Everything old is new again: Quoted Domain Specific Languages

Shayan Najd
The University of Edinburgh
sh.najd@ed.ac.uk

Sam Lindley
The University of Edinburgh
sam.lindley@ed.ac.uk

Josef Svenningsson
Chalmers University of Technology
josefs@chalmers.se

Philip Wadler
The University of Edinburgh
wadler@inf.ed.ac.uk

Abstract

Fashions come, go, return. We describe a new approach to domain specific languages (DSLs), called Quoted DSLs (QDSLs), that resurrects two old ideas: quotation, from McCarthy's Lisp of 1960, and the subformula property, from Gentzen's natural deduction of 1935. Quoted terms allow the domain specific language to share the syntax and type system of the host language. Normalising quoted terms ensures the subformula property, which guarantees that one can use higher-order code in the source while guaranteeing first-order code in the target and enables using types to guide fusion. We test our ideas by re-implementing Feldspar, which was originally implemented as an Embedded DSL (EDSL), as a QDSL; and we compare the QDSL and EDSL variants.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.1 [*Formal Definitions and Theory*]; D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords domain-specific language, DSL, EDSL, QDSL, embedded language, quotation, normalisation, subformula property

1. Introduction

Don't throw the past away
You might need it some rainy day
Dreams can come true again
When everything old is new again
– Peter Allen and Carole Sager

Implementing domain-specific languages (DSLs) via quotation is one of the oldest ideas in computing, going back at least to McCarthy's Lisp, which was introduced in 1960 and had macros as early as 1963. Today, a more fashionable technique is Embdedded DSLs (EDSLs), which may use shallow embedding, deep embedding, or a combination of the two. Our goal in this paper is to reinvigorate the idea of building DSLs via quotation, by introducing a new approach which we dub Quoted DSLs (QDSLs). A key feature of QDSLs is the use of normalisation to ensure the subformula property, first proposed by Gentzen in 1935.

Imitation is the sincerest of flattery.
— Charles Caleb Colton

Cheney et al. (2013) describes a DSL for language-integrated query in F# that translates into SQL. The approach relies on the key features of QDSL—quotation, normalisation of quoted terms, and the subformula property—and the paper conjectures that these may be useful in other settings. We are particularly interested in DSLs that perform staged computation, where at generation-time we use host code to compute target code that is to be executed at run-time.

Generality starts at two. Here we test the conjecture of Cheney et al. (2013) by reimplementing the EDSL Feldspar Axelsson et al. (2010) as a QDSL. We describe the key features of the design, and show that the performance of the two versions is comparable. We compare the QDSL and EDSL variants of Feldspar, and argue that, from the user's point of view, the QDSL approach may sometimes offer a considerable simplification as compared to the EDSL approach.

Davies and Pfenning (1996, 2001) also suggest quotation as a foundation for staged computation, and note a propositions-as-types connection between quotation and a modal logic; our type $Qt\ \ a$ corresponds to their type $\bigcirc a$. They also mention in passing the utility of normalising quoted terms, although they do not mention the subformula property.

The .NET Language-Integrated Query (LINQ) framework as used in C# and F#, and the Lightweight Modular Staging (LMS) framework as used in Scala, exhibit considerable overlap with the techniques described here. Notably, they use quotation to represent staged DSL programs, and they make use to a greater or lesser extent of normalisation. In F# LINQ quotation is indicated in the normal way (by writing quoted programs inside special symbols), while in C# LINQ and Scala LMS quotation is indicated by type inference (quoted terms are given a special type).

In short, some researchers and developers are already exploiting this technique. We propose QDSL as a name to capture the commonalities among these approaches, and we observe the utility of the subformula property in this context.

ICFP 2015, August 31–September 2, 2015, Vancouver, Canada..

http://dx.doi.org/10.1145/...

> Perhaps we may express the essential properties of such a normal proof by saying: it is not roundabout.
>
> — Gerhard Gentzen

Our approach exploits the fact that normalised terms satisfy the subformula property, first introduced in the context of natural deduction by Gentzen (1935), improved by Prawitz (1965), and applicable to lambda calculus via the principle of Propositions as Types (Howard 1980; Wadler 2015). The subformula property states that every proof can be put into a normal form, where the only propositions that appear in the proof are subformulas of the hypotheses and conclusion of the proof. The subformulas of a formula are its subparts; for instance, the subformulas of $A \to B$ are the formula itself and the subformulas of A and B. In our application, we will regard formulas as types.

The subformula property provides users of the DSL with useful guarantees, such as the following:

- they may write higher-order terms while guaranteeing to generate first-order code;
- they may write a sequence of loops over arrays while guaranteeing to generate code that fuses those loops;
- they may write intermediate terms of nested type while guaranteeing to generate code that operates on flat data.

The first two of these are used in this paper, while the third is central to Cheney et al. (2013).

The subformula property is closely related to conservativity. A conservativity result expresses that adding a feature to a system of logic, or to a programming language, does not make it more expressive. Consider intuitionistic logic with conjunction; conservativity states that adding implication to this logic proves no additional theorems that mention conjunction alone (that is, where implication does not appear in the statement of the theorem). Such a conservativity result is an immediate consequence of the subformula property; since the hypotheses and conjuction of the proof only mention conjunction, any proof, even if it uses implication, can be put into a normal form that only mentions conjunction.

Viewed through the lens of Proposition as Types, the conservativity result of implication over conjuction applies to well-typed terms built with function abstraction and application, and construction and destruction of pairs. It says that if the types of the free variables of the term and the type of the term itself only mention pairs, then the term normalises to a form that only mentions pairs; any internal use of functions will be eliminated during normalisation. Such a result generalises to the first bullet point above; see Proposition 4.4 in Section 4.

As another example, the third bullet point above corresponds to a standard conservativity result for databases, namely that nested queries are no more expressive than flat queries (Wong 1993). This conservativity result, as implied by the subformula property, is used central to (Cheney et al. 2013) to show that queries that use intermediate nesting can be translated to SQL, which only queries flat tables and does not support nesting of data.

The subformula property holds only for terms in normal form. Previous work, such as Cheney et al. (2013) uses a call-by-name normalisation algorithm that performs full beta-reduction, which may cause computations to be repeated. Here we present call-by-value and call-by-need normalisation algorithms, which guarantee to preserve sharing of computations. We also present a sharpened version of the subformula property, which we apply to characterise the circumstances under which a QDSL may guarantee to generate first-order code.

> Good artists copy, great artists steal. — Picasso

EDSL is great in part because it steals the type system of its host language. Arguably, QDSL is greater because it steals the type system, the syntax, and the normalisation rules of its host language.

