it universally or only on average. If we consider our running example again, we can see that overlaying the entirely transparent `empty` image on top of `swirl stripes` will have no effect. (This is proved by examining the definition of `over`.)

```
checker_on_stripes = checker `over` (empty `over` swirl stripes)
```

The sub-expression may simply be replaced with `swirl stripes`. That is, the following algebraic identity holds: `empty `over`` *image* = *image*. (For more examples of algebraic properties of Pan see Appendix B.)

What is exciting about our use of this technique in PanTHeon (and in the general context of EDSLs) is that we are using it in a fairly novel context: outside the compiler. A key advantage over an embedded compiler is that we only need to implement transformations specific to our EDSL, *extending* rather than overriding the optimisations of the compiler.

In general terms algebraic transformations are easy to implement. For a given expression we attempt to match it against our known algebraic identities. When successful we replace it with the equivalent optimised expression. To ensure that sub-expressions are also optimised we recursively apply to the sub-expressions left unchanged by the original transformation. We also do this when no algebraic transformation is applicable.

Implementation in Template Haskell Template Haskell's reification of code as algebraic data types in combination with its pattern matching features make algebraic transformations very easy to write (and has been noted by others [2]). Earlier we showed that the expression `empty `over`` *image* can be replaced with *image*. This particular case is implemented via the following code.

```
algTrans (AppE (AppE (VarE 'over) (VarE 'empty)) image) =
  algTrans image
```

One of the side effects of the rich syntaxes offered by modern programming languages, including Haskell, is that there is often more than one way to write essentially the same expression. This is very useful for program *generation* but in the context of program *transformation* means that separate cases must be written to transform equivalent expressions. In order to reduce the number of patterns to be matched against, a number of cases were written that put expressions in a canonical form. For instance, the example above matches on the canonical (prefix) form, `over empty` *image*, of the algebraic identity presented earlier.

Another tedious aspect of all transformations is the recursive cases. Since we wish our transformations to be applicable not just to expressions but sub-expressions also, we must have cases which recursively call on them. These cases are numerous and easily outnumber the cases that actually do interesting work. However, a recent paper [11] presents a method by which the such boiler-plate code can be "scrapped"; that is the traversal can be done in a handful of lines of code. We have used these techniques in our source code.

An example of the code reductions are shown below. The function `inlineExp` checks whether an expression is a variable and inlines the appropriate function if so and returns the expression unchanged if not.

The function `inline` is defined using the `everywhereM` combinator. It can be used on any code representation data structure and will transform any component of such data strucutre that contains an expression, no matter how deeply nested. For lack of space we have omitted the function `mkInlinedExp` which creates an lambda expression equivalent to the looked up function definition.

```
inlineExp :: [(String, FunDecl)]
          -> ([(String, FunDecl)] -> (forall a. Data a => a -> Q a))
          -> Exp -> Q Exp

inlineExp tbl inline e@(VarE nm) =
 case lookup (nameBase nm) tbl of
  (Just (funDec, _)) -> mkInlinedExp (inline tbl) funDec
  Nothing -> return e
inlineExp _ _ exp = return exp

inline :: [(String, FunDecl)] -> (forall a. Data a => a -> Q a)
inline tbl = everywhereM (mkM (inlineExp tbl inline))
```

4 Benchmarks

Performance testing on PanTHeon has been conducted in two ways - optimised effects have been compared with their unoptimised counterparts as well as against the original Pan implementation.

4.1 PanTHeon vs. itself

Figure 4 compares the frame rate of an effect for which different combinations of optimisations have been applied. When both unboxing and inlining are applied the effects run at least twice as fast, and for one particular example the optimisations led to a nine-fold speed-up.

The effects were run on a 1Ghz Apple Powerbook G4 with 512MB of RAM. We have left out the effect of algebraic transformations only because our sample size is so small. Naturally, we could contrive an effect with much computational redundancy which would show off its effectiveness, but this would not tell us much. Only by collecting a large number of effects can we say anything about its effectiveness.

