Partial Evaluation for Domain-Specific Embedded Languages in a Higher Order Typed Language

Duncan Coutts

October 2004

Abstract

Partial evaluation is a mature area of research that has demonstrated the power of the technique and the wide range of potential applications. One might have expected that by now partial evaluation tools would have found a place in the programmer's toolbox along side other commonly used tools like type checkers, parsers generators and program development environments. This is unfortunately not the case.

This report looks at some of the reasons why there are few usable partial evaluation tools available for modern programming languages. We propose an alternative philosophy, which instead of aiming for full automation, emphasises programmer control, semi-automatic methods and process visibility.

This report gives an introduction to partial evaluation and its application to domain-specific embedded languages. It describes an initial prototype partial evaluation tool for a first order subset of Haskell and some experiments with example programs. Finally, we describe the technical challenges involved in developing a practical tool that could be successfully applied to optimising domain-specific embedded languages and other active libraries.

Contents

1 Partial Evaluation **3**

1.1 Theoretical basis . . . 3

1.2 Simplified re-formalisation . . . 3

1.3 Notation . . . 5

1.4 When specialisation is useful . . . 5

1.5 Staged computations . . . 5

1.6 On-line and off-line partial evaluation . . . 6

1.7 Binding times and program divisions . . . 6

1.8 The Futamura projections . . . 7

1.9 Generating extensions . . . 9

1.10 The cogen approach . . . 9

1.11 Types . . . 9

1.12 Implementation considerations . . . 11

1.13 Terminology . . . 12

2 Domain-Specific Embedded Languages **12**
2.1 Parsing combinators . . . 13
2.2 PAN . . . 13

3 Why Partial Evaluation does not apply now **14**
3.1 The problem of binding time inference . . . 14
3.2 Types . . . 15
3.3 Higher-order static values . . . 15
3.4 Lack of implementations . . . 15

4 Current Prototype **16**
4.1 Template Haskell . . . 16
4.2 Double encodings . . . 19
4.3 The generating extension transformation . . . 22
4.4 Type inference . . . 26
4.5 Partially static data structures . . . 30
4.6 Structure of the program . . . 34

5 Thesis Plan **34**
5.1 Further work and features . . . 35
5.2 Comparative research . . . 37
5.3 Applications . . . 37
5.4 Work plan . . . 38

A Examples **39**
A.1 Simple functions . . . 39
A.2 Interpreter . . . 40
A.3 Numerical examples . . . 44

List of Figures

1 Mix . . . 4
2 Cogen . . . 8
3 The Ackermann function, its incorrect and correct generating extensions . . . 18
4 Specialisation of the Ackermann function to $m = 4$ . . . 22
5 Memoising transformation . . . 26

List of Algorithms

1 Encoding algorithm . . . 21
2 Memoise function . . . 25

1 Partial Evaluation

A common technique in programming is to construct a new program by generalising an existing program. This typically involves adding an extra input to the program and substituting a constant (or constant expression) for the new program parameter. This can be useful for several reasons such as code reuse, greater modularity, better maintainability or to gain a better understanding of the program. However it is almost invariably accompanied by an increase in program running run time due to additional parameter passing overheads, reduced code cache locality and reduced opportunities for inlining and other low level optimisations.

Partial evaluation is a method for performing the reverse transformation, that of program specialisation. We typically take a general program with several parameters and specialise it to constant values of one or more of its parameters. The resulting specialised program will often be faster than the original general program when run with the same parameters.

This specialisation transformation can be automated or semi-automated and this holds out the promise that we can write modular maintainable programs and still obtain efficient programs.

1.1 Theoretical basis

The fact that specialisation can be done is shown by Kleene's *s-m-n* theorem in computability theory. In computability theory, computable functions can be encoded as large integers and so can be the input to, or result from, other computable functions. The formalism uses the following notation $\varphi_x^{(k)}$ to mean the computable function of k arguments with representation x.

The theorem states that if we start with a computable function with $m+n$ arguments

$$\varphi_x^{(m+n)}$$

and a vector $(y_1,\ldots,y_m)$ of m values then there exists another computable function x' with n arguments

$$\varphi_{x'}^{(n)}$$

such that it is equal to the original function on the remaining n arguments

$$\lambda z_1,\ldots,z_n.\varphi_x^{(m+n)}(y_1,\ldots,y_m,z_1,\ldots,z_n) = \varphi_{x'}^{(n)}$$

and furthermore not only does this specialised function x' exist, but it is computable from x and the first m arguments $(y_1,\ldots,y_m)$.

$$x' = s(x,y_1,\ldots,y_m)$$

The typical proof of this theorem gives an explicit construction for the function (program) s. The program in Kleene's proof yields specialised functions that are less efficient than the original general function. This is of no concern in computability theory but as computer scientists we would like to use this specialisation procedure to make programs run quicker.

1.2 Simplified re-formalisation

So computability theory tells us that specialisation is always possible, even if it does not always yield faster programs. Intuitively we expect that we could yield faster specialised

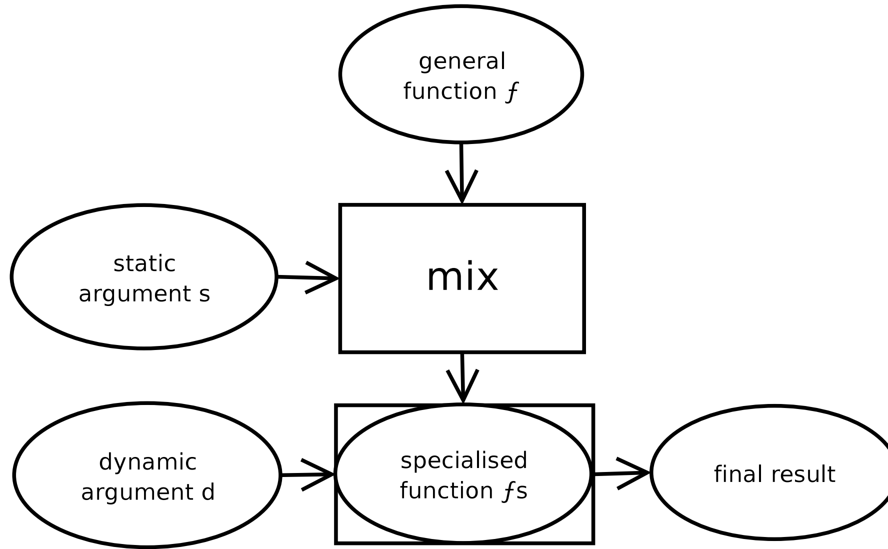

Figure 1: Mix

programs by evaluating the constant or static parts of the program and leave the rest of the computation until later. The more we can do earlier the better.

It is simpler, rather than consider a function with many parameters (as the *s-m-n* theorem does), to consider a function with just two parameters and to think about specialising on the first argument. We call the first argument *static* and the second *dynamic* since the traditional view is that we receive the first argument at compile time and do not get the second until runtime. (Note that this setup is not essential, we may get both arguments at runtime, the key point however is that the computation is *staged*, that is we get one argument before the other.)

$$[|f|] : S \to D \to X$$

So to clarify our intuitive notion of specialisation in this context, we would like to specialise by evaluating those parts of the program f that depend only on the static S argument and do not depend on the dynamic D argument. The result of our specialisation (on $s \in S$) will be the *residual* program

$$[|f_s|] : D \to X$$

For this to be a correct specialisation it must be the case that

$$[|f|]\; s\, d = [|f_s|]\; d \quad \forall d \in D$$

The program that calculates the program f_s from f and s is traditionally called *mix* and is defined by

$$[|mix|]\; f\, s = f_s$$

and by using our previous definition of f_s we arrive at the conventional form of the *mix equation*

$$[|\,[|mix|]\; f\, s|]\; d = [|f|]\; s\, d \quad \forall s \in S, d \in D$$

Figure 1 gives a pictorial representation of this equation. The arrows represent data flow, the circles represent data values and the boxes represent programs. Of course the specialised function f_s is both a data value and a program.

1.3 Notation

It is important when describing functions that manipulate representations of other functions, encoded using various languages, to keep a clear distinction between the syntactic and semantic value of programs. For this section of this dissertation we will use the standard mathematical notation which is to apply $[|\ldots|]$ semantic brackets to *code* to get its semantic value $[|\mathit{code}|]$ from its syntactic value. So in this notation, unadorned names are syntactic values or other ordinary values that a program manipulates.

In later sections we will use a different notation that follows the notation of the implementation framework we use, namely Template Haskell. Unfortunately, this notation is rather the reverse of the standard notation which can be confusing. To try to reduce this confusion, expressions written in this notation will be typeset `in a typewriter font`. In particular, in the Template Haskell notation, unadorned values are semantic values not representations and the same bracket notation is used with the *reverse* meaning, to give the syntactic value of the quoted expression `[| code |]`.

1.4 When specialisation is useful

Specialisation is most often useful when we have a function that is run many times where some parameters change frequently and others assume only one or a small number of values. An ideal candidate is a function that is always called with one argument that is a compile time constant. A slight generalisation of this situation is where a function is called with a small finite number of different constant arguments, say because the function is used in several different parts of a larger program.

As noted before it need not be the case that the argument we specialise on be a compile time constant. A ray tracer is an example of this. A standard ray tracer reads a scene description from a file and then produces an image by evaluating a ray casting function for every (x,y) pair in the final image. So we have a function that is evaluated many thousands of times; one argument, the scene description, is the same for every invocation. Furthermore, calculating the colour of each pixel is fairly expensive, so there is the potential that a ray casting function specialised to the scene at hand could yield significant run-time savings. Crucially, for specialisation to be a benefit, the saving from the specialised ray caster must outweigh the time required to produce the specialised ray caster from the general ray caster.

In this dissertation we will restrict ourselves to considering the former situation where the value we specialise upon is a compile time constant because it is the most common situation. Also, it allows us to unambiguously refer to computations that take place at "compile time" or "run time", which can be confused in the latter situation as a typical implementation requires embedding a compiler and linker into the run time system.

1.5 Staged computations

A staged computation, as the name suggests, proceeds in two or more stages. One stage produces some intermediate data structure which is interpreted by the following stage or equivalently one stage produces another program which *is* the next stage. Staging a program, that is converting an ordinary one-stage program into a two or more staged program, is a fairly common technique. Examples include the traditional implementation of regular expression libraries where a regular expression is "compiled" into a finite state machine which is then interpreted. Manually staging a program usually involves a good deal of effort and so is only done when there are sufficiently great performance benefits.

By using specialisation we can stage a program automatically. Of course this will not necessarily stage the program in an ingenious way that a programmer might; it will follow

the structure of the computation. We could not write a simple direct regular expression interpreter and hope by partial evaluation to end up with a sophisticated NFA → DFA algorithm[1]. Nevertheless, useful speed improvements can often be obtained.

By looking again at the mix equation it is clear that we do obtain a staged computation.

$$[|\,[|mix|]\ f\ s|]\ d = [|f|]\ s\ d$$

The first stage to run is $[|mix|]\ f\ s$, which produces the residual program f_s which is run as the second stage[2].

1.6 On-line and off-line partial evaluation

There are two major techniques for performing partial evaluation called *on-line* and *off-line* partial evaluation. Both types are rather like an interpreter (indeed they must embed an interpreter to perform the static computations), they take the general program and its static input and produce the residual program as output. The distinction between the techniques is on what basis they decide to do reduction or unfolding. On-line partial evaluators use the actual values calculated during specialisation to help decide what should be evaluated while an off-line technique only looks at the program. On-line partial evaluators have more information available and so potentially can make better decisions yielding more specialisation, however off-line partial evaluation is often simpler and makes it easier to provide guidance to the specialiser since only the original program need be considered. It also makes self-application significantly easier (see section 1.8). The implementation described in this paper uses an off-line technique. Indeed it would not be possible to use an on-line technique because the static argument is not even available. The reason for this will be explained shortly.

Off-line partial evaluation is typically split into two phases, where the first phase produces *binding time annotations* for the program to be specialised and the second phase blindly follows these annotations as it specialises the program. This split allows the second phase to be relatively simple and makes it easier to use various methods to create the binding time annotations.

1.7 Binding times and program divisions

Binding time annotations specify which parts of the program are static and which are dynamic. The term binding time refers to when expressions can be assigned values, e.g. static expressions can be evaluated at compile time and are said to have static binding time. For imperative languages this often called a *division* because it divides the program's variables into static and dynamic sets. A division must be congruent; each static variable may only depend on other static variables, otherwise specialisation is impossible. A congruent division always exists since the trivial division that classifies every expression as dynamic (and yields no specialisation) is congruent. It is usually better to be able to classify more expressions as static since that leads to doing more of the computation at compile time and leaving less for run time.

Creating the binding time annotations can either be manual, that is the programmer gives fully explicit binding time annotations, or semi or fully automatic. When the process is automatic is is often called *binding time analysis*, abbreviated to BTA.

[1] Although in some cases such a hope is not so far fetched. The KMP string matching algorithm can be obtained from a simple string matcher with suitable binding time annotations[1].

[2] If this seems impractical because the first stage program embeds the partial evaluator see sections 1.8 and 1.9 for a solution using generating extensions that does not have this deficiency.

Why might one not want fully automatic binding time analysis? There is one important theoretical difficulty in partial evaluation which is that the problem of finding an optimal division is uncomputable. An optimal division would classify as much of the program as static as is possible while still being congruent. So a fully automatic binding time analysis cannot always achieve the most specialisation possible. There are several algorithms which can achieve good divisions. However in practice they do not always give quite the results that users want, either because they do not recognise opportunities for specialisation that the user does, or they perform too much specialisation which can lead to producing very large residual programs. So semi-automatic algorithms may be preferable in some cases.

1.8 The Futamura projections

Much early interest in partial evaluation was inspired by Futamura[7]. He discovered that it is possible to generate compilers and compiler generators from interpreters, using a partial evaluator and self-application of the partial evaluator.