In theory, an EDSL should also steal the syntax of its host language, but in practice the theft is often only partial. For instance, an EDSL such as Feldspar or Nicola, when embedded in Haskell, can exploit the overloading of Haskell so that arithmetic operations in both languages appear identical, but the same is not true of comparison or conditionals. In QDSL, of necessity the syntax of the host and embedded languages must be identical. For instance, this paper presents a QDSL variant of Feldspar, again in Haskell, where arithmetic, comparison, and conditionals are all represented by quoted terms, and hence identical to the host.

In theory, an EDSL also steals the normalisation rules of its host language, by using evaluation in the host to normalise terms of the target. In Section 5 we give several examples comparing our QDSL and EDSL versions of Feldspar. In the first of these, it is indeed the case that the EDSL achieves by evaluation of host terms what the QDSL achieves by normalisation of quoted terms. However, in other cases, the EDSL must perform normalisation of the deep embedding corresponding to what the QDSL achieves by normalisation of quoted terms.

> Try to give all of the information to help others to judge the value of your contribution; not just the information that leads to judgment in one particular direction or another.
>
> — Richard Feynman

The subformula property depends on normalisation, but normalisation may lead to an exponential blowup in the size of the normalised code. In particular, this occurs when there are nested conditional or case statements. We explain how the QDSL technique can offer the user control over where normalisation does and does not occur, while still maintaining the subformula property.

Some researchers contend that an essential property of an embedded DSL which generates target code is that every term that is type-correct should successfully generate code in the target language. Neither the P-LINK of Cheney et al. (2013) nor the QFeldspar of this paper satisfy this property. It is possible to ensure the property with additional preprocessing; we clarify the tradeoff between ease of implementation and ensuring safe compilation to target at compile-time rather than run-time.

> This is the short and the long of it. — Shakespeare

The contributions of this paper are:

- To introduce QDSLs as an approach to building DSLs based on quotation, normalisation of quoted terms, and the subformula property by presenting the design of a QDSL variant of Feldspar. (Section 2.)
- To compare QDSL and EDSL implementations of Feldspar, and show that they are of comparable length and offer comparable performace. (Section 3.)
- To explain the role of the subformula property in formulating DSLs, and to describe a normalisation algorithm suitable for call-by-value or call-by-need, which ensures the subformula property while not losing sharing of quoted terms. (Section 4.)
- To compare the QDSL variant of Feldspar with the deep and shallow embedding approach used in the EDSL variant of Feldspar. (Section 5.)

Section 6 describes related work, and Section 7 concludes.

2. Feldspar as a QDSL

Feldspar is an EDSL for writing signal-processing software, that generates code in C (Axelsson et al. 2010). We present a variant, QFeldspar, that follows the structure of the previous design closely, but using the methods of QDSL rather than EDSL. We make a detailed comparison of the QDSL and EDSL designs in Section 5.

2.1 The top level

In QFeldspar, our goal is to translate a quoted term to C code. The top-level function of QFeldspar has the type:

$$qdsl :: (Rep\ a, Rep\ b) \Rightarrow Qt\ (a \to b) \to C$$

Here $Qt\ \ a$ represents a Haskell term of type a, its *quoted* representation, and type C represents code in C. The top-level function expects a quoted term representing a function from type a to type b, and returns C code that represents this function (a `main` routine).

Not all types representable in Haskell are easily representable in the target language, C. For instance, we do not wish our target C code to manipulate higher-order functions. The argument type a and result type b of the main function must be representable, which is indicated by the type-class restrictions $Rep\ a$ and $Rep\ b$. Representable types include integers, floats, and pairs where the components are both representable.

$$\begin{array}{l}
\mathbf{instance}\ Rep\ Int \\
\mathbf{instance}\ Rep\ Float \\
\mathbf{instance}\ (Rep\ a, Rep\ b) \Rightarrow Rep\ (a, b)
\end{array}$$

It is easy to add triples and larger tuples.

2.2 An introductory example

Let's begin by considering the "hello world" of program generation, the power function, raising a float to an arbitrary integer. Since division by zero is undefined, we arbitrarily choose that raising zero to a negative power yields zero. Here is the power function represented using QDSL:

$$\begin{array}{l}
power :: Int \to Qt\ (Float \to Float) \\
power\ n = \\
\quad \mathbf{if}\ n < 0\ \mathbf{then} \\
\qquad [||\lambda x \to \mathbf{if}\ x == 0\ \mathbf{then}\ 0 \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ 1\ /\ (\$\$(power\ (-n))\ x)||] \\
\quad \mathbf{else\ if}\ n == 0\ \mathbf{then} \\
\qquad [||\lambda x \to 1||] \\
\quad \mathbf{else\ if}\ even\ n\ \mathbf{then} \\
\qquad [||\lambda x \to \$\$sqr\ (\$\$(power\ (n\ \mathrm{div}\ 2))\ x)||] \\
\quad \mathbf{else} \\
\qquad [||\lambda x \to x \times (\$\$(power\ (n-1))\ x)||] \\
sqr\ ::\ Qt\ (Float \to Float) \\
sqr = [||\lambda y \to y \times y||]
\end{array}$$

The typed quasi-quoting mechanism of Template Haskell is used to indicate which code executes at which time. Unquoted code executes at generation-time while quoted code executes at run-time. Quoting is indicated by $[||...||]$ and unquoting by $\$\$(\cdots)$.

Evaluating $power\ (-6)$ yields the following:

$$\begin{array}{l}
[||\lambda x \to \mathbf{if}\ x == 0\ \mathbf{then}\ 0\ \mathbf{else} \\
\quad 1\ /\ (\lambda x \to (\lambda y \to y \times y) \\
\qquad ((\lambda x \to (x \times ((\lambda x \to (\lambda y \to y \times y) \\
\qquad\quad ((\lambda x \to (x \times ((\lambda x \to 1)\ x)))\ x))\ x)))\ x))\ x||]
\end{array}$$

Normalising using the technique of Section 4, with variables renamed for readability, yields the following:

$$\begin{array}{l}
[||\lambda u \to \mathbf{if}\ u == 0\ \mathbf{then}\ 0\ \mathbf{else} \\
\qquad\qquad \mathbf{let}\ v = u \times 1\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ w = u \times (v \times v)\ \mathbf{in} \\
\qquad\qquad 1\ /\ (w \times w)||]
\end{array}$$

With the exception of the top-level term, all of the overhead of lambda abstraction and function application has been removed; we explain below why this is guaranteed by Gentzen's subformula property. From the normalised term it is easy to generate the final C code:

```
float main (float u) {
  if (u == 0) {
    return 0;
  } else {
    float v = u * 1;
    float w = u * (v * v);
    return 1 / (w * w);
  }
}
```

By default, we always generate a routine called `main`; it is easy to provide the name as an additional parameter if required.