Effect	**Base**	**Inlined**	**Unboxed**	**Unboxed & Inlined**
checker_swirl	**8.86 f/s**	1.309x	2.190x	2.258x
circle	**11.241 f/s**	1.324x	2.126x	2.083x
checker_on_stripes	**1.302 f/s**	1.027x	8.361x	9.003x
four_squares	**2.512 f/s**	1.366x	4.184x	4.152x
triball	**1.244 f/s**	1.914x	2.578x	2.707x
tunnel_view	**4.62 f/s**	1.223x	2.042x	2.661x

Fig. 4. Effect of optimisations on frame rate for effects displayed at 320x200 resolution.

4.2 PanTHeon vs. Pan

How well does the performance of PanTHeon compares with that of Pan? Unfortunately, this is difficult to compare because of platform disparity. PanTHeon has been implemented for *nix[6] platforms while Pan only runs on Microsoft Windows. Nevertheless, we performed measurements on PanTHeon and Pan on the same machine: a 733 MHz Pentium III, 384 MB RAM, at 400x300 resolution.

Pan still outperforms PanTHeon. The `checker_swirl` effect has performance that compares favourably; at a resolution of 400x300 it runs at 4.78 *frames/s* in PanTHeon and at 10.61 *frames/s* in Pan. Other effects such as `triball` and `four_squares` perform far better in Pan ($>$ 18 *frames/s*) and very slowly in PanTHeon (1.68 *frames/s* and 6.54 *frames/s* respectively). Both these effects makes substantial use of the `over` primitive for layering images on top of each other. Pan seems to do substantial unrolling of expressions, which is an optimisation we have not yet implemented in PanTHeon. We suspect that it will significantly improve effects' performance.

However, we have shown that the issue does not lie with any inherent deficiencies in the quality of the code that GHC produces. We hand-coded (and optimised) a very simple effect which ran at a speed comparable to the same effect in Pan (24 million pixels/s).

Another aspect of the Template Haskell implementation that has hindered further tuning or creation of optimisations is the lack of support for profiling of programs which contain splicing.

[6] It has been successfully built on Debian GNU/Linux and Mac OS X.

4.3 Relative code base sizes

Finally, we compare the amount of code needed to implement Pan and PanTHeon as a crude means of comparing implementation effort. Both Pan and PanTHeon have two components – a language definition and a display client. The PanTHeon library, plus optimisations totals at about 3000 lines of code. The client is implemented in under 1000 lines of code. Pan, in its entirety, exceeds 13000 lines of code. The inheritance of a code generator and host language optimisations by PanTHeon is the primary reason for this difference.

5 Template Haskell specifics

We found Template Haskell to be an excellent language for extensional metaprogramming. It's quasi-quote notation, its novel approach to typing and its ability to represent and inspect code made the task of writing elegant compiler-like optimisations possible. However, we believe that there are ways in which the language could be improved further and in this section we review difficulties we had with the current Template Haskell implementation and any solutions we devised. We envisage that this section will be of most use to other users of Template Haskell and may be skipped safely by those who are not interested.

5.1 Reification of top-level functions

Both unboxing of arithmetic and inlining, require the ability to reify top-level function declarations. Currently, such reification is unsupported in Template Haskell. There is, however, a relatively simple work-around to this problem. We can create a look up table in two steps. First, we place the entire module in declaration reification brackets and call the resulting data structure something appropriate, such as `moduleImageFuns`. We can then create the look-up table for this module by applying a function which creates a list of pairs matching names to function declarations.

An interesting dilemma arises when one wishes to write a module, say `M2TH`, which refers to functions defined in module `M1TH`[7]. But the functions in `M1TH` are not in scope and will only become so if they are spliced in somewhere. So we create a module `M1` which splices in the reified functions from this module and then import `M1`, not `M1TH`, inside module `M2TH`. The basic idea is summarised in Figure 5.