Suppose we start with a program *source* written in language L_{source}, we can run it on input *in* to get the result *out*

$$out = [|source|]_{L_{source}} \ in$$

Now suppose we have an interpreter *int* for a language L_{source} written in language L_{target}. We can obtain the same result *out* by running the interpreter *int* on the text of the *source* program and its input *in*.

$$out = [|int|]_{L_{target}} \ source \, in$$

Suppose further that we have a partial evaluator *mix*, that accepts and emits programs in language L_{target}, then by applying the mix equation we obtain

$$out = [| \, [|mix|] \ int \, source|]_{L_{target}} \ in$$

and by defining $target = [|mix|] \ int \, source$ we can see clearly that we have obtained the text of a program written in language L_{target} that has the same semantics as our starting program *source*.

$$out = [|target|]_{L_{target}} \ in$$

So we have "compiled" a program from our source language to our target using just an interpreter and a partial evaluator. This is very promising since writing interpreters is significantly easier than writing compilers and the investment in the partial evaluator can be reused for many different interpreters. This equation

$$target = [|mix|] \ int \, source$$

is known as the first Futamura projection. Note that it did not matter in what language *mix* was written. For the second Futamura projection we must assume that *mix* is written in the same language that it accepts and emits namely L_{target}. The key idea in the second Futamura projection is to consider *mix* as a program text as well as just a function for transforming other program texts, and to contemplate what we might get if we apply *mix* to its own program text. Starting with the equation for the first projection, we apply the mix equation again

$$\begin{aligned} target &= [|mix|] \ int \, source \\ &= \left[\left|[|mix|]_{L_{target}} \ mix \, int\right|\right]_{L_{target}} \ source \\ &= [|compiler|]_{L_{target}} \ source \end{aligned}$$

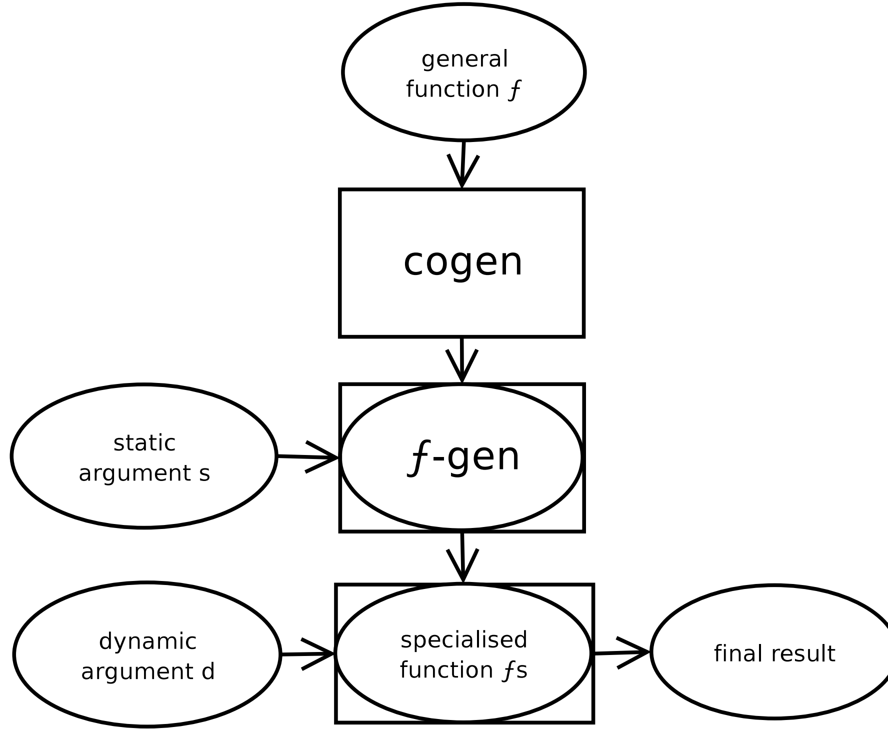

Figure 2: Cogen

and find that we end up with a program that translates *source* to *target*, that is it is a compiler from L_{source} to L_{target}. The second Futamura projection is then

$$compiler = [|mix|]_{L_{target}} \, mix \, int$$

This might be useful since the compiler program is likely to run somewhat faster than running mix on the interpreter and a source program because the overhead of mix interpreting the interpreter may be reduced or eliminated. The third Futamura projection is what we get if we follow this investigation to its logical conclusion and apply the mix equation again

$$\begin{aligned} compiler &= [|mix|]_{L_{target}} \, mix \, int \\ &= \left[\left|[|mix|]_{L_{target}} \, mix \, mix\right|\right]_{L_{target}} \, int \\ &= [|cogen|]_{L_{target}} \, int \end{aligned}$$

In the third Futamura projection

$$cogen = [|mix|]_{L_{target}} \, mix \, mix$$

we use the name *cogen* because it is a program that generates compilers from interpreters. Again, this might be somewhat faster and could be useful if one was generating many compilers from many interpreters. This does seem a rather uninteresting niche application, however it is worth noting that the third projection no longer mentions interpreters or compilers at all. It is in fact much more general than that; it is quite similar to mix itself. While *mix* is like an interpreter, *cogen* is like a compiler (see figure 2). It transforms an arbitrary program into its *generating extension*.

1.9 Generating extensions

The generating extension of a program *P* is a program *P–gen* that accepts the arguments we want to specialise *P* upon and produces the residual program (which accepts the remaining arguments and returns the final output). Of course this is the same job that *mix* does, but *mix* is a general program for doing this, it works for any input program we wish to specialise. A generating extension is already a specialised program. It is specialised to producing the residual program for just one original general program *P*. As a consequence we might expect it to produce the residual program more quickly than running *mix* on the original program. However that is not the only advantage as we shall see later when we come to consider using typed languages (see section 1.11).

Put like this, it is not surprising that we can construct a program's generating extension *P–gen* by specialising *mix* to *P*. As noted before, using *cogen* is likely to be faster than running *mix* on itself and an input program.

1.10 The cogen approach

The cogen approach is to do partial evaluation by writing *cogen* rather than *mix*. There are a few reasons why one might do this. Since writing compilers is harder than writing interpreters it may appear that writing *cogen* would be harder than writing *mix*. Remember however that *mix* must embed an interpreter to evaluate the static parts of the program. On the other hand, *cogen* transforms source code to source code and an existing tool for the language can be reused to compile and run the resulting generating extension. So depending on the complexity of the semantics of the language we are targeting, it may actually be less work to use the cogen approach rather than the direct approach. For Haskell this is certainly the case since writing a Haskell interpreter is a significant undertaking while it turns out that a simple implementation of *cogen* (without binding time analysis) can be written very compactly (see section 4).

As one can see from figure 2, *cogen*'s only input is the program to be specialised. This is why the cogen approach is incompatible with on-line partial evaluation, *cogen* must operate on the original general program without access to its static arguments.

Note that if we have *cogen* we not not also need *mix* since:

$$\begin{aligned} [|f|]\, s\, d &= [|\,[|mix|]\, f\, s|]\, d \\ &= [|\,[|\,[|cogen|]\, f|]\, s|]\, d \end{aligned}$$

That is, anything we can do with *mix* we can do with *cogen* (albeit slightly less directly).

1.11 Types

Most research in partial evaluation has been done in untyped languages like the untyped lambda calculus and languages from the LISP family such as Scheme.

Before considering types we must establish some notation to denote the types of encodings of typed programs or values. That is if we have a program syntax *prog* and its semantic value $[|prog|]$ is of type T, we would like to have a notation for the type of the representation *prog* itself. It is common in the literature to use an over-bar $\overline{T}$ to denote this type, however we will use a more Haskell-like notation and say that the encoding of expressions of type T will be of type `Exp T`.

We will now look at assigning a sensible type to our partial evaluation function *mix*. It is obvious that it must take two arguments: a function to be specialised and that function's

static argument. The result will be the specialised function. However we must consider if and how these values will be represented or encoded. It is clear that we need access to the program's syntax[3] to be able to interpret and manipulate it. If we assume the program we are specialising has type `a -> b` then the syntax of the program is of type `Exp (a -> b)`. Note that assuming the function to be specialised has dynamic arguments, the program will actually have type `Exp (a -> b -> c)` or similar. This is just an instance of `Exp (a -> b)` so we need consider only this case. The residual program will also be an encoding, so will have type `Exp b`. The question then is whether the function's static argument is to be encoded or not. That is, should the type of mix be

$\mathtt{mix} :: \mathtt{Exp}\,(a \to b) \to a \to \mathtt{Exp}\,b$
or
$\mathtt{mix} :: \mathtt{Exp}\,(a \to b) \to \mathtt{Exp}\,a \to \mathtt{Exp}\,b$

In Haskell the former could not be implemented because we would not be able to perform case analysis on the argument. The problem is that the type of the second argument depends on the value of the first argument and the first argument could encode an unbounded number of types but in traditional type systems, a finite program can only have finitely many types. A dependent type system allows an unbounded number of types in a program by allowing types to be parameterised by values. So it seems likely that a dependently typed language could allow an implementation of mix with an unencoded second argument.

In light of the similarity of a partial evaluator and an interpreter it is not surprising that a partial evaluator written in an non-dependently typed language must use a universal type to encode values used in the target program, since typed interpreters must do this too while dependently typed self-interpreters can work with unencoded values.

So a Haskell implementation would have to use the latter type; it must take an encoding of the value the program will manipulate. Let us now consider the types we would get for the various Futamura projections under each type of *mix*.

For a version of *mix* that takes an unencoded second argument we get these types

$$
\begin{array}{ll}
\mathtt{mix}_1 & :: \mathtt{Exp}\,(a \to b) \to a \to \mathtt{Exp}\,b \\
\mathtt{mix}_1\ [|\mathtt{mix}_1|] & :: \mathtt{Exp}\,(a \to b) \to \mathtt{Exp}\,(a \to \mathtt{Exp}\,b) \\
\mathtt{mix}_1\ [|\mathtt{mix}_1|]\ [|\mathtt{mix}_1|] & :: \mathtt{Exp}\,(\mathtt{Exp}\,(a \to b) \to \mathtt{Exp}\,(a \to \mathtt{Exp}\,b))
\end{array}
$$

whereas a version of *mix* that takes an encoded second argument has the following types for its Futamura projections

$$
\begin{array}{ll}
\mathtt{mix}_2 & :: \mathtt{Exp}\,(a \to b) \to \mathtt{Exp}\,a \to \mathtt{Exp}\,b \\
\mathtt{mix}_2\ [|\mathtt{mix}_2|] & :: \mathtt{Exp}\,(\mathtt{Exp}\,(a \to b)) \to \mathtt{Exp}\,(\mathtt{Exp}\,a \to \mathtt{Exp}\,b) \\
\mathtt{mix}_2\ [|\mathtt{mix}_2|]\ [|\,[|\mathtt{mix}_2|]\,|] & :: \mathtt{Exp}\,(\mathtt{Exp}\,(\mathtt{Exp}\,(a \to b)) \to \mathtt{Exp}\,(\mathtt{Exp}\,a \to \mathtt{Exp}\,b))
\end{array}
$$

In both cases the type of the third projection is really just the same as the second. The difference is the extra $\mathtt{Exp}\,(\ldots)$ around the outside, in other words it is a program text. We can easily obtain the program itself, eliminating the extra $\mathtt{Exp}\,(\ldots)$ and giving us the same type as the second projection.

The key difference to note is, in the second version, the double encoding of the program to be specialised and that the result program takes the static argument as an encoding which means it must manipulate it as an encoding rather than a native value.

As an example to make these complicated types somewhat clearer, suppose we have an interpreter for a simple imperative language with the following type

$\mathtt{runProg} :: \mathtt{Prog} \to \mathtt{State} \to \mathtt{Value}$

[3] This is perhaps not so obvious since one of the claims for Type Directed Partial Evaluation (TDPE) is to be able to perform partial evaluation with access only to a compiled representation of the program rather than its full abstract syntax[5].

It takes the abstract syntax for a program, the initial state of the global variables and returns the final result of the program. Let us now generate a compiler from this interpreter. For $\text{mix}_1/\text{cogen}_1$ we can apply cogen_1 to the text of the interpreter and get a compiler with the obvious type

$$\texttt{compiler} = [|\texttt{cogen}_1|]\ \texttt{runProg}$$

$$[|\texttt{compiler}|] :: \texttt{Prog} \rightarrow \texttt{Exp}\,(\texttt{State} \rightarrow \texttt{Value})$$

For $\text{mix}_2/\text{cogen}_2$ the situation is more complicated. We must supply a double encoding of the interpreter (the text of a program that produces the text of the interpreter as output) and the result is a compiler that accepts an encoding of the abstract syntax rather than simply the abstract syntax.

$$\texttt{compiler} = [|\texttt{cogen}_2|]\ \texttt{runProg}' \qquad \text{where } [|\texttt{runProg}'|] = \texttt{runProg}$$

$$[|\texttt{compiler}|] :: \texttt{Exp}\ \texttt{Prog} \rightarrow \texttt{Exp}\,(\texttt{State} \rightarrow \texttt{Value})$$

It is clear that were the former type of mix implementable it would be much preferable since the double encoding involved in the latter is inelegant and would introduce a great deal of overhead. Indeed this is exactly the problem that Dal-Zilio and Hughes[4] discovered when they built a partial evaluator for a subset of Haskell in Haskell. The double encoding had a huge impact on memory use and running time. They went to significant lengths to try to combat the memory bloat but were ultimately only partially successful. We would like to avoid this problem of double encoding of the values manipulated by the specialised program[4].

While in an ordinary typed language it may be impossible to generically manipulate an unencoded value from the program we are specialising, it is certainly possible to generically manipulate *code* that directly acts on unencoded values. We have already seen this technique, it is the cogen approach. Indeed Launchbury[14] suggested the cogen approach as a solution to the problem of typed partial evaluators.

This is the technique that we will follow in the rest of this dissertation.

1.12 Implementation considerations

The standard technique for performing specialisation involves duplicating chunks of code for each value of their free static variables and then specialising each chunk using those static variables. The residual program contains all these specialised chunks. More precisely, we duplicate and specialise particular classes of program points. For example in a standard imperative language it is typical to pick basic blocks as the program points to specialise upon. In a first-order functional language, named functions is a typical choice. In that case the static free variables are simply the static arguments to the function. It is possible to pick different program points to specialise upon. Another common choice is to specialise at dynamic conditionals. In many examples this gives residual programs that have less code duplication than the standard technique (see [12] section 5.5.4).

The result of this standard specialisation technique typically produces a large number of trivial transitions or function calls. To get better residual code we can unfold some of the trivial calls. There are a number of different *unfolding strategies* one can devise, from the trivial to extremely sophisticated. The unfolding can either be integrated into the main specialisation phase or done as a post-processing phase. In the former case the binding time annotations/analysis determine which calls to unfold and which to residualise. The latter admits a number of simple ad-hoc unfolding strategies. In all cases termination of the

[4]The approach presented later in this paper does use double encodings, but only of programs and not of the values they manipulate so there is no run-time overhead. See section 4.2 for details.

unfolding is a concern. It is easy to write a strategy that fails to terminate in many ordinary cases.

In languages with side-effects the specialisation phase and unfolding strategy must be constrained to not discard, duplicate or re-order potentially side-effecting computations. For Haskell this is of course not an issue since the language semantics do not include side-effects.

1.13 Terminology

Describing the behaviour of code generating programs can become confusing if we do not keep clear what stage computations are happening in. There are three stages to distinguish. We will use the term "generation time" to denote the calculations that are performed as the generating extension is constructed. The term "compile time" will refer to code that is run when the generating extension is run to produce the residual program. The term "run time" will be used for calculations performed when the final residual program is executed.

2 Domain-Specific Embedded Languages

Building domain-specific languages is a useful abstraction mechanism and a common technique to make the implementation of the DSL easier is to embed them in a general purpose host language like Haskell. Embedding allows features of the host language to be reused such as language design decisions, parsing and semantic checks, type checks and a code generation back-end. DSLs can be embedded in many different high level languages but Haskell is often cited as having features which make it a particularly good host language. These features include, flexible syntax (including user defined operators and precedence), higher order functions and a sophisticated type system. There are drawbacks to the embedding approach including the difficulty of domain specific error messages (it would be better to express some error messages in terms of the embedded language's domain) and domain specific optimisation.

Domain specific optimisations can be important but the compiler for the general language knows nothing about the semantics of the embedded language or extra laws that hold in it. There are various approaches to solving this problem including using modular or extensible compilers where the author of the DSEL writes an extra pass which gets performed by the compiler. This has the disadvantage of non-portability and requiring the language author to know a great deal about the design of the compiler. Other solutions involve writing custom preprocessors but this negates much of the advantages of using an embedding a language since the preprocessor will need much of the infrastructure of a compiler front end. Also, using preprocessors does not scale well as only one preprocessor can be used at once while a system may need to use more than one DSEL.