Depending on your point of view, quotation in this form of QDSL is either desirable, because it makes manifest the staging, or undesirable because it is too noisy. QDSL enables us to "steal" the entire syntax of the host language for our DSL. The EDSL approach can use the same syntax for arithmetic operators, but must use a different syntax for equality tests and conditionals, as we will see in Section 5.

Within the quotation brackets there appear lambda abstractions and function applications, while our intention is to generate first-order code. How can the QFeldspar user be certain that such function applications do not render transformation to first-order code impossible or introduce additional runtime overhead? The answer is Gentzen's subformula property.

2.3 The subformula property

Gentzen's subformula property guarantees that any proof can be normalised so that the only formulas that appear within it are subformulas of one of the hypotheses or of the conclusion of the proof. Viewed through the lens of Propositions as Types (Howard 1980; Wadler 2015), also known as the Curry-Howard Isomorphism, Gentzen's subformula property guarantees that any term can be normalised so that the type of each of its subterms is a subtype of either the type of one of its free variables (corresponding to hypotheses) or the term itself (corresponding to the conclusion). Here the subtypes of a type are the type itself and the subtypes of its parts, where the parts of $a \to b$ are a and b, the parts of (a, b) are a and b, and the only part of $Arr\ \ a$ is a, and that types int and $float$ have no parts.

Further, it is easy to sharpen Gentzen's proof to guarantee a sharpened subformula property: any term can be normalised so that the type of each of its proper subterms is a proper subtype of either the type of one of its free variables (corresponding to hypotheses) or the term itself (corresponding to the conclusion). Here the proper subterms of a term are all subterms save for free variables and the term itself, and the proper subtypes of a type are all subtypes save for the type itself.

In the example of the previous subsection, the sharpened subformula property guarantees that after normalisation a term of type $float \to float$ will only have proper subterms of type $float$, which is indeed true for the normalised term.

(Careful readers will have noticed a small difficulty. One of the free variables of our quoted term is multiplication over floats. In Haskell, $m \times n$ abbreviates $(((\times)\ \ m)\ \ n)$, which has $((\times)\ \ m)$ as a subterm, and the type of $(\times)$ is $(float \to (float \to float))$, which has $(float \to float)$ as a subtype. We alleviate the difficulty by a standard trick: each free variable is assigned an arity and must always be fully applied. Taking $(\times)$ to have arity 2 requires we always write $m \times n$ in our code. Then we may, as natural, regard m and n as the only subterms of $m \times n$, and float as the only subtype of the type of $(\times)$. Details appear in Section 4.)

2.4 Maybe

In the previous code, we arbitrarily chose that raising zero to a negative power yields zero. Say that we wish to exploit the Maybe type to refactor the code, separating identifying the exceptional case (negative exponent of zero) from choosing a value for this case (zero). We decompose power into two functions power' and power'', where the first returns $\mathit{Nothing}$ in the exceptional case, and the second maps $\mathit{Nothing}$ to a suitable default value.

The Maybe type is a part of the standard prelude.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \\
\mathit{maybe} :: b \to (a \to b) \to \mathit{Maybe}\ a \to b \\
\mathit{return} :: a \to \mathit{Maybe}\ a \\
(\gg\!=)\ \ :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b
\end{array}
$$

Here is the refactored code.

$$
\begin{array}{l}
\mathit{power}' :: \mathit{Int} \to \mathit{Qt}\ (\mathit{Float} \to \mathit{Maybe}\ \mathit{Float}) \\
\mathit{power}'\ n = \\
\quad \mathbf{if}\ n < 0\ \mathbf{then} \\
\qquad [||\lambda x \to \mathbf{if}\ x == 0\ \mathbf{then}\ \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else\ do}\ y \leftarrow \$\$(\mathit{power}'\ (-n))\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (1\ /\ y)||] \\
\quad \mathbf{else\ if}\ n == 0\ \mathbf{then} \\
\qquad [||\lambda x \to \mathit{return}\ 1||] \\
\quad \mathbf{else\ if}\ \mathit{even}\ n\ \mathbf{then} \\
\qquad [||\lambda x \to \mathbf{do}\ y \leftarrow \$\$(\mathit{power}'\ (n\ \mathrm{div}\ 2))\ x \\
\qquad\qquad\qquad\qquad \mathit{return}\ (\$\$\mathit{sqr}\ y)||] \\
\quad \mathbf{else} \\
\qquad [||\lambda x \to \mathbf{do}\ y \leftarrow \$\$(\mathit{power}'\ (n-1))\ x \\
\qquad\qquad\qquad\qquad \mathit{return}\ (x \times y)||] \\
\mathit{power}'' :: \mathit{Int} \to \mathit{Qt}\ (\mathit{Float} \to \mathit{Float}) \\
\mathit{power}''\ n = \\
\quad [||\lambda x \to \mathit{maybe}\ 0\ (\lambda y \to y)\ (\$\$(\mathit{power}'\ n)\ x)||]
\end{array}
$$

Here sqr is as before. Evaluation and normalisation of $\mathit{power}\ (-6)$ and $\mathit{power}''\ (-6)$ yield identical terms (up to renaming), and hence applying qdsl to these yields identical C code.

The subformula property is key: because the final type of the result does not involve Maybe it is certain that normalisation will remove all its occurrences. Occurrences of **do** notation are expanded to applications of $(\gg\!=)$, as usual. Rather thank taking return, $(\gg\!=)$, and maybe as free variables (whose types have subtypes involving Maybe), we treat them as known definitions to be eliminated by the normaliser. The Maybe type is essentially a sum type, and normalisation for these is as described in Section 4.

We have chosen not to make Maybe a representable type, which prohibits its use as argument or result of the top-level function passed to qdsl. An alternative choice is possible, as we will see when we consider arrays, in Section 2.6 below.

2.5 While

Code that is intended to compile to a `while` loop in C is indicated in QFeldspar by application of while.

$$\mathit{while} :: (\mathit{Rep}\ s) \Rightarrow \mathit{Qt}\ ((s \to \mathit{Bool}) \to (s \to s) \to s \to s)$$

Rather than using side-effects, while takes three arguments: a predicate over the current state, of type $s \to \mathit{Bool}$; a function from current state to new state, of type $s \to s$; and an initial state of type s; and it returns a final state of type s. Since we intend to compile the while loop to C, the type of the state is constrained to representable types.

We can define a for loop in terms of a while loop.

$$
\begin{array}{l}
\mathit{for} :: (\mathit{Rep}\ s) \Rightarrow \mathit{Qt}\ (\mathit{Int} \to s \to (\mathit{Int} \to s \to s) \to s) \\
\mathit{for} = [||\lambda n\ s\_0\ b \to \mathit{snd}\ (\mathit{while}\ (\lambda(i,s) \to i < n) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda(i,s) \to (i+1, b\ i\ s)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (0, s\_0))||]
\end{array}
$$

The state of the while loop is a pair consisting of a counter and the state of the for loop. The body b of the for loop is a function that expects both the counter and the state of the for loop. The counter is discarded when the loop is complete, and the final state of the for loop returned.