There is just one more tiny problem. We wish to transform both the functions in `M1TH` and `M2TH` before bringing them into scope for display in the PanTHeon client, but the solution outlined above causes the functions in `M2TH` to refer to the functions in scope in `M1`. The reified form of such functions contain *original names* which are of the form `M:f` (where `M` and `f` are module and function names respectively). In order to refer to whatever is in scope at post-transformation splice time we must remove the module prefix from the original names.

[7] By convention we append TH to the end of modules constructed in the above manner

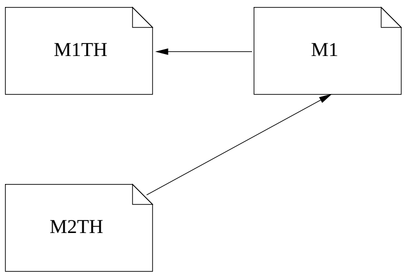

Fig. 5. M2TH imports M1 which splices in declarations in M1TH.

The addition to Template Haskell of a native means to deal with the reification of top-level function declarations would greatly simplify the implementation of PanTHeon and similar programs, and would be less error prone.

5.2 Lack of type information

In Section 3.3 we mentioned that lack of type information prevented us from satisfactorily implementing the unboxing transformation. This is for three main reasons:

1. We require the type of literals in order to choose the correct primitive unboxed arithmetic functions.
2. Knowing the type that an invocation of a polymorphic function would be instantiated to is also necessary to choose the correct primitive unboxed arithmetic functions.
3. Polymorphic data structures cannot contain unboxed values. Therefore, specialised data structures are required. Again, types are needed. The next subsection discusses this further.

Template Haskell recently underwent a substantial revision. One of the features that was added was the ability to reify variable names to glean, among other things, their types. Unfortunately, this is only possible if the variable name in question was brought into scope for the one of the following reasons: it came from another module, it was not generated by a splice and appears somewhere in the current module, or it was generated by a top-level splice occurring earlier in the current module.

In fact, this is the only type information that *can* be available without splicing the declaration in which the variable appears, for in general it is undecidable as to whether an arbitrary meta-program, once run, will produce correctly typed code. (This was the motivation behind the design of Template Haskell's typing system which defers type checking until meta-programs have been run.)

However, in the special case that the reified code was *closed*, in the sense that it contained no further splices and all variable names were in scope, it would be possible, in principle, to type the code. Fortunately, this is precisely the sort of declaration in PanTHeon that we wish to glean type information from.

5.3 Unboxing in the context of polymorphic data structures

Section 3.3 hinted at a problem with unboxed values in the context of polymorphic data structures. One of the restrictions on unboxed values is that they may not be stored in polymorphic data structures. This necessitates the specialisation of polymorphic data structures to monomorphic counterparts. Disregarding the difficulty of doing this in the absence of typing information there is an additional difficulty. While it is possible to reify data type declarations in other modules (using the Template Haskell primitive `reifyDecl`) it is not possible to reify the definitions of functions in those modules. The following example illustrates some of the difficulty arising from this.

```
weird :: Point
weird = head (zipWith (,)  [1] [0.5])
```

Without the ability to reify the definitions of `zipWith` and `head`, and specialise them to work on a monomorphic version of the list data type the only other solution is to marshal data to and from unboxed/monomorphic representations at key points within the function definition, which to be feasible also requires access to type information. At present, it is not clear whether the ability to reify entire modules or functions in other modules will be added to Template Haskell or not. The latter solution will be necessary in case it does not.

5.4 The question of rewrite rules

In our implementation we have chosen not to use rewrite rules to perform algebraic transformation of programs, even though it would, in principle, be possible. Template Haskell provides full control over the timing of the application of transformations unlike GHC's rewrite rules [10]. Experience suggests that it is notoriously difficult to ensure that rewrite rules will be applied when one intends them to be. Because of their complex interaction with the other optimisations of GHC it can often be the case that they are not nearly as applicable as one would like. Also, since we cannot apply the unboxing transformation to rewrite rules we would have to do this by hand.