Active Libraries[23] is the term given to a collection of related techniques for tackling this problem. They typically extend the language with features that allow libraries to take an active role in generating or optimising client code. The idea is that the library author can specify how the abstractions defined in the library are reduced to lower level efficient code, taking advantage of the authors knowledge of the semantics of the abstraction. Some of these techniques work by using transformation rules while other work by code generation. A disadvantage of transformational techniques is that they often do not provide sufficient control to reliably apply high-level domain-specific optimisations.

2.1 Parsing combinators

Libraries for parsing was one of the first examples of a DSL embedded in Haskell. They allow top-down parsers to be written in a compositional way that is similar to a high level BNF description of the grammar. In fact an advantage over BNF (which is a DSL in itself) is that it allows abstractions that BNF does not. For example they allow higher order combinator parsers such as a parser that accepts the result of another parser surrounded by brackets. They also allow semantic actions to be embedded in the parser and type checking to ensure that the parsers are used consistently.

While combinator parsers are very elegant, traditionally they have suffered from poor performance. This is partly due to subtle issues involving backtracking and retention of input which increases memory consumption but more fundamentally it is from the lack of static analysis. Bottom up parser generators give better performance because they are able to analyse the grammar before hand and produce parsing tables which can be interpreted efficiently by a push down automaton. Monadic parser combinators lack any introspection capacity and so cannot do the analysis required to take the low-level table-driven approach.

A different form of combinator developed by S.D. Swierstra and L. Duponcheel[20] does allow some static analysis to be performed before parsing starts which enables the precomputation of tables which speeds the later work. The crucial point about the form of the combinator is that it allows some work to be done based purely on how the building blocks of the parser are combined and independent of the text to be parsed. Their technique is claimed to achieve "implicit partial evaluation" by taking advantage of lazy evaluation and carefully ordering the computations.

2.2 PAN

PAN is a language for interactive functional images. It is implemented as an embedded domain-specific language in Haskell. It is an interesting example of a DSEL because aggressive domain specific optimisations are required to achieve acceptable performance. A simple implementation as a Haskell library relying on the standard compiler's optimisation features produce unacceptably slow animations. The implementation consists of a combinator library that constructs an intermediate representation of the program which is then subject to aggressive unfolding followed by common sub-expression elimination to reduce the degree of duplication introduced by the unfolding. The result is emitted as a C source program which is then compiled by the C compiler and loaded up and executed to produce the image or animation.

In feedback on this project it was suggested that it might be possible to explain the transformations in the PAN optimiser in a more elegant manner by partial evaluation of an interpreter to the intermediate data structure.

Another more recent re-implementation of PAN by Sean Seefried called PanTHeon[18] uses Template Haskell in its implementation. It takes a similar approach of applying transformations to the abstract syntax tree of the image function. Because it is implemented using Template Haskell[5], instead of the combinators constructing an intermediate representation, the Haskell code itself is manipulated and optimised. The advantage of this system is that since the resulting code is fed back to the Haskell compiler, the effort of building a C code generation back end is saved. While this implementation does not reach the level of performance of the original PAN implementation in all cases, it does come close in several examples. This suggests that the main problem is not with the quality of the Haskell compiler's code generator but in the difficulty of doing the right sort of transformations needed to get good low level code for a wide variety of examples.

[5] See section 4.1 for an introduction to Template Haskell.

3 Why Partial Evaluation does not apply now

3.1 The problem of binding time inference

3.1.1 Clever compiler syndrome

Partial evaluators suffer from “clever compiler syndrome”. Clever compiler syndrome is a phenomenon noticed particularly in highly optimising or parallelising compilers, particularly those for imperative languages. Optimisations in a compiler serve a dual purpose. One is to straightforward: to make programs run faster. The other is to allow programs to be written in a more high level style and still be as fast as their low level counterparts. This is one reason commercial software development moved away from assembly languages - it was possible to achieve similar performance with mid-level languages and an optimising compiler. This is the central point, that for projects with performance requirements the use of certain optimisations in the complier enables *greater abstraction* because the programmer does not need to know about the low level code produced. However for many modern optimising compilers, particularly for complex languages such as C++ with templates or parallelising C and Fortran compilers this abstraction benefit is lost or heavily mitigated. The reason is that many of the optimisations performed only apply for certain patterns and with complex side conditions (aliasing conditions for example). This means that for a programmer to reliably take advantage of these optimisations and so produce readable high level code, they must be intimately familiar with the implementation of the compiler they are using. This destroys abstraction. It also makes code non-portable between compilers. These problems are particularly acute in the field of C++ template libraries[22] where code must be “ported” from one standard-compliant compiler to another so that each compiler can be made to recognise the intended optimisation opportunities.

3.1.2 Too little specialisation

Fully-automatic partial evaluators have this same problem although to a lesser degree. The issue is with the binding time analysis. As stated before, it is impossible to compute the optimal binding time division though there are several algorithms that achieve good divisions. However in particularly complex situations the algorithm may not recognise that some value is static though the programmer with their greater knowledge of the domain may know it is static. In this case the programmer may be forced to rewrite the code to demonstrate more clearly to the tool that the code has the desired binding times. To make things worse (from the predictability point of view) many partial evaluators implement various *binding time improvement* transformations. These are semantics preserving, syntactic changes such as re-associating numerical expressions that increase the *binding time separation* of the code in question allowing more code to classified as static. Automating these transformations relieve the programmer from performing them manually but again means that the programmer must know which ones are implemented if they wish to take advantage of them.

Hudak makes just this point about fully-automated partial evaluators in[10]:

> “I feel that more user-friendly techniques are needed. In particular, in contrast to a fully-automated approach, a *semi-automated* approach would have two advantages: First, automatic approaches have not matured enough in recent years to give us the confidence we need to meet our goals. Second, I think that it’s important for the user to have better, more explicit control over the transformation process. Reasoning about the behavior of fully-automated systems can be difficult, and it gets worse as the sophistication of the automation increases.” (emphasis in original)

3.1.3 Too much specialisation

Then there is also the opposite problem; partial evaluators duplicate code in pursuit of performance however they will do this without regard to how important a piece of code is to the overall performance of a program (since this is in general a very difficult thing to estimate). As a consequence an automatic partial evaluator may bloat parts of the program that were never performance critical, thereby reducing the overall performance of the program by making it take longer to load and by placing greater demands on virtual memory. In both situations it would be beneficial for the programmer to be able to influence what gets specialised by guiding the binding time inference.

3.2 Types

Partial evaluation research has traditionally been done using untyped languages. Using a typed language presents a number of difficulties as explained in section 1.11. The primary problem is that traditional approaches involve self-application of the partial evaluator which becomes difficult in a typed language with the need to use encodings of the values of the subject program.

There has been comparatively less interest and fewer tools for typed as for untyped languages. This may also have a cultural element; the early research in partial evaluation was done with untyped languages because they were what was available, and so there is a certain inertia for later research that extends the techniques to continue to use the same family of languages.

3.3 Higher-order static values

Modern functional languages and techniques encourage the use of higher-order functions while the traditional specialisation algorithms rely on equality checks. The Similix[3] partial evaluator for Scheme can handle higher order values by doing a closure analysis and closure conversion which allows closures to be compared for equality. However the technique is not straightforward, for example, while Birkedal and Welinder covered almost every aspect of Standard ML they chose to avoid specialisation with respect to higher-order values (see [2] section 3.1.4).

3.4 Lack of implementations

The lack of practical and usable implementations is an obvious restraint on the widespread application of partial evaluation techniques. It is not for the lack of demand however. For example in [9] for the lack of a partial evaluator for Haskell, Hudak resorts to translating into Scheme, using a partial evaluation tool for Scheme and then translating back into Haskell.

Of course the lack of implementations stems partly from the previously mentioned problems but there is also the difficulty of dealing with a real language rather than a toy subset. Using a well behaved subset of a language or designing some simple language allows the research to concentrate more on advancing the theory and techniques but means that any tools produced cannot be applied by others to real programs. The notable exceptions are Similix[3] and Birkedal and Welinder's thesis on partial evaluation for Standard ML[2]. One advantage of targeting Haskell over SML or Scheme is that it is semantically simpler as it lacks mutable references and side effects.

4 Current Prototype

The current prototype transforms explicitly binding time annotated programs into their generating extensions. There is no binding time checking so binding time errors manifest as type errors in the generating extension. The prototype only works for annotated programs that do not manipulate higher order static data. There is some limited support for partially static data.

The current prototype works for a range of small examples using non-primitive recursion, mutual recursion and nested recursion. It also works for a simple language interpreter and a few examples of numerical programs that manipulate partially static lists.

The cogen function is split into two phases, conversion to memoising form and then encoding. A binding time checking or inference phase would be added at the beginning.

4.1 Template Haskell

The current prototype makes substantial use of the Template Haskell[19] infrastructure implemented in GHC[8]. Template Haskell is a language extension that allows compile-time meta-programming. It allows the programmer to get a representation of selected fragments of code, to perform arbitrary manipulation on that code and then to "splice" the result back into their program.

It extends Haskell with two extra syntactic constructs: quoting brackets `[| ... |]` and splice brackets written `$( ... )`. The value of a quoted fragment of code is the abstract syntax tree for that code represented as an ordinary Haskell data value. Because of the need to rename identifiers the value is returned in the Q monad. That is it has type `Q Exp`. A splice bracket computes the value of the enclosed expression (which must be code, that is it must be of type `Q Exp`) and inserts the code into the program in place of the splice brackets. The code that is spliced in is typechecked in the context of where it is inserted[6], so forming ill-typed programs is not possible.

The rest of the system is implemented by a library. The library provides the definition of the abstract syntax tree and the Q monad. The Q monad provides unique name generation, error reporting and to a limited extent allows names to be looked up in the compiler's symbol maps to get information about definitions and data types. It also provides a class of types that support a `lift :: a -> Exp a` operation which lifts values to code that would evaluate to that value. This also proves useful in the implementation.

Using the Template Haskell infrastructure confers a number of advantages. There are the straight-forward obvious advantages that TH provides an abstract syntax tree, a parser, pretty printer and a monad providing unique name supply and error reporting. Because it is part of a compiler, the compiler also does the other ordinary semantic checks including typechecking. This is obviously useful since our partial evaluator can only be expected to work for well formed and typed Haskell programs and so we are spared from performing these checks. A less obvious advantage is that we may be able to do binding time checking in terms of a slightly modified Haskell type system and implement it using the existing compiler's type checker rather than having to write one from scratch (See section 4.4.2).

The most useful feature is that it facilitates the cogen approach by allowing code to be run at compile time to generate code. This is precisely what the splice brackets `$( ... )` do. This allows the partial evaluator to be implemented as a library rather than as a separate preprocessor. This should make such a partial evaluator more practical since distributing

[6] Unfortunately this means that type errors are reported in terms of the generated code rather than the code that generated the ill-typed code. There is on-going research to address this problem[15].

and maintaining a library is typically easier than a source code preprocessor. As an example, recall from section 1.11 the construction of a compiler from an interpreter. With an implementation of *cogen* using Template Haskell we can write this example directly

```
compiler :: Prog -> Exp (State -> Value)
compiler = $(cogen [| runProg |])
```

Another advantage of TH over a custom infrastructure is that because it is more widely known it provides a commonly understood notation for describing staged computation in Haskell. Since it is an existing working system and is suitable for writing generating extensions directly, it makes experimentation easier. One can first write the desired generating extension for a function, then test it and then work backwards to find a suitable transformation procedure.

4.1.1 Alternatives

The obvious alternative would be to implement a partial evaluator as a source to source preprocessor. There are existing Haskell parsers and pretty printers available but one would need to implement other semantic checks. Using such a preprocessor would not be very easy since the compiler needs to be invoked once on the result of the preprocessor (the generating extension), then that compiled program would need to be invoked and the result of that (the residual program) compiled again. In contrast, when using TH the whole procedure can be accomplished automatically by the system when it compiles an expression of the form[7]:

```
$( $(cogen [| prog |]) static ) dynamic
```

Another alternative would be to perform partial evaluation not on Haskell source code but on GHC Core language. This is an explicitly typed intermediate language. It has a much simpler abstract syntax than Haskell which would be an advantage and the easy availability of type information may prove useful. However the result would not be as easy to read as Haskell source code, especially by an ordinary programmer. This is fairly important since programmers will want to see the result of specialisation to check that they are getting what they expected or desired.

A totally different technique would be to use Type Directed Partial Evaluation (TDPE). It is a way of doing partial evaluation based on the structure of the type of the program rather than the syntax of the program. It claims a couple advantages over conventional syntax directed techniques: that it can specialise types as well as values which leaves fewer redundant injections and projections in residual programs and also that it can work on compiled representations of programs rather than their abstract syntax which would have practical benefits such as making separate compilation easier. A disadvantage is that TDPE currently only works for monomorphic programs and supports only monovariant binding times. This might be fine for applying to whole/closed programs which have a monomorphic type but it would be a particular drawback for application to libraries which by their nature are program fragments and are used in many different contexts which makes polyvariant binding times particularly valuable.

4.1.2 Example

As an example, consider the Ackermann function which we will specialise on its first argument. Figure 3 shows the original version and two possible versions of its generating

[7] The current version of TH rejects nested splice brackets but the same effect can be achieved by using a named intermediate value.

Figure 3: The Ackermann function, its incorrect and correct generating extensions

```
ack :: Int -> Int -> Int
ack m n =
  if m == 0
    then n + 1
    else if n ==0
           then ack (m-1) 1
           else ack (m-1) (ack m (n-1))

genex_ack' :: Int -> ExpQ
genex_ack' n =
  if m == 0
    then [| \n -> n + 1 |]
    else [| \n -> if n == 0
                    then $(genex_ack' (m-1)) 1
                    else $(genex_ack' (m-1)) ($(genex_ack' m) (n-1))
          |]

genex_ack :: Int -> ExpQ
genex_ack n =
  if m == 0
    then [| \n -> n + 1 |]
    else [| let genex_ack_m = \n ->
                  if n == 0
                    then $(genex_ack (m-1)) 1
                    else $(genex_ack (m-1)) (genex_ack_m (n-1))
              in genex_ack_m
          |]
```

extension. However the first will fail to terminate at compile time for any interesting value of its only parameter. This is because the Ackermann function has a non-primitive recursion pattern such that it cannot be fully unrolled; the emitted code must contain loops. The second version does exactly this, it emits the appropriate loops in the residual program. This is discouraging since writing the generating extension would seem to require a careful analysis of the pattern of recursion of the function to be specialised. However by applying the methods of partial evaluation we can avoid this manual analysis. Section 4.3 describes the appropriate general form of the generating extension.