As an example, we can define Fibonacci using a for loop.

$$
\begin{array}{l}
\mathit{fib} :: \mathit{Qt}\ (\mathit{Int} \to \mathit{Int}) \\
\mathit{fib} = [||\lambda n \to \$\$\mathit{for}\ n\ (\lambda(a,b) \to (b, a+b))\ (0,1)\ ||]
\end{array}
$$

Again, the subformula property plays a key role. As explained in Section 2.3, primitives of the language to be compiled, such as $(\times)$ and while, are treated as free variables of a given arity. As will be explained in Section 4, we can ensure that after normalisation every occurence of while has the form

$$\mathit{while}\ (\lambda s \to \cdots)\ (\lambda s \to \cdots)\ (\cdots)$$

where the first ellipses has type Bool, and both occurrences of s and the second and third ellipses all have the same type.

Unsurprisingly, and in accord with the subformula property, for each occurrence of while in the normalised code will contain subterms with the type of its state. The restriction of state to representable types increases the utility of the subformula property. For instance, since we have chosen that Maybe is not a representable type, we can ensure that any top-level function without Maybe in its type will normalise to code not containing Maybe in the type of any subterm. An alternative choice is possible, as we will see in the next section.

The subformula property depends on normalisation of terms, but complete normalisation is not always possible or desirable. The extent of normalisation may be controlled by introducing uninterpreted constants, such as while. In a context with recursion, we take $\mathit{fix} :: (a \to a) \to a$ as an uninterpreted constant. In a context where we wish to avoid unfolding a reduction $L\ M$, we take $\mathit{id} :: a \to a$ as an uninterpreted constant, and replace $L\ M$ by $\mathit{id}\ L\ M$.

2.6 Arrays

A key feature of Feldspar is its distinction between two types of arrays, manifest arrays Arr which may appear at run-time, and "pull arrays" Vec which are eliminated by fusion at generation-time. Again, we exploit the subformula property to ensure no subterms of type Vec remain in the final program.

The type Arr of manifest arrays is simply Haskell's array type, specialised to arrays with integer indices and zero-based indexing. The type Vec of pull arrays is defined in terms of existing types, as a pair consisting of the length of the array and a function that given an index returns the array element at that index.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Arr}\ a = \mathit{Array}\ \mathit{Int}\ a \\
\mathbf{data}\ \mathit{Vec}\ a = \mathit{Vec}\ \mathit{Int}\ (\mathit{Int} \to a)
\end{array}
$$

Values of type Arr are representable, assuming that the element type is representable, while values of type Vec are not representable.

$$\mathbf{instance}\ (\mathit{Rep}\ a) \Rightarrow \mathit{Rep}\ (\mathit{Arr}\ a)$$

For arrays, we assume the following primitive operations.

$$
\begin{aligned}
&\mathit{makeArr} :: (\mathit{Rep}\ a) \Rightarrow \mathit{Int} \to (\mathit{Int} \to a) \to \mathit{Arr}\ a \\
&\mathit{lenArr}\ \ :: (\mathit{Rep}\ a) \Rightarrow \mathit{Arr}\ a \to \mathit{Int} \\
&\mathit{ixArr}\ \ \ :: (\mathit{Rep}\ a) \Rightarrow \mathit{Arr}\ a \to \mathit{Int} \to a
\end{aligned}
$$

The first populates a manifest array of the given size using the given indexing function, the second returns the length of the array, and the third returns the array element at the given index.

We define functions to convert between the two representations in the obvious way.

$$
\begin{aligned}
&\mathit{toVec}\ \ :: \mathit{Qt}\ (\mathit{Arr}\ a \to \mathit{Vec}\ a) \\
&\mathit{toVec}\ \ = [||\lambda a \to \mathit{Vec}\ (\mathit{lenArr}\ a)\ (\lambda i \to \mathit{ixArr}\ a\ i)||] \\
&\mathit{fromVec} :: \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Arr}\ a) \\
&\mathit{fromVec} = [||\lambda(\mathit{Vec}\ n\ g) \to \mathit{makeArr}\ n\ (\lambda x \to g\ x)||]
\end{aligned}
$$

It is straightforward to define operations on vectors, including combining corresponding elements of two vectors, summing the elements of a vector, dot product of two vectors, and norm of a vector.

$$
\begin{aligned}
&\mathit{zipVec}\ \ :: \mathit{Qt}\ ((a \to b \to c) \to \mathit{Vec}\ a \to \mathit{Vec}\ b \to \mathit{Vec}\ c) \\
&\mathit{zipVec}\ \ = [||\lambda f\ (\mathit{Vec}\ m\ g)\ (\mathit{Vec}\ n\ h) \to \\
&\qquad\qquad \mathit{Vec}\ (m\ \text{`}\mathit{min}\text{`}\ n)\ (\lambda i \to f\ (g\ i)\ (h\ i))||] \\
&\mathit{sumVec}\ :: (\mathit{Rep}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to a) \\
&\mathit{sumVec}\ = [||\lambda(\mathit{Vec}\ n\ g) \to \$\$\mathit{for}\ n\ 0\ (\lambda i\ x \to x + g\ i)||] \\
&\mathit{dotVec}\ \ :: (\mathit{Rep}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Vec}\ a \to a) \\
&\mathit{dotVec}\ \ = [||\lambda u\ v \to \$\$\mathit{sumVec}\ (\$\$\mathit{zipVec}\ (\times)\ u\ v)||] \\
&\mathit{normVec} :: \mathit{Qt}\ (\mathit{Vec}\ \mathit{Float} \to \mathit{Float}) \\
&\mathit{normVec} = [||\lambda v \to \mathit{sqrt}\ (\$\$\mathit{scalarProd}\ v\ v)||]
\end{aligned}
$$

The second of these uses the for loop defined in Section 2.5, the third is defined using the first two, and the fourth is defined using the third. Recall that our sharpened subformula property required that $(\times)$ be fully applied, so before normalisation $(\times)$ is expanded to $\lambda x\ y \to x \times y$.

Our final function cannot accept Vec as input, since the Vec type is not representable, but it can accept Arr as input. Invoking $\mathit{qdsl}\ (\mathit{normVec} \circ \mathit{toVec})$ produces the following C code.

```
float main(float[] a) {
  float x = 0;
  int i = 0;
  while (i < lenArr a) {
    x = x + a[i] * a[i];
    i = i+1;
  }
  return sqrt(x);
}
```

[TODO: Shayan to check that above is correct.]