6 Future Work

We have identified a number of ways in which PanTHeon could be further improved. However, the latter two suggestions are not limited to this particular EDSL; they benefit the extensional meta-programming approach we have taken.

Conversion of domain data types We are still interested in converting the components that make up the colour data type to integers in the range $[0, 255]$ and converting all functions which manipulate colour to operate upon this range. As we noted before, this is not feasible without being able to ascertain the type of individual components of a functional declaration. We believe that this type information would allow more radical conversion. A key function in PanTHeon is `toRGB32` which takes a colour, converts each component to be in the range $[0, 255]$ (which fits in 8 bits) and then packs the four components into a 32-bit integer using bitwise operations. It would be interesting to see if the colours could be represented by this 32-bit representation at all times. This would require colour manipulating functions to use bitwise operations on the colours instead of the native pattern matching facilities of the Haskell language.

Moving optimisation phases of the host language compiler into libraries There is a subtle problem with the interaction between GHC's optimisations and those presented in this paper: our optimisations are performed before any of GHC's. Unfortunately the order in which optimisations are done is often important; some increase the effectiveness of others if performed at the correct time. While we have not (yet) encountered this problem with PanTHeon, we feel this problem would benefit from closer scrutiny.

The solution to this problem is tantalising. One proposal is that *all* optimisations are written in Template Haskell. This would have the effect of shifting a large portion of the compiler into its support libraries and would arguably simplify the implementation considerably.

However this raises new problems. Many of the optimisations in GHC depend upon information gleaned from the structure of the input program; examples include strictness analysis, various forms of data flow analysis, and call graphs. Further, these optimisations are applied to the Core language, a simplified intermediate representation of Haskell programs, that is nonetheless *not* Haskell. (Since it is syntactically simpler and has less complicated semantics, writing optimisations in Core is easier.) The idea of adding meta-programming facilities to Core has been put forward but it is far from clear how this would be implemented. Nonetheless, the potential benefits almost certainly outweigh the difficulty of this proposal.

Access to type information before execution of meta-programs We are interested in implementing a modest extension to the type checker of Template Haskell that allows *closed* reified code to be typed. It would also be interesting to see whether declarations that are not closed could at least be partially typed.

7 Related Work and Conclusion

Many languages have been developed using the embedded approach to domain specific language construction. Examples include an XML translation language

[18] , Fran [3], FranTk [15], and Haskore [9]. These languages are written in an interpretive style in which domain types are modelled as algebraic data types and language primitives as functions which recursively traverse over them. This leads to inefficient code and was the original motivation behind the embedded compiler approach of Pan.

But we feel the embedded compiler approach sacrifices too much. Apart from having to write the entire back-end of a compiler, one also introduces a disparity in the semantics of the host language and the target language which can lead to such problems as the loss of host language features, and the burden of having to duplicate the effect of some constructs, such as *if-then-else* expressions, due to the synthetic nature of the *types* used in the embedded compiler.

Much of the inefficiency of EDSLs results from the host compiler not being able to *see* at the level of the new language. Compilers often have intimate knowledge of semantics of their primitive constructs, and such knowledge allows them to perform sophisticated optimisations to improve the performance of code. But they cannot understand an EDSL in such a manner.

Meta-programming solves this problem by allowing the programmer to use their knowledge of the EDSL to write domain specific optimisations. In this paper we have demonstrated that:

- Optimisations are easy to write. This is facilitated by the quasi-quote notation and pattern matching facilities of Template Haskell. Incidentally, this affirms the half-serious remark that Haskell is a domain specific language for writing compilers.
- Extensional optimisations work well with those inherited from the host language.
- The optimisations are effective. There was at least a factor of two speed up in all the examples we tested on.
- The implementation effort, in terms of raw lines of code, is significantly less.
- Additional meta-programming facilities would increase the power of extensional meta-programming. In particular, more type information is the key to writing more sophisticated transformations.