4.1.3 Binding time annotations

The current implementation requires fully explicit binding time annotations as there is no binding time inference yet. There are three kinds of annotation that are required, annotating static expressions, dynamic expressions and static expressions appearing in a dynamic context. One common way of providing these annotations is to have two versions of each basic syntactic construct, so for example a static "if" and a dynamic "if"; a static fixpoint and a dynamic fixpoint. Because we are using an existing general purpose system we cannot modify its syntax to such a degree. The solution that I have adopted in the current implementation is "dummy" function application. We define a function for each of the annotations and we make the rule that the annotation applies to the expression to which the function is *syntactically* applied. These annotations should be considered as syntactic

constructs rather than as functions. So that the annotations do not interfere with anything, they are given the identity function type[8].

It should be noted that this form of annotation is rather different to the form where each construct comes in static and dynamic flavours because the annotations are like brackets and they may nest. For example a dynamic expression occurring in a static context leads to a dynamic annotation occurring inside a static annotation application. There is undoubtedly a relationship between the two forms of annotation (at least for well annotated programs) however I have not characterised it precisely. It may be useful to do so since the former form may be easier for users to work with. One can imagine a binding time editor that allows users to highlight and mark constructs or expressions as static or dynamic.

The nested annotation form is rather closer to an implementation as there is a one-one relation between the annotations and TH's quote and splice brackets. A valid Template Haskel program can be obtained by substituting the static annotation for splice brackets and the dynamic annotation for quasi quotes. Of course this is before we have performed the memoising transformation (section 4.3) so as in figure 3 the program may not be what we want.

4.2 Double encodings

Template Haskell is a meta-language for Haskell but not for itself. That is, it can represent the syntax of Haskell programs but not the syntax of Template Haskell programs. The abstract syntax data types have no constructors for the special Template Haskell syntax - the quasi-quote and splice brackets. This is an issue since the generating extensions are most naturally expressed as Template Haskell programs; after all generating extensions are Haskell programs that generate code which is exactly what Template Haskell was designed to support.

Note that the double encoding of code described here should not be confused with the double encoding of values that plagued some typed partial evaluators (see page 11). In contrast we are only using a double encoding of parts of the program during the construction of the generating extension. Once the generating extension is constructed, it manipulates values with no encoding overhead. In particular the bloat introduced by the double encoding is proportional only to the size of the program being specialised and not proportional to the running time of the specialisation phase.

Although we cannot represent Template Haskell programs directly in the TH abstract syntax, we can use an encoding. As mentioned above, our intermediate representation contains annotations to indicate what parts of the program are static and which are dynamic. In the encoding phase we can consider these annotations to be the abstract syntax representation for the Template Haskell splice and quasi-quote brackets. The job of the encoding phase is to convert this intermediate representation into valid Haskell code. No annotation functions should remain in the final output.

The key to understanding the encoding is to note that quasi-quote brackets lift the level of encoding by one, that is it converts an term with type $\texttt{Exp}^n\, e$ to a term with type $\texttt{Exp}^{n+1}\, e$. The splice brackets do the exact opposite, they lower the level of encoding by one. Since we are already working with an encoding of a program we will be selectively raising the encoding from $\texttt{Exp}^1\, e$ to $\texttt{Exp}^2\, e$. So when we say we are doubly encoding parts of the program we mean that we lift the level of encoding from 1 to 2, not that we lift it by 2 levels.

Where we would like to emit a quasi-quoted expression we should instead doubly encode it. This operation is best explained by giving a type and an equational definition:

[8] In section 4.4.2 we suggest changing these types to help implement binding time checking.

```
quasi_quote :: Exp a -> Exp (Exp a)
$( quasi_quote [| code |] ) = [| code |]
```

The reverse operation would be difficult to implement but fortunately we do not need it. Quasi-quotes and splice brackets can nest alternately inside each other. An obvious algorithm would be to doubly encode everything in place of a quasi quote, then inside this we would perform the reverse if we encountered a splice bracket. However we can save work by simply not doubly encoding inside of a splice that is within a quasi quote. This is the only case we need to consider since we will never generate code that has a splice at the top level. This is because the top level of the program is a dynamic context. We choose this convention since it means that if the program contains no static/dynamic annotations then producing the generating extension is the identity operation.

To see that this encoding does what we want, consider a quasi-quoted expression that contains a splice.

```
[| let x = $( lift (primes !! 1000) ) in x |]
```

If we want to emit this code in the generating extension then the code in the splice will need to run at compile time while the rest will be deferred until run time. This implies that the code in the splice should appear as ordinary code so that it will be evaluated and produce code when the generating extension is run. The rest of the code in the quasi quote will not run at compile time but will be deferred to run time, thus at compile time it must produce the code that will be executed at run time, so it must be doubly encoded.

The algorithm to implement the encoding scheme is as follows: we traverse the abstract syntax tree and when in a dynamic context we doubly encode the syntax tree and when in a static context we simply traverse it without making any changes. So the algorithm flip-flops between two states, dynamic encoding context and static traversing context.

There is also the matter of `liftstatic`, the annotation for static computations appearing in a dynamic context. These are translated into a call to the TH function `lift` that lifts values to code. To see that this is the right translation recall what the `liftstatic` annotation is for. It is for static values that appear in otherwise dynamic contexts. We will calculate the value of the static expression at compile time, but because the value is required at runtime we need a way of "lifting" the value which is known at compile time so that it is available in the residual program. The obvious solution is that the value should be embedded as a constant in the source text of the residual program. To do this however requires that we convert from a raw value to abstract syntax that can be included in the residual program. The same issue crops up with Template Haskell programs and so the TH system provides a `Lift` class with a `lift` function.

The actual double encoding is fairly simple; for each of the constructors in the TH abstract syntax tree we generate code that returns that constructor applied to encodings of its arguments. As an example consider an "if" expression. We start with the abstract syntax for the "if"expression: the constructor `IfE` and its three arguments

```
encodeExp (IfE condition true-branch false-branch) = ...
```

The result must be code for a program that applies the `IfE` constructor to something. So that gives us

```
[| IfE |] `AppE` ...
```

Algorithm 1 Encoding algorithm

```
encodeExp :: Exp -> ExpQ
encodeExp (VarE name)    = [| varE |] `AppE` encodeName name
encodeExp (VarE name)    = [| varE |] `AppE` encodeName name
encodeExp (ConE name)    = [| conE |] `AppE` encodeName name
encodeExp (LitE lit)     = [| litE |] `AppE` encodeLiteral lit
encodeExp (AppE (VarE name) e)
   | name == 'static     = traverseExp e   --do not encode in static context
   | name == 'dynamic    = error "dynamic expression in dynamic context"
   | name == 'liftstatic = [| lift |] `AppE` traverseExp e
encodeExp (AppE e1 e2)   = [| appE |] `AppE` encodeExp e1 `AppE` encodeExp e2

traverseExp :: Exp -> ExpQ
traverseExp (VarE name)  = varE name
traverseExp (ConE name)  = conE name
traverseExp (LitE lit)   = litE lit
traverseExp (AppE (VarE name) e)
   | name == 'dynamic    = encodeExp e     --encode in dynamic context
   | name == 'static     = error "static expression in static context"
   | name == 'liftstatic = error "liftstatic expression in static context"
traverseExp (AppE e1 e2) = appE (traverseExp e1) (traverseExp e2)
```

`AppE` is the abstract syntax tree constructor for function application. As we said earlier the code must apply the `IfE` constructor to encodings of its arguments. In the case of "if" these are all expressions too so we use `encodeExp` recursively

```
[| IfE |] `AppE` encodeExp condition
          `AppE` encodeExp true-branch
          `AppE` encodeExp false-branch
```

The implementation for the other constructors follows the same regular pattern. This, rather tedious code, could probably be machine generated or written more concisely using some generic programming technique. The overall algorithm can be implemented as two tree traversals, one for the dynamic context and one for the static context. Control flips between them when annotations are encountered. Algorithm 1 shows the interesting part of the code.

We can see that this algorithm enforces the restriction on the nesting of `static`, `dynamic` and `liftstatic`. Static and dynamic contexts but nest alternately. `liftstatic` may only appear in dynamic contexts and its body is considered a static context.

While this encoding scheme is fairly simple to describe and implement, I do not have a formal specification of the transformation (beyond the operational description given by the algorithm). In section 4.4.1 we consider the relationship between the types of original annotated programs and their generating extensions with the aim of deriving a type system for the original annotated programs from the ordinary Haskell type system applied to the generating extensions. Having a suitable specification of the encoding transformation would no doubt help with this task.

Figure 4: Specialisation of the Ackermann function to $m = 4$

```
let ack_0 = \n_1 -> if n_1 == 0
                    then ack_2 1
                    else ack_2 (ack_0 (n_1 - 1))
    ack_2 = \n_1 -> if n_1 == 0
                    then ack_3 1
                    else ack_3 (ack_2 (n_1 - 1))
    ack_3 = \n_1 -> if n_1 == 0
                    then ack_4 1
                    else ack_4 (ack_3 (n_1 - 1))
    ack_4 = \n_1 -> if n_1 == 0
                    then ack_5 1
                    else ack_5 (ack_4 (n_1 - 1))
    ack_5 = \n_6 -> n_6 + 1
 in ack_0
```

4.3 The generating extension transformation

4.3.1 The specialisation algorithm

In the traditional non-cogen approach, specialisation is typically done by the following general scheme: we keep a "pending" list of functions that need to be specialised along with the values that they will be specialised to. Another list of already specialised functions is also kept, each with the values they were specialised to. Specialisation proceeds by taking an item from the pending list, assigning it a unique name and running the static parts to yield residual code that is added to the "done" list. As the static parts are run, where calls to other static functions are encountered, we do not simply continue into the function being called. Instead, it is looked up in the done list, taking into account both the name of the function and the values of the static arguments. If an exact match is found a call to the specialised function is emitted, if not the function and the static arguments to which it should be specialised is added to the pending list.

The algorithm is initialised with an empty done list and the pending list containing just the initial top-level function call with its static arguments. When the pending list becomes empty the main phase of the algorithm is finished. The final residual program is constructed by taking all the residual specialised functions from the done list and emitting them wrapped up in a recursive `let`. It is important to note that these residual functions are in general mutually recursive; in fact this is the whole point of the method. If we imagine a naive specialiser algorithm that just proceeds into specialising child calls without checking if it already exists on the done list: any loop in the residual functions is a point where the naive specialiser would have failed to terminate. For example consider again Ackermann's function from figure 3. The middle, incorrect, version implements the naive specialisation algorithm we have just described. It generates an infinite amount of code for any $m \neq 0$. In comparison, the algorithm described in this section produces the output[9] in figure 4 when we specialise the Ackermann function to $m = 4$. As one can see the algorithm has produced five specialised versions, one for each value of m. Also, each one is self-recursive, which is what causes the naive specialiser to loop, and each one also calls the next except for the base case of $m = 0$.

[9]This is the actual output from the current implementation, except that for clarity, fully qualified names have been removed (otherwise it would be `Prelude.==`, `Prelude.-` etc.).

Note that if we omit the memoising transformation and just do the encoding phase we will get exactly the naive generating extension which will fail if there are any loops.

It is sometimes beneficial to use separate pending and done lists for each function being specialised. In a typed language it is necessary since lists must have a homogeneous element type while the tuple of free variables has a different type for each function. It also has the advantage that we use many short lists rather than one large list so the lookups are quicker. Furthermore, in the cogen approach it becomes possible to statically select the pending and done list in use at any one point, while in the traditional interpretive approach this choice must be made dynamically.

The algorithm described thus far does not take into account nested `lets`, it implicitly assumes we only have top level functions. It is not hard to extend to allow named functions to be declared in `lets` anywhere in an expression. We essentially repeat the whole algorithm for each `let` construct. We create new pending and done lists for each function in the `let` group and start specialising the body of the `let`, which if it makes calls to functions defined in the `let` will cause specialised versions of them to be generated. When the local pending list(s) are empty the specialised functions from the done list(s) are collected and together with the specialised body, returned wrapped in a `let` construct. Of course the functions defined in the `let` we are considering may make calls to functions defined in higher levels of scope. We just do the ordinary procedure; these calls must be recorded in the appropriate function's pending list if they are new values of that function's static variables, or a call to a previously specialised version must be generated. It is worth noting that these functions in a higher level of scope cannot re-enter the `let` block that we are currently considering because any such call would already be on the done list and the name of previously specialised version would be returned. As this is the case it is safe to finalise the residual code that is generated by the `let` block before the program as a whole has finished being specialised. This simplifies the control flow pattern of the specialisation algorithm which is an important consideration as we shall see.

In the cogen method, specialisation proceeds in a similar way using "pending" and "done" lists. The difference of course is that in the cogen method we have to generate code that manipulates these lists rather than manipulating them directly. Because of this extra step of removal it is important that the structure of the code that does the specialisation be simple so that this code is not too complex to generate. Not all the code has to be generated however, some code can be put into a library that the generating extension will import, in which case we only need to generate calls to these functions. In fact Template Haskell makes this particularly easy as such library functions can be included in the same library that implements the partial evaluation functions; they do not have to be explicitly imported into the generated programs. If the work to be done by the generating extension can be suitably partitioned into "generic" code and code that needs to be customised at each instance where it is used, then much work can be saved by using these library functions since it allows the generating code to be significantly simpler. The fact that Haskell has a polymorphic type system which allows a certain amount genericity helps in this respect. In fact a language with even better generic programming facilities might make the task even easier though one would have to consider the trade-off with efficiency[10]. In the current implementation the major piece of "generic" code that has been separated out is the `memoise` function (see algorithm 2).

[10] This issue is an example of specialisation in itself; the consideration of deferring work from generation time to more general code at compile time. In this instance it makes sense to advocate generalisation because we do not consider the performance of the compile-time code to be very important.

4.3.2 Implementation of the specialisation algorithm

So now we must actually face the issue of how to construct a program that implements the specialisation algorithm, with its complicated control flow pattern and "pending" and "done" mappings. We would like a simple template that we can generate easily. It is obviously desirable for the generating extension to be as syntactically similar as possible to the original program since the effort in writing the transformation that produces the generating extension will be more or less related to the degree of syntactic changes that we have to make. A purely functional implementation is clearly possible but a good deal of plumbing would need to be added to each function to accept, modify and return the pending and done lists. What we would like is implicit arguments. In a language with imperative features we can use mutable variables to give the effect of extra arguments and return values. In a purely functional language the equivalent technique is to use monads. Fortunately we are already working in a monad: Template Haskell's Q monad. This is because all the dynamic parts of the program, those that return code, do so in the Q monad.

We use a method due to Birkedal and Welinder[2] who implemented it for ML using mutable references. We use the same technique using "`QRefs`" in the Q monad. `QRefs` are mutable references that live in the Q monad. There are operations to read, write and to construct new references. The general idea of the method is that for each named function in the original there is a corresponding one in the generating extension that generates specialised versions of the original. That is it accepts the static arguments of the original function and returns code. For example, suppose we had a function in the original `f :: Int -> Int -> Int` where the first argument is static and the second argument and the result are dynamic, then the generating extension will have a function `f :: Int -> ExpQ`[11]. Of course this is not everything, we still have not dealt with the done and pending lists.

We use `QRefs` to hold the done list. The algorithm does not actually require a pending list because instead of adding new calls to specialise to the done list we just recuse directly into the function. That is we follow a depth-first rather than breadth-first path. This is easier to implement because it means there is no need to generate code that adds new cases to the pending list, however we need to be very careful to update the done list appropriately to deal with recursive calls properly. There is the danger that we would not record the fact that we have specialised a particular function to some value of its static arguments until that function call has completed, however if it made a recursive call with the same static arguments then it would loop endlessly. It is these subtle manipulations of the done list that is performed by the `memoise` function (see algorithm 2). We can then replace each function by its `memoise`d version.