Types and the subformula property help us to guarantee fusion. The subformula property guarantees that all occurrences of Vec must be eliminated, while occurrences of Arr will remain. There are some situations where fusion is not beneficial, notably when an intermediate vector is accessed many times fusion will cause the elements to be recomputed. An alternative is to materialise the vector as an array with the following function.

$$
\begin{aligned}
&\mathit{memorise} :: \mathit{Syn}\ a \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Vec}\ a) \\
&\mathit{memorise} = [||\mathit{toVec} \circ \mathit{fromVec}||]
\end{aligned}
$$

For example, if

$$\mathit{blur} :: \mathit{Rep}\ a \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Vec}\ a)$$

averages adjacent elements of a vector, then one may choose to compute either

$$[|\$\$\mathit{blur} \circ \$\$\mathit{blur}\ |] \quad \text{or} \quad [|\$\$\mathit{blur} \circ \$\$\mathit{memorise} \circ \$\$\mathit{blur}\ |]$$

with different trade-offs between recomputation and memory usage. Strong guarantees for fusion in combination with $\mathit{memorize}$ gives the programmer a simple interface which provides powerful optimisation combined with fine control over memory usage.

[TODO: The subformula property guarantees that all occurrences of Vec will vanish from the final program. The same guarantee applies regardless of whether Vec is defined as "pull arrays" or "push arrays". Does this mean the difference is irrelevant for QFeldspar?]

3. Implementation

The original Feldspar generates values of an algebraic type (called $\mathit{Dp}\ \ a$ in Section 5), with constructs that represent while and manifest arrays similar to those above. A backend then compiles values of type $\mathit{Exp}\ a$ to C code. QFeldspar provides a transformer from $\mathit{Qt}\ a$ to $\mathit{Exp}\ a$, and shares the Feldspar backend.

The transformer from Qt to Exp performs the following steps.

- In any context where a constant c is not fully applied, it replaces c with $\lambda\overline{x}.$. It replaces identifiers connected to the type Maybe, such as return, $(\gg\!=)$, and maybe, by their definitions.
- It normalises the term to ensure the subformula property, using the rules of Section 4. The normaliser does not support all Haskell data types, but does support tuples, and the types Maybe and Vec.
- It traverses the term, converting Qt to Dp. It checks that only permitted primitives appear in Qt, and translates these to their corresponding representation in Dp. Permitted primitives include: $(==)$, $(<)$, $(+)$, $(\times)$, and similar, plus while, $\mathit{makeArr}$, lenArr, and ixArr.

An unfortunate feature of typed quasiquotation in GHC is that the implementation discards all type information when creating the representation of a term. Type $\mathit{Qt}\ \ a$ is equivalent to $\mathit{TH.Q}\ \ (\mathit{TH.TExp}\ \ a)$, where TH denotes the library for Template Haskell, $\mathit{TH.Q}$ is the quotation monad of Template Haskell (used to look up identifiers and generate fresh names), and $\mathit{TH.TExp}\ a$ is the parse tree for a quoted expression returning a value of type a. Type $\mathit{TH.TExp}\ \ a$ is just a wrapper for $\mathit{TH.Exp}$, the (untyped) parse tree of an expression in Template Haskell, where a is a phantom type variable. Hence, the translator from $\mathit{Qt}\ \ a$ to $\mathit{Dp}\ \ a$ is forced to re-infer all the type information for the subterms of the term of type $\mathit{Qt}\ \ a$. This is why we translate the Maybe monad as a special case, rather than supporting overloading for monad operations. [TODO: say something about how overloading for arithmetic is handled.]

We measured the behaviour of five benchmark programs.

IPGray	Image Processing (Grayscale)
IPBW	Image Processing (Black and White)
FFT	Fast Fourier Transform
CRC	Cyclic Redundancy Check
Windowing	Average array in a sliding window

Figure 1 lists the results. Columns Hc and Hr list compile-time and run-time in Haskell, and Cc and Cr list compile-time and run-time in C. Runs for CDSL are shown both with and without common subexpression elimination (CSE), which is supported by a simple form of observable sharing. QDSL does not require CSE, since the normalisation algorithm preserves sharing. One benchmark, FFT, exhausts memory without CSE. All benchmarks produce essentially the same C for both QDSL and CDSL, which run in essentially the same time. The one exception is FFT, where Feldspar appears to introduce spurious conversions that increase the runtime.

Measurements were done on a PC with a quad-core Intel i7-2640M CPU running at 2.80 GHz and 3.7 GiB of RAM, with GHC

	QDSL				EDSL without CSE				EDSL with CSE			
	Hc	Hr	Cc	Cr	Hc	Hr	Cc	Cr	Hc	Hr	Cc	Cr
IPGray	11.40	0.00	0.08	0.42	6.22	0.00	0.08	0.45	6.55	0.00	0.07	0.45
IPBW	11.26	0.00	0.08	0.20	6.22	0.00	0.07	0.20	6.55	0.00	0.07	0.20
FFT	11.36	0.24	0.08	5.08	6.15	0.47	0.13	—	6.65	0.16	0.09	6.15
CRC	11.29	0.01	0.08	0.14	6.13	0.00	0.08	0.15	6.49	0.00	0.08	0.15
Windowing	11.19	0.01	0.08	0.31	6.21	0.01	0.08	0.33	6.51	0.00	0.08	0.33

Figure 1. Comparison of QDSL and EDSL MiniFeldspar (all times in seconds)

Version 7.8.3 and GCC version 4.8.2, running on Ubuntu 14.04 (64-bit).

4. The subformula property

This section introduces a collection of reduction rules for normalising terms that enforces the subformula property while ensuring sharing is preserved. The rules adapt to both call-by-need and call-by-value.

We work with simple types. The only polymorphism in our examples corresponds to instantiating constants (such as $while$) at different types.

Types, terms, and values are presented in Figure 2. Let A, B, C range over types, including base types (ι), functions ($A \to B$), products ($A \times B$), and sums ($A + B$). Let L, M, N range over terms, and x, y, z range over variables. Let c range over constants, which are fully applied according to their arity, as discussed below. As usual, terms are taken as equivalent up to renaming of bound variables. Write $FV(M)$ for the set of free variables of M, and $N[x := M]$ for capture-avoiding substitution of M for x in N. Let V, W range over values, and P range over terms that are not values.

Let Γ range over type environments, which are sets of pairs of variables with types $x : A$. Write $\Gamma \vdash M : A$ to indicate that term M has type A under type environment Γ. Typing rules are standard.

Reduction rules for normalisation are presented in Figure 3. The rules are confluent, so order of application is irrelevant to the final answer, but we break them into three phases to ease the proof of strong normalisation. It is easy to confirm that all of the reduction rules preserve sharing and preserve order of evaluation.