The source code of PanTHeon, along with instructions for building it, can be found at `http://www.cse.unsw.edu.au/~sseefried/pantheon.html`.

The comments and suggestions of a number of people helped improve the presentation of this paper. In chronological order of contribution, the authors would very much like to thank Ian Lynagh, Simon Peyton Jones, John O'Donnell, Anthony Sloane, Donald Stewart, Nicolas Magaud and André Pang.

A Listing of `module Image`

Below is a listing of just the parts of `module Image` needed to understand the examples presented in this paper.

```
module Image
where

type Point = (Float, Float)
type Colour = (Float, Float, Float, Float)
type Image c = Point -> c
type ImageC = Image Colour
type Warp = Point -> Point

whiteT :: Colour
whiteT = (0,0,0,0)

whiteH, blackH :: Colour
whiteH = (1,1,1,0.5)
blackH = (0,0,0,0.5)

lift0 h       = \p -> h
lift1 h f1    = \p -> h (f1 p)
lift2 h f1 f2 = \p -> h (f1 p) (f2 p)

empty :: ImageC
empty  = lift0 whiteT

distO :: Point -> Float
distO (x,y) = sqrt (x*x + y*y)

swirl :: Float -> Warp
swirl r p  = rotateP ((distO p) * (2*pi/r)) p

cOver :: Colour -> Colour -> Colour
cOver (r1,g1,b1,a1) (r2,g2,b2,a2) = (h r1 r2, h g1 g2, h b1 b2, h a1 a2)
     where h x1 x2 = a1* x1 + (1 - a1) * x2

over :: ImageC -> ImageC -> ImageC
over = lift2 cOver
```

B Some algebraic properties of Pan

```
empty ‘over‘ image = image
image ‘over‘ image = image
translate (x1,y1) (translate (x2,y2) im) = translate (x1+x2, y1+y2) im
rotate a im = rotate (a - n*2*pi) im  (where n = a ‘div‘ 2*pi)
rotate a1 (rotate a2 im) = rotate (a1 + a2) im
scale (x1,y1) (scale (x2,y2) im) = scale (x1*x2, y1*y2) im
fromPolar  (toPolar f) = f
```

References

1. The Glasgow Haskell Compiler. `http://haskell.org/ghc`.
2. Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. URL: http://www.cs.rice.edu/~taha/publications.html, 2003.
3. Conal Elliott. Functional implementations of continuous modeled animation. *Lecture Notes in Computer Science*, 1490:284–, 1998.
4. Conal Elliott. Functional Image Synthesis. In *Proceedings Bridges 2001, Mathematical Connections in Art, Music, and Science*, 2001.
5. Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, May 2003.
6. Daniel P. Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 348–355, 1984.
7. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.
8. Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
9. Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.
10. Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. *International Conference on Functional Programming (ICFP 2001). Haskell Workshop.*, September 2001.
11. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
12. Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. URL: http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/Unrolling_and_Simplifying_Expressions_with_Template_Haskell.ps, May 2003.
13. André Pang, Donald Stewart, Sean Seefried, and Manuel Chakravarty. Plugging Haskell In. To be published in *Haskell Workshop 2004*, June 2004.
14. Arch D. Robinson. The Impact of Economics on Compiler Optimization. In *Proceedings of the ACM 2001 Java Grande Conference, Standford*, pages 1–10, June 2001.
15. Meurig Sage. FranTk — a declarative GUI language for Haskell. *ACM SIGPLAN Notices*, 35(9):106–117, 2000.
16. Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. *ACM SIGPLAN Notices: PLI Workshops*, 37(12):60–75, 2002.
17. Brian Cantwell Smith. Reflection and Semantics in Lisp. *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages*, pages 23–35, 1984.
18. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.