We call it `memoise` because it behaves in a similar way to ordinary memo functions in that it keeps a cache of previous argument-result pairs and only runs the underlying function when the argument is not in the cache. In this case the list is not a cache, it is the done list. For reasons that will become clear, the done list is split into two lists that can be updated independently. Both are kept in a `QRef` that is passed as `memoise`'s first argument. The next argument is the name of the function that we are memoising. The reason for using this is so that the specialised versions of the function we are specialising can be given names related to the original which considerably aids readability of the residual code. The next argument is the function itself which takes an input value to code. We can deal with functions of any arity if we assume that the function is in uncurried form. At the call site will will have to deal with the uncurrying and also passing the name of the function. The last argument is the current static arguments to the function we are specialising. `memoise` starts by looking up this value in the done list. More specifically, the first part of the done list is a mapping from the static arguments to specialised function names. If the value is found then the corresponding name is returned and we are done. If the value is not found then we will

[11] Note that `ExpQ` is just a synonym for `Q Exp`.

Algorithm 2 Memoise function

```
memoise :: Eq value => QRef ([(value, Name)], [DecQ])
        -> String -> (value -> ExpQ) -> value -> ExpQ
memoise seen baseName f x = do
  (seen', decls) <-  readQRef seen
  case lookup x seen' of
    (Just name) -> varE name
    Nothing -> do
                 name <- newName baseName
                 modifyQRef seen (\(seen, decls) -> ((x, name):seen, decls))
                 body <- f x
                 let decl = valD (varP name) (normalB $ return body) []
                 modifyQRef seen (\(seen,decls) -> (seen, decl:decls))
                 varE name
```

have to call the function and specialise it for this new tuple of static arguments. We start by generating a new name for this specialised version. As we noted before, this name is based on the name of the function we are specialising. Now before we call the function we record the value of the static arguments and the new function's name in the done list. This is to deal with the issue we identified earlier where a recursive call could cause our algorithm to loop. If a recursive call is made with the same values of static arguments then the value will be in the done list and the already generated name will be returned. We then make the actual call and obtain the specialised residual code for the function with the current tuple of static arguments. We make a simple declaration by binding the specialised function name to its residual code and add it to the list of specialised functions that makes up the second component of the done list.

Note that here is where the first order restriction is introduced. The memoisation introduces a requirement that the values can be compared for equality. Equality is needed to lookup the values in the done list.

As we noted earlier, replacing each function in the let block with its memoised version is not quite as simple as just applying `memoise`. Finding the name of the function being called is easy, but doing the uncurrying is rather harder. The problem is that we do not actually know the arity of the functions we are memoising. We can easily inspect the number of arguments that the function takes syntactically but this is not always the same (for example if we write in a point-free style). To do this properly requires type information for the program (see section 4.4). In the current implementation we just make the requirement that the functions are fully eta-expanded and so the number of syntactic arguments is the same as the arity. This is a deficiency that should be improved upon in future work (see section 5.1.1). To aid readability of the generating extension and because the syntactic transformations involved are smaller, we do not apply `memoise` to each function's body, rather we rename each function and bind the original name to the application of `memoise` on the renamed function, see figure 5.

Now all that remains is to implement the rest of the algorithm described a few paragraphs back; we must initialise the done list, start the main specialisation phase and then collect all the residual specialised functions at the end and wrap them in a let block. The general form of the transformation is given in figure 5. See also appendix A.

An infidelity of the current implementation is that it rules out any static function that both accepts code and produces code because the type of the input code will be `Q Exp` which cannot be compared for equality because of the need for a Q environment. This could be solved by making the input code have type `Exp` rather than `Q Exp`. Such a change would

Figure 5: Memoising transformation

```
let f x   = E1
    g x y = E2
in E3
```

The previous code is transformed into the following

```
do
  seen_1 <- newQRef ([], [])
  seen_2 <- newQRef ([], [])
  let f  x   = memoise seen_1 "f" (\(x) -> f' x) (x)
      f' x   = E1
      g  x y = memoise seen_2 "g" (\(x, y) -> f' x y) (x, y)
      g' x y = E2
  exp <- E3
  (_, decls_1) <- readQRef seen_1
  (_, decls_2) <- readQRef seen_2
  letE (decls_1 ++ decls_2) (return exp)
```

require changes in the encoding phase and some changes in the memoising transformation to insert calls to `return` in appropriate places.

4.4 Type inference

One problem with the system described thus far is that is assumes that all static `let`s are values or functions that return code. This is obviously not the case and rules out ordinary `let`s that for example just give a name to a common subexpression. The memoisation transformation applies the `memoise` function to each function in a static let and the `memoise` function assumes that the function returns code. So for Static $\rightarrow$ Static functions this results in a type error in the generating extension. The obvious solution is to not memoise functions which to not yield a dynamic result however implementing this test requires that we know the binding times of each function.

Finding out this information amounts to type checking and because we are using a higher order language this requires an inference algorithm. As we noted earlier, applying the `memoise` function properly requires knowing the arity of the function to which we are applying it, which also requires type inference.

So it is clear that we need type information for the program we are specialising. Unfortunately, the current Template Haskell system does not give us type information for quoted programs, just the abstract syntax[12]. However, even if it did give us type information it is not clear that the ordinary Haskell type system fits all our needs. If the `dynamic` and `static` annotations are "transparent" to the type system (because they have type $id = \forall t.t \rightarrow t$) then knowing the types in the original program tells us nothing about the binding time types in our program. We may require a modified type system and corresponding inference algorithm.

[12]This may change in the future as there is considerable interest in such a feature.

4.4.1 Towards binding-time checking

We can make a precise specification about the conversion from an annotated program to its generating extension, which is: assuming the original program is well typed and well annotated, the generating extension will be well typed. To put it another way, if one makes a mistake in the binding time annotations it will be manifest as a type error in the generating extension, because one will end up in the generating extension with an `Exp` type where an unencoded expression should be, or vice versa.

However understanding the cause of the error from the type error message in the generating extension is very difficult. It would obviously be better to be able to check that the annotations are correct before applying the generating extension transformation and if they are not correct to be able to give a more helpful message. This is the problem of checking if a division is congruent. The normal technique is to look at each part of each expression to check that there are no static expressions depending on dynamic expressions.

Since we have already specified the correctness in terms of types of the result it seems we may be able to derive a check upon the original program also in terms of types. We obviously have an operational specification which is to just perform the translation algorithm and check the result with the ordinary Haskell type rules. It would be better however, to have a reasonably formal extensional specification. As we have noted, the generating extension is ordinary Haskell and is typed according to the ordinary Haskell type system, so what we want is to derive a type system for the preimage of the generating extension transformation. It is useful to note that the memoisation phase does not change the types of any of the code it manipulates, it is only the encoding phase that does.

4.4.2 A type system for binding-time-annotated programs

Our typing rules for this "Haskell with binding time annotations" language will obviously be based upon the standard typing rules since the language contains ordinary Haskell as a subset. The difference is that will have to assign portions of the code `Exp` types based on the `static`, `dynamic` and `liftstatic` annotations.

Note, for the rest of this section we will use Template Haskell notation rather than `static/dynamic` annotations because it is less confusing.

At first it would seem that is would suffice to say that splice brackets takes syntax to semantics and quasi-quotes do the opposite. That is to say we could extend the ordinary Haskell typing rules with these extra rules

$$\frac{\varepsilon \vdash e : a}{\varepsilon \vdash [|\, e \,|] : Exp\, a}$$

$$\frac{\varepsilon \vdash e : Exp\, a}{\varepsilon \vdash \$(\, e\,) : a}$$

While applications of "lift" are implicit in the TH syntax, we must make them explicit. Note that only types from the `Lift` class can be lifted, so in particular function types cannot be lifted.

$$\frac{\varepsilon \vdash e : a \quad a \in Lift}{\varepsilon \vdash \mathrm{lift}\, e : Exp\, a}$$

However these rules are not sufficient. We can still write programs that according to these rules are type correct and yet their generating extensions have type errors. Consider the expression

$$\varepsilon \vdash \mathrm{let}\ x = 3\ \mathrm{in}\ [|\, \lambda y . x + y \,|] : Exp\ (\mathrm{Int} \rightarrow \mathrm{Int})$$

. This expression is well typed according to the rules we have just defined. Here is the full type proof tree

$$\dfrac{\varepsilon\,[x \mapsto \mathrm{Int}] \vdash 3 : \mathrm{Int} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\varepsilon \vdash (+) : \mathrm{Int} \to (\mathrm{Int} \to \mathrm{Int}) \qquad \varepsilon\,[x \mapsto \mathrm{Int}] \vdash x : \mathrm{Int}}{\varepsilon\,[x \mapsto \mathrm{Int}] \vdash ((+)\ x) : \mathrm{Int} \to \mathrm{Int}} \qquad \varepsilon\,[y \mapsto \mathrm{Int}] \vdash y : \mathrm{Int}}{\varepsilon\,[x \mapsto \mathrm{Int}, y \mapsto \mathrm{Int}] \vdash ((+)\ x)\ y : \mathrm{Int}}}{\varepsilon\,[x \mapsto \mathrm{Int}] \vdash \lambda y.x + y : \mathrm{Int} \to \mathrm{Int}}}{\varepsilon\,[x \mapsto \mathrm{Int}] \vdash [|\,\lambda y.x + y\,|] : Exp\ (\mathrm{Int} \to \mathrm{Int})}}{\varepsilon \vdash \mathrm{let}\, x = 3 \text{ in } [|\,\lambda y.x + y\,|] : Exp\ (\mathrm{Int} \to \mathrm{Int})}$$

However the expression should be rejected, the intended expression is

$$\varepsilon \vdash \mathrm{let}\, x = 3 \text{ in } [|\,\lambda y.\mathrm{lift}\, x + y\,|] : Exp\ (\mathrm{Int} \to \mathrm{Int})$$

Since the x is a static value appearing in a dynamic context it needs to be lifted to code otherwise the generating extension will not type, or rather it will use undeclared variables.

$$\begin{array}{l} genex\ (\mathrm{rec}\, x = 3 \text{ in } [|\,\lambda y.x + y\,|]) \\ = \\ \mathrm{let}\, x = 3 \text{ in LamE}\,'y'\,(\mathrm{InfixE}\ (\mathrm{VarE}\,'x')\,(\mathrm{VarE}\,'+')\,(\mathrm{VarE}\,'y')) \end{array}$$

Running this program gives the code

$$\lambda y.x + y$$

at which point we can see that this is not a closed term. We have lost the 3 and there is no x variable.

The translation of the correct version

$$\begin{array}{l} genex\ (\mathrm{let}\, x = 3 \text{ in } [|\,\lambda y.\mathrm{lift}\, x + y\,|]) \\ = \\ \mathrm{let}\, x = 3 \text{ in LamE}\,'y'\,(\mathrm{InfixE}\ (\mathrm{lift}\, x)\,(\mathrm{VarE}\,'+')\,(\mathrm{VarE}\,'y')) \end{array}$$

Running this program gives us the code

$$\lambda y.3 + y$$

which is what we intended. The problem seems to be that as we type terms inside quasi-quote brackets [| |] we do it in isolation, we are not "aware" that we're in an encoded context as so do not notice the error of applying an encoding of a function to a non-encoded argument (in this case $+$ to x). So a solution perhaps is to remember the context we are in so that we can detect and disallow variables to be defined in one context and used in another.

We take the ordinary typing rules but add an extra component to the typing judgements, a "context". The form of judgements is

$$\overline{\varepsilon, c \vdash e : T}$$

where $c \in \{S, D\}$. The extra component tells us if we are in a static or dynamic context. We can consider lift context to be the same as static. We also record the context in each variable binding in the environment. Environment bindings now have the form $\varepsilon\,[x \mapsto_c a]$. For most rules the context does not matter except that it be the same in the antecedents and consequents. The context is determined purely syntactically, it just tells us if we are inside splice or quasi quotes. The context of each variable binding in the environment is

determined by the context in which the variable was bound. The rule for using a variable says that the context in which the variable was bound must be the same is the context in which it is now being used.

$$\frac{}{\varepsilon, c \vdash b : \text{Bool}}$$

$$\frac{}{\varepsilon, c \vdash n : \text{Int}}$$

$$\frac{}{\varepsilon\,[x \mapsto_c a]\,, c \vdash x : a}$$

$$\frac{\varepsilon, c \vdash f : a \to b \quad \varepsilon, c \vdash x : a}{\varepsilon, c \vdash f\,x : b}$$

$$\frac{\varepsilon\,[x \mapsto_c a]\,, c \vdash e : b}{\varepsilon, c \vdash \lambda x.e : a \to b}$$

$$\frac{\varepsilon, c \vdash e_1 : \text{Bool} \quad \varepsilon, c \vdash e_2, e_3 : a}{\varepsilon, c \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : a}$$

$$\frac{\varepsilon, c\,[x \mapsto_c a] \vdash e_1 : a \quad \varepsilon\,[x \mapsto_c a]\,, c \vdash e_2 : b}{\varepsilon, c \vdash \text{let } x = e_1 \text{ in } e_2 : b}$$

13

$$\frac{\varepsilon, S \vdash e : a}{\varepsilon, D \vdash [|\,e\,|] : Exp\,a}$$

$$\frac{\varepsilon, D \vdash e : Exp\,a}{\varepsilon, S \vdash \$(\,e\,) : a}$$

$$\frac{\varepsilon, S \vdash e : a \quad a \in Lift}{\varepsilon, D \vdash \text{lift}\,e : Exp\,a}$$

Checking programs against this type system requires a type inference algorithm as does ordinary Haskell type checking. However it may be possible to implement this check as a combination of an ordinary Haskell type check followed (or indeed preceded) by an additional check on the consistency of contexts and variable bindings. This is because the rules about splice brackets, quasi quotes and lift can be expressed as ordinary Haskell functions.

```
static, lift :: Exp a -> a
dynamic      :: a -> Exp a
```

The additional check would not need to use an inference algorithm, it could simply traverse the program collecting variable bindings and their context, then check that they are used in a matching context.

I have proposed that this type system satisfies the criteria I set out at the beginning of the section, namely that annotated programs that are typeable under this system have well typed generating extensions, however I have not given a formal or semi-formal justification of this. I would aim to formalise this argument in a final thesis.

[13] Details of generalisation in the let rule omitted.

4.4.3 The current implementation

We now have several reasons for wanting type information: proper handling of applications of `memoise`, the decisions of whether or not to memoise at all and also to perform binding time checking. For these reasons we include a type inference algorithm in the partial evaluator.

This current implementation does not include binding time checking, though as described in the previous section we are not far from being able to add it. It just requires an extra pass to check that variables are being used in contexts consistent with their declaration. However for the binding time checker to be useful to users it needs to give helpful error messages, which would obviously require more work.

For the current implementation we must assume that the program is already type correct, we just need to recover the type information for the program. The current implementation uses a modified version of Mark Jones's Typing Haskell in Haskell[11] type checker. To integrate it into the implementation it has been modified to work with Template Haskell, in particular to use the TH abstract syntax and to use the Q monad for its unique name supply and global type environment. The checks on type classes and type's kinds were removed because Template Haskell cannot provide sufficient information to implement them (though this may change in future revisions of Template Haskell). This is not a problem however because we already assume that the compiler has already type checked the code and this would include kinds and type classes.