Write $M \mapsto_i N$ to indicate that M reduces to N in phase i. Only Phases 1 and 2 are required to normalise terms, but if the semantics is call-by-need then Phase 3 may also be applied. Let F and G range over two different forms of evaluation frame used in Phases 1 and 2 respectively. We write $FV(F)$ for the set of free variables of F, and similarly for G. The reduction relation is closed under compatible closure.

The normalisation procedure consists of exhaustively applying the reductions of Phase 1 until no more apply, then similarly for Phase 2, and finally for Phase 3. Phase 1 performs let-insertion, naming subterms that are not values, along the lines of a translation to A-normal form (Flanagan et al. 1993) or reductions (let.1) and (let.2) in Moggi's metalanguage for monads (Moggi 1991). Phase 2 performs standard β and commuting reductions, and is the only phase that is crucial for obtaining normal forms that satisfy the subformula property. Phase 3 "garbage collects" unused terms as in the call-by-need lambda calculus (Maraist et al. 1998). Phase 3 should be omitted if the intended semantics of the target language is call-by-value rather than call-by-need.

Every term has a normal form.

PROPOSITION 4.1 (Strong normalisation). *Each of the reduction relations $\mapsto_i$ is confluent and strongly normalising: all $\mapsto_i$ reduction sequences on well-typed terms are finite.*

The only non-trivial proof is for $\mapsto_2$, which can be proved via a standard reducibility argument (see, for example, (Lindley 2007)). If the target language includes general recursion, normalisation should treat the fixpoint operator as an uninterpreted constant.

The *subformulas* of a type are the type itself and its components. For instance, the subformulas of $A \to B$ are itself and the subformulas of A and B. The *proper subformulas* of a type are all its subformulas other than the type itself.

The *subterms* of term are the term itself and its components. For instance, the subterms of $\lambda x.$ are itself and the subterms of N and the subterms of $L\ M$ are itself and the subterms of L and M.

Constants are always fully applied; they are introduced as a separate consturct to avoid consideration of irrelevant subformulas and subterms. The type of a constant c of arity k is written

$$c : A_1 \to \cdots A_k \to B$$

and its subtypes are itself and $A_1, \ldots, A_k$, and B (but not $A_i \to \ldots \to A_k \to B$ for $i > 1$). An application of a constant c of arity k is written

$$c\ M_1\ \cdots\ M_k$$

and its subterms are itself and $M_1, \ldots, M_k$ (but not $c\ M_1\ \cdots\ M_j$ for $j < k$). Free variables are equivalent to constants of arity zero.

Terms in normal form satisfy the subformula property.

PROPOSITION 4.2 (Subformula property). *If $\Gamma \vdash M : A$ and M is in normal form, then every subterm of M has a type that is either a subformula of A, a subformula of a type in Γ, or a subformula of the type of a constant in M.*

The proof follows the lines of Prawitz (1965). The differences are that we have introduced fully applied constants (to enable the sharpened subformula property, below), and that our reduction rules introduce **let**, in order to ensure sharing is preserved.

Normalisation may lead to an exponential increase in the size of a term, for instance when there are nested **case** expressions. This was not a problem for the examples we considered in Section 3, but may be a problem in some contexts. Normalisation may be controlled by introduction of uninterpreted constants (see next paragraph), but further work is needed to understand the contexts in which complete normalisation is desirable and the contexts in which it is problematic.

As we noted in Section 2.5, the subformula property depends on complete normalisation of terms, but complete normalisation is not always possible possible or desirable. The extent of normalisation may be controlled by introducing uninterpreted constants, such as $while$. In a context with general recursion, we take $fix :: (a \to a) \to a$ as an uninterpreted constant. In a context where we wish to avoid unfolding a reduction $L\ M$, we take $id :: a \to a$ as an uninterpreted constant, and replace $L\ M$ by $id\ L\ M$.

Examination of the proof in Prawitz (1965) shows that in fact normalisation achieves a sharper property.

PROPOSITION 4.3 (Sharpened subformula property). *If $\Gamma \vdash M : A$ and M is in normal form, then every proper subterm of M that*

$$
\begin{array}{llll}
\text{Types} & A, B, C & ::= & \iota \mid A \to B \mid A \times B \mid A + B \\
\text{Terms} & L, M, N & ::= & x \mid c\,\overline{M} \mid \lambda x.N \mid L\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid (M, N) \mid \mathbf{fst}\ L \mid \mathbf{snd}\ L \mid \\
 & & & \mathbf{inl}\ M \mid \mathbf{inr}\ N \mid \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{inl}\ x.\,M;\ \mathbf{inr}\ y.\,N\} \\
\text{Values} & V, W & ::= & x \mid \lambda x.N \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ W
\end{array}
$$

Figure 2. Types, Terms, and Values

Phase 1 (let-insertion)

$$F \quad ::= \quad c\,(\overline{V}, [\,], \overline{N}) \mid [\,]\ M \mid V\ [\,] \mid ([\,], M) \mid (V, [\,]) \mid \mathbf{fst}\ [\,] \mid \mathbf{snd}\ [\,] \mid \mathbf{inl}\ [\,] \mid \mathbf{inr}\ [\,] \mid \mathbf{case}\ [\,]\ \mathbf{of}\ \{\mathbf{inl}\ x.\,M;\ \mathbf{inr}\ y.\,N\}$$

$$(\mathit{let}) \quad F[P] \quad \mapsto_1 \quad \mathbf{let}\ x = P\ \mathbf{in}\ F[x], \quad x \text{ fresh}$$

Phase 2 (symbolic evaluation)

$$G ::= [\,]\ V \mid \mathbf{let}\ x = [\,]\ \mathbf{in}\ N$$

$$
\begin{array}{llcll}
(\kappa.\mathit{let}) & G[\mathbf{let}\ x = P\ \mathbf{in}\ N] & \mapsto_2 & \mathbf{let}\ x = P\ \mathbf{in}\ G[N], & x \notin FV(G) \\
(\kappa.\mathit{case}) & G[\mathbf{case}\ z\ \mathbf{of}\ \{\mathbf{inl}\ x.\,M;\ \mathbf{inr}\ y.\,N\}] & \mapsto_2 & \mathbf{case}\ z\ \mathbf{of}\ \{\mathbf{inl}\ x.\,G[M];\ \mathbf{inr}\ y.\,G[N]\}, & x, y \notin FV(G) \\
(\beta.\to) & (\lambda x.N)\ V & \mapsto_2 & N[x := V] & \\
(\beta.\times_1) & \mathbf{fst}\ (V, W) & \mapsto_2 & V & \\
(\beta.\times_2) & \mathbf{snd}\ (V, W) & \mapsto_2 & W & \\
(\beta.+_1) & \mathbf{case}\ (\mathbf{inl}\ V)\ \mathbf{of}\ \{\mathbf{inl}\ x.\,M;\ \mathbf{inr}\ y.\,N\} & \mapsto_2 & M[x := V] & \\
(\beta.+_2) & \mathbf{case}\ (\mathbf{inr}\ W)\ \mathbf{of}\ \{\mathbf{inl}\ x.\,M;\ \mathbf{inr}\ y.\,N\} & \mapsto_2 & N[y := W] & \\
(\beta.\mathit{let}) & \mathbf{let}\ x = V\ \mathbf{in}\ N & \mapsto_2 & N[x := V] &
\end{array}
$$

Phase 3 (garbage collection)

$$(\mathit{need}) \quad \mathbf{let}\ x = P\ \mathbf{in}\ N \quad \mapsto_3 \quad N, \quad x \notin FV(N)$$

Figure 3. Normalisation rules

is not a free variable or a subterm of a constant application has a type that is a proper subformula of A or a proper subformula of a type in Γ.

The sharpened subformula property says nothing about the type of subterms of constant applications, but this is immediately apparent by recursive application of the sharpended subformula property. Given a subterm that is a constant application $c\ \overline{M}$, where c has type $\overline{A} \to B$, then the subterm itself has type B, each subterm M_i has type A_i, and every proper subterm of M_i that is not a variable or a subterm of a constant application has a type that is a proper subformula of A_i or a proper subformula of the type of one of its free variables.

In Section 2, an important property is that every top-level term passed to qdsl is suitable for translation to C after normalisation. Here we are interested in C as a *first-order* language. The exact property required is somewhat subtle. One might at first guess the required property is that every subterm is representable, in the sense introduced in Section 2.1, but this is not quite right. The top-level term is a function from a representable type to a representable type. And the constant while expects subterms of type $s \to \mathit{Bool}$ and $s \to s$, where the state s is representable. Fortunately, the exact property required is not hard to formulate in a general way, and is easy to ensure by applying the sharpened subformula property.

We introduce a variant of the usual notion of *rank* of a type, with respect to a notion of representability. A term of type $A \to B$ has rank $\min(m+1, n)$ where m is the rank of A and n is the rank of B, while a term of representable type has rank 0.

The property we need to ensure ease of translation to C (or any other first-order language) is as follows.

PROPOSITION 4.4 (Rank and representability). *Consider a term M of rank 1, where every free variable of M has rank 0 and every constant in M has rank at most 2. Then M normalises to a form where every subterm either is of representable type or is of the form $\lambda\overline{x}.$ where each of the bound variables x_i and the body N has representable type.*

The property follows immediately by observing that any term L with type of rank 1 can be rewritten in the form $\lambda\overline{x}.$ where each bound variable and the body has representable type, and then normalising and applying the sharpened subformula property.

5. Feldspar as an EDSL

This section reviews the combination of deep and shallow embeddings required to implement Feldspar as an EDSL, and considers the tradeoffs between the QDSL and EDSL approaches.

5.1 The top level

The top-level function of EFeldspar has the type:

$$\mathit{qdsl} :: (\mathit{Rep}\ a, \mathit{Rep}\ b) \Rightarrow (\mathit{Dp}\ a \to \mathit{Dp}\ b) \to C$$

Here $\mathit{Dp}\ a$ is the deep representation of a term of type a. The deep representation is described in detail in Section 5.4 below, and is chosen to be easy to translate to C. As before, type C represents code in C, and type class Rep restricts to representable types.

5.2 An introductory example

Here is the power function of Section 2.2, now represented using EDSL:

$$
\begin{array}{l}
\mathit{power} :: \mathit{Int} \to \mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float} \\
\mathit{power}\ n\ x = \\
\quad \mathbf{if}\ n < 0\ \mathbf{then} \\
\quad\quad x \mathbin{.{==}.} 0\ ?\ (0, 1\ /\ \mathit{power}\ ({-n})\ x)
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{else if } n == 0 \textbf{ then} \\
\qquad 1 \\
\quad \textbf{else if } \mathit{even}\ n \textbf{ then} \\
\qquad \mathit{sqr}\ (\mathit{power}\ (n \operatorname{div} 2)\ x) \\
\quad \textbf{else} \\
\qquad x \times \mathit{power}\ (n-1)\ x \\
\mathit{sqr} :: \mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float} \\
\mathit{sqr}\ y = y \times y
\end{array}
$$

Type $Q\ (\mathit{Float} \to \mathit{Float})$ in the QDSL variant becomes $\mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float}$ in the EDSL variant, meaning that $\mathit{power}\ n$ accepts a representation of the argument and returns a representation of that argument raised to the n'th power.

In EDSL, no quotation is required, and the code looks almost—but not quite!—like an unstaged version of power, but with different types. Clever encoding tricks, explained later, permit declarations, function calls, arithmetic operations, and numbers to appear the same whether they are to be executed at generation-time or run-time. However, as explained later, comparison and conditionals appear differently depending on whether they are to be executed at generation-time or run-time, using $M == N$ and $\textbf{if } L \textbf{ then } M \textbf{ else } N$ for the former but $M \mathbin{.==.} N$ and $L ? (M, N)$ for the latter.

Evaluating $\mathit{power}\ (-6)$ yields the following:

$$
\begin{array}{l}
(\lambda u \to (u \mathbin{.==.} 0)\ ?\ (0, \\
\quad 1\ /\ ((u \times ((u \times 1) \times (u \times 1))) \times \\
\qquad (u \times ((u \times 1) \times (u \times 1))))))
\end{array}
$$

Applying common-subexpression elimination, or using a technique such as observable sharing, permits recovering the sharing structure.

$$
\begin{array}{l|l}
v & (u \times 1) \\
w & u \times (v \times v) \\
\mathit{main} & (u \mathbin{.==.} 0)\ ?\ (0, 1\ /\ (w \times w))
\end{array}
$$

From the above, it is easy to generate the final C code, which is identical to that in Section 2.2.

Here are some points of comparison between the two approaches.