4.5 Partially static data structures

Up to this point we have considered values to be either wholly static or wholly dynamic. There are many examples however where it would be useful to take advantage of the fact that parts of a data structure are static and parts are dynamic. A simple example is a pair where the first component is static and the second component is dynamic. If in some static part of the program, the first component is extracted then this can be done statically where as if the whole pair where classified as dynamic then the projection and any expression depending on it would also have to be classified as dynamic.

As well as partially static product types it is also possible to have partially static sum types. It means we know statically on which side of the disjoint union the value lies and so case expressions that deconstruct the value can be classified as static. Dealing with partially static product and sum types opens a large range of common data structures to greater specialisation opportunities. The most obvious example is lists.

A useful case is when the length of the list is known statically but the values of the elements of the list are not known statically. This case corresponds to the disjoint sum (between *nil* and *cons*) being static and the first element in the *cons* product being dynamic and the second element static.

There is a connection here with the domain theoretic description of recursive product and sum types, and in particular the subtleties in describing types that may contain values of bounded or unbounded size. It these descriptions it all comes down to the treatment of bottom $\bot$ when forming products and sums. The difference is whether we choose representations that identify or distinguish $\bot =_? \bot \times y$, $\bot =_? x \times \bot$ for products and $\bot =_? \bot + y$, $\bot =_? x + \bot$ for sums. Strict or "smash" sums and products do not allow infinite components while lazy or "lifted" ones do. We would define lazy lists as the least fixed point of the following domain equation

$$List\,X = \mathbf{1} + X_{\uparrow} \times (List\,X)_{\uparrow}$$

where both sides of the product are lifted. While if we want to ban unbounded lists we just make the the tail of the list unlifted or strict. That way an unbounded tail makes the whole list bottom.

$$FinList\,X = \mathbf{1} + X_{\uparrow} \times FinList\,X$$

The analogy here is that strict components are like static values, we will fully compute them at compile time and they had better not be bottom, while the lifted components are like the dynamic values, they are allowed to compute bottom because we never execute the dynamic code at compile time.

Another way of describing partially static structures is using types. If a fully dynamic list has type $Exp\ [\alpha]$ then a list of static length but with dynamic elements has type $[Exp\,\alpha]$. The problem with this notation is that there is no way to express dynamic components other than those that are parameters of the type constructor. For example suppose we had a data type

```
data IntOrSomething a = AnInt Int | SomethingElse a
```

We can easily express that the only dynamic component is the something else `IntOrSomething (Exp a)` but if we wanted the int to be dynamic then there is no type to express this. Indeed this would be a real problem in an implementation because the generating extension must be well typed and it would be a type error to, in this case, use a `Exp` type where the data constructor requires an `Int`. It may be necessary to change or duplicate data type definitions or simply to disallow partially static types that are inconsistent with the data type definition. This is an issue that will need further investigation.

Of course a great many examples fall into the category of "fixed shape data structure, dynamic contents" so it practice it may not be too much of a problem since all common collection types are parameterised by their element type(s).

Let us start with the example of summing a fixed length list; first of all an ordinary Haskell version. An approach worth considering is to use the ordinary algorithm but to make the recursion be controlled by a static argument rather than the dynamic list. Here is what we might like to write

```
vsum :: Int -> [Int] -> [Int] -> [Int]
vsum 0 []     []     = []
vsum n (x:xs) (y:ys) = x + y : vsum (n-1) xs ys
```

We have added an extra argument `n` that determines if we make a recursive call or if we have reached the base case. If the length of the supplied lists did not exactly match `n`, the function would fail. We can certainly write this function and indeed specialises it for any value of n, but it does not give us much advantage since the list itself is still dynamic and so the pattern matching still needs to be done. Indeed, upon reflection this must always be the case, if what we start out with is a fully dynamic list then we must dynamically deconstruct it, (though the shape that we pattern match against may well be static). To get a significant benefit from knowing the length of the list we must start with a genuinely partially static list. We may be able to perform a series operations that use static lists and have the intermediate static lists specialised away. The further we can push back the need to convert to or from fully dynamic lists the more specialisation opportunities are available. For example we may be able to use static lists with dynamic elements everywhere within some program and only have to convert to or from fully dynamic lists when we read the program input or write the output. So we can split the problem of dealing with partially static lists into two: how to convert a dynamic lists to a static list and how to manipulate them once we have them.

As a small example, consider adding three vectors of known length. If we use `vsum` above

```
vsum n (vsum n xs ys) zs
```

we will end up generating a dynamic list as the result of `(vsum n xs ys)` only for the next call of `vsum` to deconstruct this list. But we know there is no need to build and tear down these intermediate lists since they have the same static length. Each element of the final list can be expressed as a simple combination of the elements of the three input lists with no intermediate lists required.

So let us assume that we start with a static list with dynamic elements and we want to do the same vector summing operation

```
vsum' :: [ExpQ] -> [ExpQ] -> [ExpQ]
vsum' []     []     = []
vsum' (x:xs) (y:ys) = [| $x + $y |] : vsum' xs ys
```

Now if we try the same example of summing three vectors of known length we get exactly what we want

```
        [ [| 1 |], [| 2 |], [| 3 |] ]
`vsum'` [ [| 4 |], [| 5 |], [| 6 |] ]
`vsum'` [ [| 7 |], [| 8 |], [| 9 |] ]
 = [ [| (1+4)+7 |], [| (2+5)+8 |], [| (3+6)+9 |] ]
```

We can apply this same technique to slightly larger examples such as matrix multiplication with much the same result. We have used Template Haskell notation here but equally we could use `static/dynamic` annotations. When using `static/dynamic` annotations, this example relies on the type inference extension discussed in the previous section so that the `vsum'` function is not memoised; `vsum'` returns `[ExpQ]` not `ExpQ` so it is type-incompatible with the `memoise` function.

A remaining problem is how to convert a dynamic list to a static list with dynamic elements. Firstly we must know how long the static list will be. The trick is similar to our first `vsum` function in that we will deconstruct a dynamic list under the control of a static integer argument. The difference is that instead of rebuilding another dynamic list we build a static list that consists of the *variables* that we use in pattern matching on the dynamic list. We can write down what the ideal code would be in Template Haskell notation

```
case xs of [x1, x2, x3] -> f [ [| x1 |], [| x2 |], [| x3 |] ]
```

where `f` is some function that operates on static lists with dynamic elements. If we put this together with `vsum'` we can write down our ideal residual code for the example as a whole

```
case xs of [x1, x2, x3] ->
case ys of [y1, y2, y3] ->
case zs of [z1, z2, z3] ->
  [(x1 + y1) + z1,
   (x2 + y2) + z2,
   (x3 + y3) + z3]
```

Now we are not going to be able to generate the complex patterns in these case expressions because they are not a syntactic form that can be built recursively, but we could generate a series of nested case expressions that each decompose one list element at a time, which is just as good.

The most tricky issue is that the pattern variables are only valid within the scope of the case expression. We must somehow return both the static list of variables and the enclosing case expression. While we might like to write the following simple code, it will not work

```
fixLength :: Int -> ExpQ -> [ExpQ]
fixLength n exp = case exp of []       -> []
fixLength n exp = case exp of (x : xs) -> x : fixLength xs
```

There is no way of annotating this code with `static/dynamic` that will be consistent with the type we want. We must do the case decomposition dynamically but then the `dynamic` would have to extend over the body of the case expression too which gives us an `ExpQ` overall, not a list `[ExpQ]`.

It is possible using a more convoluted form that takes a "continuation" function to apply in the body. That way we return an `ExpQ` overall and can make sure that the pattern variables will remain within the scope of the case expression

```
fixLength' :: Int -> ExpQ -> ([ExpQ] -> ExpQ) -> [ExpQ] -> ExpQ
fixLength' 0 exp body accum = [| case $exp of [] -> $(body (reverse accum)) |]
fixLength' n exp body accum = [| case $exp of
   (x:xs) -> $(fixLength' (n-1) [|xs|] body ([|x|]:accum)) |]
```

This does translate directly into a `static/dynamic` annotated version, but is still totally impractical since the form is so convoluted. We cannot expect users to rewrite their code to this degree. Again, the problem here is the need to return both the list of static variables and to make sure they stay within the scope of the case expression. If we could eliminate the need to carry around an enclosing scope that everything must fit into then we could write our code in a more straightforward fashion. What we need is for each element in the list to be an independent expression that can be moved about without regard for any special bits of context code that bind variables in the expression. Fortunately, this is entirely possible, instead of using case expressions to deconstruct values we can use projection functions.

Let us contrast the methods. The first needs a case expression and makes the pattern variables available in the body.

```
case xs of (x:xs') -> ...[| x |]...[| xs' |]...
```

The second method uses projection functions so each expression is self contained.

```
[| head xs |], [| tail xs |]
```

So for our `fixLength` function we use `head` and `tail` functions

```
fixLength'' :: Int -> ExpQ -> [ExpQ]
fixLength'' 0 exp = []
fixLength'' n exp = [| head($exp) |] : fixLength'' (n-1) [| tail($exp) |]
```

This looks much better; it has a straightforward form. However, if we rewrite this function slightly we can see the downside to this approach.

```
fixLength'' :: Int -> ExpQ -> [ExpQ]
fixLength'' 0 exp = []
fixLength'' n exp = [| exp!!n |] : fixLength'' (n-1) exp
```

If we use `fixLength''` we are going to generate many nested calls to tail which is just list indexing. Accessing each element of the list is independently going to traverse the list, which is obviously inefficient. It is clear why this should be the case, while a case expression binds all the pattern variables in one go, applying projection functions will do the pattern match each time they are called.

It seems likely that some special support in the partial evaluator will be required to be able to retain the efficiency of case expressions while allowing the flexibility of using projection functions. Perhaps an extra annotation to denote a static pattern match against a dynamic expression would be helpful so the partial evaluator can invoke some special behaviour in this situation. One idea is that the pattern variables introduced could be tracked and case expressions inserted at appropriate locations in the residual program. One would have to be careful that such a technique does not alter the strictness properties of the program. This is a topic in need of further investigation.

4.6 Structure of the program

There are three phases: binding type inference, the memoising transformation and encoding. The last two phases could easily be combined but it is helpful to keep them separate as it aids reasoning and testing. Furthermore reading programs emitted after the encoding phase is rather difficult.

The type inference phase builds a map of identifiers to types but does not otherwise alter the input program. The memoising transform phase then modifies the code, using the type information to help in some situations. Finally the encoding phase yields the final generating extension. So the overall process is just a simple composition of the three phases.

```
genex f = encode (memoiseTransform (typeCheck f) f)
```

The program is implemented as a library. It has a very limited interface. It just exports the `genex` function[14] and the three annotations `dynamic`, `static` and `liftstatic`.

5 Thesis Plan

This section lays out the subject for a final thesis and a plan for how to get there.

The core claim of the thesis will be that partial evaluation can provide a better payoff/difficulty balance than other generative programming techniques. I intend to demonstrate this by building a practical partial evaluation tool for the language Haskell98 with common language extensions and to apply it to domain-specific embedded languages and other active libraries.

The intention is to be driven by the applications, that it to add features to the tool implementation only when the applications require it. Otherwise there are far too many potential features and I would only be recreating previous work. However there are some features that we can predict will be useful for the applications I have stated; these are discussed in section 5.1.

[14] For development and testing purposes the components of the `genex` function are currently also exported.

5.1 Further work and features

5.1.1 Correct existing deficiencies

Section 4 already mentions a few aspects of the current tool which are less than satisfactory. These are deficiencies which have a reasonably clear solution but which I have not yet had time to improve.

In section 4.3.2 we noted that in general type information is required to apply the memoising function correctly, otherwise we must restrict ourselves to an annoying syntactic convention. The current implementation does gather type information but does not yet use it in this case to do the transformation. This should not be a difficult feature to implement. It is mostly a matter of adding some more plumbing to get the type information to the appropriate part of the algorithm.

Also in section 4.3.2 we note that the current implementation of the encoding transformation means that static functions that take code as an input (such as some functions that work on partially static data) get that code as an `Q Exp` type. Unfortunately this means that if the function also returns code and so gets memoised we end up with a type error because `Q Exp` is not an instance of the equality class as is required. This passing of code as input parameters wrapped in the Q monad is unnecessary. The implementation should be changed to pass just `Exp` types which would solve the problem as that type is in the equality class.

5.1.2 Proper binding time checking

Section 4.4.1 discusses the need for binding time checking and an approach to performing the check by type checking with a slightly modified type system.

This area will need further reading and research to give a more formal basis to the check. It may happen that the binding time checking can be subsumed into the binding time inference, so I plan to investigate these issues simultaneously.

5.1.3 User assisted binding time inference

Crucial to success of this project is the ability of the user to control which parts of their program get specialised whilst not having to fully specify every minor detail. Traditional binding time inference tries to work fully automatically and there are many clever tricks proposed in the literature to improve the results of the inference. The goal for this project as far as binding time inference is concerned must be to devise an algorithm that gives predictable, if not necessarily clever, results and takes into account users binding time annotations so that users can apply their knowledge of the problem domain and better control what parts of their program are specialised.

It may also be useful to be able to give various levels of detail in terms of binding time annotations. The current use of static/dynamic annotations in the expressions themselves are fully precise annotations. Less intrusive but correspondingly less precise annotations might be to annotate function declarations or data declarations. There is research to suggest that this is sufficient in many cases (see [21] section 4, last paragraph).

Since user control and understanding is important it may also prove useful to build some simple support tools. For example a tool which shows the result of the binding time inference and allows the user to play with adding annotations to see how that affects the result of the inference would help users to annotate their libraries in accordance with their goals. Such a tool is sometimes called a “binding time debugger”.

5.1.4 Higher order values

A significant limitation of the current system is that programs cannot manipulate higher order static data because the specialisation algorithm uses equality checks and functions cannot be compared for equality. This is a significant limitation because ordinary idiomatic Haskell code uses higher order functions quite heavily. This is especially true of domain-specific embedded languages which we hope to be the main application for this project. One of the main reasons that Haskell is touted as a good host language for these DSELs is its support for higher order functions.

Fortunately there are potential solutions. The Scheme partial evaluator Similix[3] handles higher order values by converting to explicit closures. A similar technique in Haskell is employed in the debugging tool Buddha[17], albeit for a different purpose. The idea is to convert every occurrence of a higher order value to a data value that wraps the function but itself can be compared for equality.

```
data Fun a b = Fun !Unique (a -> b)
instance Eq Fun where
  (Fun fId f) == (Fun gId g) = fId == gId
```

It also requires modifying code that passes or receives higher order values to wrap and unwrap them in the appropriate places. This is the aspect that will likely prove most difficult from an implementation point of view since it requires access to all the source code so that a modified version can be produced.

5.1.5 Support for partially static data

Section 4.5 discusses the utility of partially static data structures and how the idea can be supported in an implementation. There is work left to be done to make this feature user friendly and efficient. It is not a priority in and of itself but if applications require this feature then I will invest more time in finding a satisfactory solution. I do not imagine that I will make the binding time inference discover partially static data values. If it is a required feature it will probably only be available by manual annotations.

An example where it may be necessary is in the environments of interpreters for embedded languages. The environment can be naturally thought of as a partially static structure with static labels but dynamic contents. Alternatively the environment can be two separate structures which gives a cleaner binding time separation but is more awkward to use.

5.1.6 Possible extra features

To achieve better speed improvements when specialising language interpreters, it may prove useful to implement a phase that can remove from the residual program the majority of the conversions to and from a universal type used in an interpreter. Past research[16] has shown that this can achieve optimal partial evaluation; optimal in the sense of removing all the interpretive overhead when specialising an interpreter.

Another post-processing phase can be used to compress the duplication of code that is inherent in partial evaluation. It uses a form of common sub-expression elimination. Although this might be a nice feature, it is only a technique that we would apply if we found that a significant class of examples would benefit greatly from it. As noted before, working in the context of an optimising compiler alleviates these code quality problems to some extent.

5.1.7 Laziness

Haskell is a typed lazy language. I have covered the issues raised by types in some detail but I have not given as great consideration to the implications of laziness on partial evaluation. If ignored, partial evaluation techniques introduce additional strictness which can introduce additional cases of non-termination in the generating extension. This is unfortunate as partial evaluation is supposed to be semantics preserving. However the non-termination is in some sense safe since - I conjecture - it is only introduced to the specialisation phase and not into the residual program.

I intend to consider more carefully the cases where strictness (and thus potential non-termination) is introduced and to try to formalise this. It may be possible to develop an analysis to determine where it is safe and non-termination will be avoided, however it is more in keeping with the philosophy of this project to document clearly how binding time annotations introduce strictness and potential non-termination and to make the user aware of the issue.

There are potential cases of non-termination even without laziness such as static loops controlled by static data (or no data at all) so for a usable tool there needs to be pragmatic support to detect the easily detectable cases and abort with an informative error message. An easily detectable cases is when the generating extension produces an infinite residual program since we can simply put an upper limit on the number of specialised clauses allowed.

5.2 Comparative research

Type directed partial evaluation is a very different approach to partial evaluation than the one presented here. To compare it properly to my current approach, in particular the advantages and limitations of the latest TDPE techniques, I will need to review more of the recent papers in this area.

Meta-OCaml is a version of OCaml that allows one to write multi-stage programs. Staged programs are very similar to binding time annotated programs, so having a better understanding of the Meta-OCaml type system should give some insight into the type system I have been considering for binding time annotated Haskell programs.

When I come to design the binding time inference phase I will need to review in greater depth the literature on binding time analysis since a useful semi-automated binding time analysis is at the core of my claim about the practicality of partial evaluation.

5.3 Applications

As I mentioned the intention is for this to be an application driven project. The goal is to demonstrate an easier technique for writing efficient domain-specific embedded languages and other active libraries.

There are many existing DSELs (for parsing, pretty printing, graphics, animation, GUIs, simulation and hardware design and verification) and there are others that do not get written because interpretative overheads would make performance unacceptable. In his paper[9] on building DSLs in a modular fashion, Hudak comments that “the lack of a good partial evaluator for Haskell remains as the one stumbling block to more effective use of our overall methodology”. For other DSLs people sometimes have had to resort to heavyweight techniques such as generating C code from an original specification in a DSEL. This cancels out much of the savings in using a host language if a whole code generation back-end has to be written.

A convincing demonstration would be to re-create - with the same performance improvements - some of the partial evaluation examples in Hudak's DSLs paper[9], in which for the lack of a partial evaluator for Haskell he had to use a tool for Scheme and do manual translations.

Another example would be to re-implement PAN[6] but use partial evaluation to try to achieve speedups over the naive Haskell implementation. It is unlikely that this would rival the performance of the original implementation which applied many traditional optimisations and generated C code, however a satisfactory demonstration would be to achieve a reasonable fraction of the original's speedups but with a great deal less effort.

It would be interesting to apply the tool to some of the generic programming techniques that have been developed for Haskell in recent years. The straightforward implementation of these techniques often incurs a great deal of overhead because of redundant run-time checks or conversions. However this may require features like type specialisation that the tool does not cover since the primary applications probably do not require such a feature.

Another potential example to investigate might be specialisation of combinator libraries based on certain kinds of arrows. Some arrow-style combinators can collect information on the structure of the computation. This information is obviously static and so may provide opportunities for optimisation by partial evaluation. In essence it allows library authors to enforce good binding time separation on the users of the library which should enable better predictions of the speedups obtained in users code by specialisation. Monadic combinators do not have this property since they allow values lifted into the monad to affect the path of later computation. This application would rely heavily on support for specialising programs that use higher order static data.

5.4 Work plan

I intend to start with two applications, Hudak's modular DSLs and PAN. I will use these to drive extensions to the partial evaluator tool. Both examples will need support for higher order static data and also binding time checking is a practical necessity for any examples of a non-trivial size. In the course of developing these features and applications, other minor existing deficiencies will need to be corrected.

I anticipate that a useful, usable and user controllable binding time inference algorithm will take substantial time to research and develop. While the extra background reading for this can begin immediately, I think it would be good to delay the detailed development work so that the experience of tackling a couple applications can inform the requirements of the user-oriented aspects of the algorithm.

A Examples

This section presents some examples of generating extensions produced by applying the code transformation functions to the annotated code.

As noted before the transformation is made of two phases, the memoising transformation and the encoding phase. In the current implementation they are implemented by `(traverse . genex)`.

A.0.1 Anatomy of the examples

```
genex_foo :: A -> ExpQ
genex_foo = $( (traverse . genex)
  [| \a ->
     let foo x = ...
      in foo a |] )
```

Overall, the function is the generating extension, that is it takes the static arguments to code for the residual program. It has type `A -> ExpQ`. It is formed by taking the abstract syntax for the annotated function `[| \a -> let foo x = ... in foo a |]`, applying the syntactic transformations `(traverse . genex)` and splicing the result in as the generating extension `genex_foo = $( ... )`.

A.1 Simple functions

A.1.1 Power function

This is just the simple power function specialised on the exponent argument.

```
genex_power :: Int -> ExpQ
genex_power = $( (traverse . genex)
  [| \a ->
     let power n =
          if n == 0 then dynamic(\x -> 1)
                    else dynamic(\x -> x * static(power (n-1)) x)
      in power a |] )
```

We can evaluate this function to get the specialised version for any value of n we choose. Here is the result of `genex_power 3`:

```
let power_0 = \x_1 -> x_1 GHC.Num.* power_2 x_1
    power_2 = \x_1 -> x_1 GHC.Num.* power_3 x_1
    power_3 = \x_1 -> x_1 GHC.Num.* power_4 x_1
    power_4 = \x_5 -> 1
 in power_0
```

This is the actual pretty-printed output. As one can see the built-in Template Haskell pretty printer fully qualifies symbols with the module they come from (`GHC.Num.*`). In later examples we will trim this off for readability. Also note that the numbers used to make unique names have no relation to the values of the static variables. The numbers are just generated sequentially.

It should be clear from this example what was noted earlier about not employing any unfolding strategy. If we employed even a simple conservative unfolding strategy, this example could be unfolded to the much more readable

```
\x_1 -> x_1 * (x_1 * (x_1 * 1))
```

However as we said, for the time being we are relying on the compiler's inlining phase to perform this transformation. It will be necessary at some point to investigate how well the compiler is able to do this kind of unfolding, especially for more complicated examples.

Here is our old favourite again, the Ackermann function's generating extension for specialisation on the m parameter.

```
genex_ack :: Int -> ExpQ
genex_ack = $( (traverse . genex)
  [| \a ->
     let ack 0 = dynamic(\n -> n+1)
         ack m = dynamic(\n ->
                   if n == 0 then static(ack (m-1)) 1
                             else static(ack (m-1)) (static(ack m) (n-1)))
      in ack a |] )
```

And the specialisation for m = 3, that is `genex_ack 3`:

```
let ack_0 = \n_1 -> if n_1 == 0
                      then ack_2 1
                      else ack_2 (ack_0 (n_1 - 1))
    ack_2 = \n_1 -> if n_1 == 0
                      then ack_3 1
                      else ack_3 (ack_2 (n_1 - 1))
    ack_3 = \n_1 -> if n_1 == 0
                      then ack_4 1
                      else ack_4 (ack_3 (n_1 - 1))
    ack_4 = \n_5 -> n_5 + 1
 in ack_0
```

A.2 Interpreter

In this example we specialise an interpreter for a trivial imperative language. The language has global integer variables, assignment, conditional and looping statements and simple arithmetic expressions. Here are the data types representing the abstract syntax tree

```
data Stmt = Var := Expr
          | If Expr Stmts Stmts
          | While Expr Stmts
          | Return Expr         --Final program return value

data Expr = Const Int        | Var Var
          | Expr :+:  Expr   | Expr :-:  Expr
          | Expr :*:  Expr
          | Expr :==: Expr   | Expr :/=: Expr
          | Expr :>:  Expr   | Expr :>=: Expr
          | Expr :<:  Expr   | Expr :<=: Expr
```

The intention in this exercise was to write a totally naive interpreter and then to see what the minimum changes would be to achieve reasonable specialisation. The initial version deliberately uses a very simple data structure for the environment/store:

```
type Value = Int             --Bools represented by zero/non-zero integer
type State = [(Var, Value)]
```

The interpreter itself consists of just five functions. There are two to manipulate the environment/store and a semantic function for each of the syntactic components and one top level function to put everything together.

```
update :: Var -> State -> Value -> State
lookup :: Var -> State -> Value

evalProg  :: Stmts -> State -> Value
evalStmts :: Stmts -> State -> Value
evalExpr  :: Expr  -> State -> Value
```

We can annotate these functions and produce a generating extension. We can take a program in our little language and compile it into Haskell by running the generating extension with the program as input

```
prog = [ While (Var "x" :/=: Const 0)
            [ "x" := Var "x" :-: Const 1 ],
         Return (Var "x") ]
```

Unfortunately the residual program does not exhibit very good specialisation

```
let update_0 = let update'_1 [] e_2 = [(['x'], e_2)]
                   update'_1 (v'_3, e'_4) : s_5 e_6 =
                     if ['x'] == v'_3
                       then (v'_3, e_6) : s_5
                       else (v'_3, e'_4) : update'_1 s_5 e_6
                in update'_1
    lookup_7 = \st_8 -> let lookup'_9 [] = error $ "uninitialised variable "
                                                ++ show ['x']
                            lookup'_9 (v'_10, e_11) : s_12 =
                              if ['x'] == v'_10
                                then e_11
                                else lookup'_9 s_12
                         in lookup'_9 st_8
    evalProg_13 = \initialState_14 -> evalStmts_15 initialState_14
    evalStmts_15 = \st_16 -> if evalExpr_17 st_16 /= 0
                               then evalStmts_18 st_16
                               else evalStmts_19 st_16
    evalStmts_19 = \st_20 -> evalExpr_21 st_20
    evalStmts_18 = \st_22 -> evalStmts_15 (update_0 st_22 (evalExpr_23 st_22))
    evalExpr_23 = \st_24 -> evalExpr_21 st_24 - evalExpr_25 st_24
    evalExpr_25 = \st_26 -> 1
    evalExpr_17 = \st_27 -> fromBool $ evalExpr_21 st_27 /= evalExpr_28 st_27
    evalExpr_28 = \st_26 -> 0
    evalExpr_21 = \st_29 -> lookup_7 st_29
 in evalProg_13
```

In particular, while the evaluation of the arithmetic expressions has been compiled into Haskell, the manipulations of the store are still done dynamically and in terms of variable names. This is not very surprising however since in annotating the functions we had to annotate the store as totally dynamic even though it contains a mixture of static information (the variable names) and dynamic information (their values). A more subtle issue that is not shown with this example input program but becomes a problem with more complicated programs is the fact that the naive store update function adds variables to the environment as they are encountered in *program execution order*. Of course *we* know that the order of the variables in the environment list does not matter but the specialiser does not know this, so the residual program will contain code to manipulate the store for all the possible store orderings given the program's control flow.

We can fix both these problems. The latter problem can be fixed by changing the interpreter so that it always use only one variable order and in particular so that it depends only on the program abstract syntax and not on the program's input. The obvious solution is to just scan the program and collect all the variables used in it. The first problem about mixing the static aspects of the environment and store in one data structure can be addressed either by using a partial evaluator that can cope with partially static data structures or to split the static and dynamic parts into two structures. The current implementation's support for partially static data structures is not sufficiently mature to deal with this example so we must take the latter route and modify the interpreter to separate the environment and store. After splitting we have a static environment, that is a mapping of variables to store indices, and a dynamic store which is an array of values.

The modified signatures are as follows

```
type StateS = [Var]
type StateD = DiffUArray Int Value  --Array of unboxed Ints with O(1) updates

update :: Var -> StateS -> StateD -> Value -> StateD
lookup :: Var -> StateS -> StateD ->  Value

evalProg  :: Prog  -> [Value] -> Value
evalStmts :: Stmts -> StateS  -> StateD -> Value
evalExpr  :: Expr  -> StateS  -> StateD -> Value
collectVars :: Prog -> [Var]        --Additional function
```

Here are the annotated functions in full

```
genex_evalProg :: Prog -> ExpQ
genex_evalProg = $( (traverse . genex)
  [| \prog ->
  let
    update :: Var -> StateS -> StateD -> Value -> StateD
    update v vars = dynamic(\st e ->
      st // [(liftstatic(fromJust $ v `elemIndex` vars), e)])

    lookup :: Var -> StateS -> StateD ->  Value
    lookup v vars = dynamic(\st ->
      st!liftstatic(fromJust $ v `elemIndex` vars))

    evalProg :: Prog -> [Value] -> Value
    evalProg prog = dynamic(\initialState ->
      static(evalStmts prog (collectVars prog))
```

```
      (listArray (0, liftstatic(length (collectVars prog) - 1))
                 initialState))

  evalStmts :: Stmts -> StateS -> StateD -> Value
  evalStmts [] vars = dynamic(\st ->
    error "program terminated without a result")

  evalStmts ((v:=e) :ss)   vars = dynamic(\st ->
    static(evalStmts ss vars)
      (static(update v vars) st (static(evalExpr e vars) st)))

  evalStmts (If e a b :ss) vars = dynamic(\st ->
    if static(evalExpr e vars) st /= 0
      then static(evalStmts (a ++ ss) vars) st
      else static(evalStmts (b ++ ss) vars) st)

  evalStmts (While e s:ss) vars = dynamic(\st ->
    if static(evalExpr e vars) st /= 0
      then static(evalStmts (s ++ While e s:ss) vars) st
      else static(evalStmts ss vars) st)

  evalStmts (Return e :ss) vars = dynamic(\st ->
    static(evalExpr e vars) st)

  evalExpr :: Expr -> StateS -> StateD -> Value
  evalExpr (Const c)  vars = dynamic(\st -> liftstatic(c))
  evalExpr (Var v)    vars = dynamic(\st -> static(lookup v vars) st)
  evalExpr (a :+:  b) vars = dynamic(\st ->
    static(evalExpr a vars) st +  static(evalExpr b vars) st)
  evalExpr (a :-:  b) vars = dynamic(\st ->
    static(evalExpr a vars) st -  static(evalExpr b vars) st)
  evalExpr (a :==: b) vars = dynamic(\st ->
    fromBool $ static(evalExpr a vars) st == static(evalExpr b vars) st)
  ... -- remaining clauses omitted for brevity

 in evalProg prog [] )
```

Now when we specialise to the same program we get much better residual code.

```
let update_0 = \st_1 e_2 -> st_1 // [(0, e_2)]
    lookup_3 = \st_4 -> st_4 ! 0
    evalProg_5 = \initialState_6 ->
                   evalStmts_7 (listArray (0, 0) initialState_6)
    evalStmts_7 = \st_8 -> if evalExpr_9 st_8 /= 0
                             then evalStmts_10 st_8
                             else evalStmts_11 st_8
    evalStmts_11 = \st_12 -> evalExpr_13 st_12
    evalStmts_10 = \st_14 -> evalStmts_7 (update_0 st_14 (evalExpr_15 st_14))
    evalExpr_15 = \st_16 -> evalExpr_13 st_16 - evalExpr_17 st_16
    evalExpr_17 = \st_18 -> 1
    evalExpr_9 = \st_19 -> fromBool $ evalExpr_13 st_19 /= evalExpr_20 st_19
    evalExpr_20 = \st_18 -> 0
    evalExpr_13 = \st_21 -> lookup_3 st_21
 in evalProg_5
```

Notice that now we have compiled expressions, control flow and environment / store manipulations into Haskell. A criticism we could make is that we had to pick a more sophisticated data structure (an $O(1)$ update array) to get reasonable residual code. If we had stuck with a store represented by a list then we would have had list indexing in the residual program. There is a feature that can be added to a partial evaluator to deal with this problem called *arity raising* where a collection value can be split into a collection of individual variables which are passed as extra parameters as appropriate (or discarded if they are not used). This technique can achieve the $O(1)$ updates that we would like in interpreter examples.

An interesting outcome of this exercise is that we notice that to get good specialisation we had to think about the design of our interpreter more carefully. In particular we made the same observation that introductory compiler texts make, that the environment can be split from the store so that variables can be statically allocated in the store and the compiler does all necessary manipulations of the environment leaving none for the compiled program. In light of the fact that we are actually building a compiler it is not surprising that we have to consider these issues and structure our interpreter accordingly. In retrospect it is quite pleasing that the issue crops up and can be solved in such a simple context (as opposed to all the details involved in a compiler).

A.3 Numerical examples

These examples are of simple numerical operations on vectors and matrices where we specialise on the size of the vector/matrix. This is a technique used in some C++ numerics libraries to generate optimal straight-line code for small vectors and matrix operations.

A.3.1 Without using partially static structures

These first few examples do not make use of partially static data structures so while indexes can be calculated statically, the collection structure itself must be wholly dynamic. The first example is a simple operation: vector inner product specialised to a fixed length vector. Here is the original code

```
innerProduct :: Num a => Int -> [a] -> [a] -> a
innerProduct 0  _      _     = 0
innerProduct n (x:xs) (y:ys) = x * y + innerProduct (n-1) xs ys
```

As we said, while the n variable can be static, the two lists must be wholly dynamic.

```
genex_innerProduct :: Int -> ExpQ
genex_innerProduct = $( (traverse . genex)
  [| \n -> let innerProduct 0 = dynamic(\ _      _     -> 0)
               innerProduct n = dynamic(\(x:xs) (y:ys) ->
                                  x * y + static(innerProduct (n-1)) xs ys)
           in innerProduct n |])
```

As we expect the residual code will contain code that deconstructs the lists at runtime. Here is the code for n = 3:

```
let innerProduct_0 = \(x_1 : xs_2) (y_3 : ys_4) -> x_1 * y_3 + innerProduct_5 xs_2 ys_4
    innerProduct_5 = \(x_1 : xs_2) (y_3 : ys_4) -> x_1 * y_3 + innerProduct_6 xs_2 ys_4
    innerProduct_6 = \(x_1 : xs_2) (y_3 : ys_4) -> x_1 * y_3 + innerProduct_7 xs_2 ys_4
    innerProduct_7 = \_ _ -> 0
 in innerProduct_0
```

If instead of a list we use an array then the only dynamic code is array indexing which is efficient.

```
genex_innerProductArray :: Int -> ExpQ
genex_innerProductArray = $( (traverse . genex)
  [| \n ->
     let innerProductArray n = dynamic(\xs ys -> static(
           let innerProductArray' i
                 | i < n     = dynamic(xs!liftstatic i * ys!liftstatic i
                                       + static(innerProductArray' (i+1)))
                 | otherwise = dynamic(0)
            in innerProductArray' 0
          ))
      in innerProductArray n |])

let innerProductArray_0 = \xs_1 ys_2 ->
      let innerProductArray'_3 = xs_1!0 * ys_2!0 + innerProductArray'_4
          innerProductArray'_4 = xs_1!1 * ys_2!1 + innerProductArray'_5
          innerProductArray'_5 = xs_1!2 * ys_2!2 + innerProductArray'_6
          innerProductArray'_6 = 0
       in innerProductArray'_3
 in innerProductArray_0
```

We can do similar specialisation for matrix multiplication, again using arrays.

```
genex_matrixMult :: (Int,Int) -> (Int,Int) -> ExpQ
genex_matrixMult = $( (traverse . genex)
  [| \(ax, ay) (bx, by) ->
     let matrixMult (ax, ay) (bx, by) = dynamic(\a b -> static(
          let matrixMult' x y
                | x == ax   = matrixMult' 0 (y+1)
                | y == by   = dynamic([])
                | otherwise = dynamic(static(innerProduct x y 0)
                                      : static(matrixMult' (x+1) y))

              innerProduct x y z
                | z < ay    = dynamic(a!liftstatic(x,z) * b!liftstatic(z,y)
                                      + static(innerProduct x y (z+1)))
                | otherwise = dynamic(0)

           in dynamic(listArray (liftstatic((0::Int,0::Int), (ax-1,by-1)))
                                (static(matrixMult' 0 0)))
         ))
      in matrixMult (ax, ay) (bx, by) |] )
```

If we specialise this for 2×2 matrices we get

```
let matrixMult_0 = \a_1 b_2 ->
      let matrixMult'_3 = innerProduct_4 : matrixMult'_5
          matrixMult'_5 = innerProduct_6 : matrixMult'_7
          matrixMult'_7 = matrixMult'_8
          matrixMult'_8 = innerProduct_9 : matrixMult'_10
```

```
            matrixMult'_10 = innerProduct_11 : matrixMult'_12
            matrixMult'_12 = matrixMult'_13
            matrixMult'_13 = []
            innerProduct_11 = a_1!(1,0) * b_2!(0,1) + innerProduct_14
            innerProduct_14 = a_1!(1,1) * b_2!(1,1) + innerProduct_15
            innerProduct_15 = 0
            innerProduct_9  = a_1!(0,0) * b_2!(0,1) + innerProduct_16
            innerProduct_16 = a_1!(0,1) * b_2!(1,1) + innerProduct_17
            innerProduct_17 = 0
            innerProduct_6  = a_1!(1,0) * b_2!(0,0) + innerProduct_18
            innerProduct_18 = a_1!(1,1) * b_2!(1,0) + innerProduct_19
            innerProduct_19 = 0
            innerProduct_4  = a_1!(0,0) * b_2!(0,0) + innerProduct_20
            innerProduct_20 = a_1!(0,1) * b_2!(1,0) + innerProduct_21
            innerProduct_21 = 0
         in listArray ((0, 0), (1, 1)) matrixMult'_3
   in matrixMult_0
```

which after a bit of obvious unfolding is the functional equivalent of straight-line code. If we do this unfolding manually we get

```
\a_1 b_2 -> listArray ((0, 0), (1, 1))
             (a_1!(0,0) * b_2!(0,0) + (a_1!(0,1) * b_2!(1,0) + 0)
             :a_1!(1,0) * b_2!(0,0) + (a_1!(1,1) * b_2!(1,0) + 0)
             :a_1!(0,0) * b_2!(0,1) + (a_1!(0,1) * b_2!(1,1) + 0)
             :a_1!(1,0) * b_2!(0,1) + (a_1!(1,1) * b_2!(1,1) + 0)
             :[]
             )
```

A.3.2 Using partially static structures

Now we have an example making use of partially static lists. It is the vector summing example where we sum three vectors without using an intermediate dynamic list. This is a version using the more readable but less efficient version of converting dynamic lists to partially static lists. The result of this is that the residual code has lots of list indexing.

```
genex_vsum3 :: Int -> ExpQ
genex_vsum3 =  $( traverse
  [| \a ->
     let vsum []     []     = []
         vsum (x:xs) (y:ys) = dynamic( static(x) + static(y) ) : vsum xs ys

         fixLength 0 exp = []
         fixLength n exp = dynamic( static(exp)!!liftstatic(n-1) )
                             : fixLength (n-1) exp

         foo n = dynamic( \xs ys zs ->
                 static( (\xs' ys' zs' -> listE $ vsum xs' (vsum ys' zs'))
                           (fixLength n (dynamic xs))
                           (fixLength n (dynamic ys))
                           (fixLength n (dynamic zs))
                       ))
      in foo a |] )
```

Here is the result for vectors of length 3:

```
\xs_0 ys_1 zs_2 -> [xs_0!!2 + (ys_1!!2 + zs_2!!2),
                    xs_0!!1 + (ys_1!!1 + zs_2!!1),
                    xs_0!!0 + (ys_1!!0 + zs_2!!0)]
```

The alternative version using the more complex version of `fixLength` looks like this

```
genex_vsum3' :: Int -> ExpQ
genex_vsum3' =  $( traverse
  [| \a ->
     let vsum []     []     = []
         vsum (x:xs) (y:ys) = dynamic( static(x) + static(y) ) : vsum xs ys

         fixLength n exp body = fixLength' n exp body []
         fixLength' 0 exp body accum = dynamic(
           case (static exp) of []     -> static(body accum) )
         fixLength' n exp body accum = dynamic(
           case (static exp) of
             (x:xs) -> static(fixLength' (n-1) (dynamic xs) body ((dynamic x):accum)))

         foo n = dynamic( \xs ys zs ->
                 static(
                   fixLength n (dynamic xs) $ \xs' ->
                   fixLength n (dynamic ys) $ \ys' ->
                   fixLength n (dynamic zs) $ \zs' ->
                     listE $ vsum xs' (vsum ys' zs')
                       ))
      in foo a |] )
```

But the residual specialised code is more efficient because it uses `case` to decompose the dynamic lists rather than repeated list indexing.

```
\xs_0 ys_1 zs_2 ->
  case xs_0 of (x_3 : xs_4) ->
  case xs_4 of (x_5 : xs_6) ->
  case xs_6 of (x_7 : xs_8) ->
  case xs_8 of [] ->
  case ys_1  of (x_9  : xs_10) ->
  case xs_10 of (x_11 : xs_12) ->
  case xs_12 of (x_13 : xs_14) ->
  case xs_14 of [] ->
  case zs_2  of (x_15 : xs_16) ->
  case xs_16 of (x_17 : xs_18) ->
  case xs_18 of (x_19 : xs_20) ->
  case xs_20 of [] ->
    [x_7 + (x_13 + x_19),
     x_5 + (x_11 + x_17),
     x_3 + (x_9 + x_15)]
```

References

[1] Mads Ager, Oliver Danvy, Henning Rohde. On obtaining KMP string matcher by partial evaluation. Proceedings of the ASIAN symposium on partial evaluation and semantics based program manipulation, pages 32-46, September 2002.

[2] Lars Birkedal and Morten Welinder. Partial Evaluation of Standard ML. Masters Thesis (Revised version: Technical report 93/22). Department of Computer Science, University of Copenhagen. 1993.

[3] Anders Bondorf, Oliver Danvy, Jesper Jørgensen. Similix 5.1. Available from http://www.diku.dk/forskning/topps/activities/similix.html

[4] Silvano Dal-Zilio and John Hughes. A self-applicable partial evaluator for a subset of Haskell. August 1993. Available from http://www.cmi.univ-mrs.fr/~dalzilio/haskell-parteval.html

[5] Oliver Danvy. Type-Directed Partial Evaluation. The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 242-257, January 1996.

[6] Conal Elliot, Oege de Moor and Sigbjorn Finne. Compiling Embedded Languages. Journal of Functional Programming, 13(3), pages 455-481, May 2003.

[7] Yoshihiko Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. Systems, Computers, Controls. 2(5), pages 721-728, 1971. Reprinted in Higher-Order and Symbolic Computation, 12(4), pages 381-391, 1999.

[8] The Glasgow Haskell Compiler. Available from http://www.haskell.org/ghc/

[9] Paul Hudak. Modular Domain Specific Languages and Tools. Proceedings of Fifth International Conference on Software Reuse, pages 134-142, June 1998. Available from http://haskell.cs.yale.edu/yale/papers/icsr98/

[10] Paul Hudak. Building Domain-Specific Embedded Languages. ACM Computing Surveys, 28A(4), page 196, December 1996.

[11] Mark Jones. Typing Haskell in Haskell. In Proceedings of the 1999 Haskell Workshop, October 1999. November 2000 version available from http://www.cse.ogi.edu/~mpj/thih/

[12] Neil Jones, Carsten Gomard and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall International, 1993. Available from http://www.dina.dk/~sestoft/pebook/

[13] S.C. Kleene. Introduction to Metamathematics. Princeton, NJ: D. van Nostrand, 1952.

[14] John Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes (ed.), Functional Programming Languages and Computer Architectures, Volume 523 of Lecture Notes in Computer Science, pages 145-164, August 1991.

[15] Ian Lynagh. Typing Template Haskell: Soft Types. August 2004. Available from http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/

[16] Henning Makholm. On Jones-Optimal Specialisation for Strongly Typed Languages. Proceedings of Workshop on Semantics, Applications and Implementation of Program Generation, pages 129-148, September 2002. Available from http://www.macs.hw.ac.uk/~makholm/

[17] Bernard Pope and Lee Naish. Specialisation of Higher-Order Functions for Debugging. Electronic Notes in Theoretical Computer Science, Vol. 64, 2002.

[18] Sean Seefried, Manuel Chakravarty and Gabriele Keller. Optimising Embedded DSLs using Template Haskell. Third International Conference on Generative Programming and Component Engineering, July 2004. To appear.

[19] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In ACM SIGPLAN Haskell Workshop 2002, pages 1-16, October 2002.

[20] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, Error-correcting Combinator Parsers. Advanced Functional Programming, John Launchbury, Erik Meijer and Tim Sheard (Eds.), Springer-Verlag, LNCS-Tutorial 1129, pages 184-207, August 1996.

[21] Todd Veldhuizen. C++ Templates as Partial Evaluation. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 13-18, January 1999.

[22] Todd Veldhuizen and Andrew Lumsdane. Guaranteed Optimization: Proving Nullspace Properties of Compilers. Proceedings of the 2002 Static Analysis Symposium, volume 2477, pages 263-277. Available via http://www.cs.chalmers.se/˜tveldhui/

[23] Todd Veldhuizen and Dennis Gannon. Active Libraries: Rethinking the roles of compilers and libraries. SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, October 1998. Available via http://www.cs.chalmers.se/˜tveldhui/