- A function $a \to b$ is embedded in EDSL as $\mathit{Dp}\ a \to \mathit{Dp}\ b$, a function between representations, and in QDSL as $\mathit{Qt}\ (a \to b)$, a representation of a function.
- EDSL requires some term forms, such as comparison and conditionals, to differ between the host and embedded languages. In contrast, QDSL enables the host and embedded languages to appear identical.
- EDSL permits the host and embedded languages to intermingle seamlessly. In contrast, QDSL requires syntax to separate quoted and unquoted terms, which (depending on your point of view) may be considered as an unnessary distraction or as drawing a useful distinction between generation-time and run-time. If one takes the former view, the type-based approach to quotation found in C# and Scala might be preferred.
- EDSL typically develops custom shallow and deep embeddings for each application, although these may follow a fairly standard pattern (as we see below). In contrast, QDSL may share the same representation for quoted terms across a range of applications; the quoted language is the host language, and does not vary with the specific domain.
- EDSL loses sharing, which must later be recovered by either common subexpression elimination or applying a technique such as observable sharing. In contrast, QDSL preserves sharing throughout.
- EDSL yields the term in normalised form in this case, though there are other situations where a normaliser is required (see Section 5.3.) In contrast, QDSL yields an unwieldy term that requires normalisation. However, just as a single representation of QDSL terms suffices across many applications, so does a single normaliser—it can be built once and reused many times.
- Once the deep embedding or the normalised quoted term is produced, generating the domain-specific code is similar for both approaches.

5.3 Maybe

In Section 2.4, we exploited the Maybe type to refactor the code.

In EDSL, we must use a new type, where Maybe, $\mathit{Nothing}$, Just, and maybe become Opt, none, some, and option, and return and $(\gg=)$ are similar to before.

$$
\begin{array}{l}
\textbf{type } \mathit{Opt}\ a \\
\mathit{none}\ \ :: \mathit{Undef}\ a \Rightarrow \mathit{Opt}\ a \\
\mathit{some}\ \ :: a \to \mathit{Opt}\ a \\
\mathit{return} :: a \to \mathit{Opt}\ a \\
(\gg=)\ \ :: \mathit{Opt}\ a \to (a \to \mathit{Opt}\ b) \to \mathit{Opt}\ b \\
\mathit{option} :: (\mathit{Undef}\ a, \mathit{Undef}\ b) \Rightarrow \\
\qquad\qquad b \to (a \to b) \to \mathit{Opt}\ a \to b
\end{array}
$$

Type class Undef is explained in Section 5.7, and details of type Opt are given in Section 5.8.

In order to be easily represented in C, type $\mathit{Opt}\ a$ is represented as a pair consisting of a boolean and the representation of the type a. For none, the boolean is false and a default value of type a is provided. For some, the boolean is true.

Here is the refactored code.

$$
\begin{array}{l}
\mathit{power}' :: \mathit{Int} \to \mathit{Dp}\ \mathit{Float} \to \mathit{Opt}\ (\mathit{Dp}\ \mathit{Float}) \\
\mathit{power}'\ n\ x = \\
\quad \textbf{if } n < 0 \textbf{ then} \\
\qquad (x \mathbin{.==.} 0)\ ?\ (\mathit{none}, \\
\qquad\quad \textbf{do } y \leftarrow \mathit{power}'\ (-n)\ x \\
\qquad\qquad \mathit{return}\ (1\ /\ y)) \\
\quad \textbf{else if } n == 0 \textbf{ then} \\
\qquad \mathit{return}\ 1 \\
\quad \textbf{else if } \mathit{even}\ n \textbf{ then} \\
\qquad \textbf{do } y \leftarrow \mathit{power}'\ (n \operatorname{div} 2)\ x \\
\qquad\quad \mathit{return}\ (\mathit{sqr}\ y) \\
\quad \textbf{else} \\
\qquad \textbf{do } y \leftarrow \mathit{power}'\ (n-1)\ x \\
\qquad\quad \mathit{return}\ (x \times y) \\
\mathit{power}'' :: \mathit{Int} \to \mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float} \\
\mathit{power}''\ n\ x = \mathit{option}\ 0\ (\lambda y \to y)\ (\mathit{power}'\ n\ x)
\end{array}
$$

Here sqr is as before.

The term of type $\mathit{Dp}\ \mathit{Float}$ generated by evaluating $\mathit{power}\ (-6)\ x$ is large and unscrutable:

$$
\begin{array}{l}
(((\mathit{fst}\ ((x == 0.0)\ ?\ (((\mathit{False}\ ?\ (\mathit{True}, \mathit{False})), (\mathit{False}\ ? \\
(\mathit{undef}, \mathit{undef}))), (\mathit{True}, (1.0\ /\ ((x \times ((x \times 1.0) \times (x \times \\
1.0))) \times (x \times ((x \times 1.0) \times (x \times 1.0)))))))))\ ?\ (\mathit{True}, \\
\mathit{False}))\ ?\ (((\mathit{fst}\ ((x == 0.0)\ ?\ (((\mathit{False}\ ?\ (\mathit{True}, \mathit{False})), \\
(\mathit{False}\ ?\ (\mathit{undef}, \mathit{undef}))), (\mathit{True}, (1.0\ /\ ((x \times ((x \times 1.0) \times \\
(x \times 1.0))) \times (x \times ((x \times 1.0) \times (x \times 1.0)))))))))\ ?\ ((\mathit{snd} \\
((x == 0.0)\ ?\ (((\mathit{False}\ ?\ (\mathit{True}, \mathit{False})), (\mathit{False}\ ?\ (\mathit{undef}, \\
\mathit{undef}))), (\mathit{True}, (1.0\ /\ ((x \times ((x \times 1.0) \times (x \times 1.0))) \times \\
(x \times ((x \times 1.0) \times (x \times 1.0)))))))))), \mathit{undef})), 0.0))
\end{array}
$$

Before, evaluating power yielded a term essentially in normal form, save for the need to use common subexpression elimination or observable sharing to recover shared structure. However, this is not the case here. Rewrite rules including the following need to be repeatedly applied.

$$
\begin{array}{rcl}
\mathit{fst}\ (M,N) & \rightsquigarrow & M \\
\mathit{snd}\ (M,N) & \rightsquigarrow & N \\
\mathit{fst}\ (L\ ?\ (M,N)) & \rightsquigarrow & L\ ?\ (\mathit{fst}\ M, \mathit{fst}\ N) \\
\mathit{snd}\ (L\ ?\ (M,N)) & \rightsquigarrow & L\ ?\ (\mathit{snd}\ M, \mathit{snd}\ N) \\
\mathit{True}\ ?\ (M,N) & \rightsquigarrow & M \\
\mathit{False}\ ?\ (M,N) & \rightsquigarrow & N \\
(L\ ?\ (M,N))\ ?\ (P,Q) & \rightsquigarrow & L\ ?\ (M\ ?\ (P,Q))\ ?\ (N\ ?\ (P,Q)) \\
L\ ?\ (M,N) & \rightsquigarrow & L\ ?\ (M[L := \mathit{True}], N[L := \mathit{False}])
\end{array}
$$

Here L,M,N,P,Q range over Dp terms, and $M\ [L := P]$ stands for M with each occurence of L replaced by P. After applying these rules, common subexpression elimination yields the same structure as in the previous subsection, from which the same C code is generated.

Hence, an advantages of the EDSL approach—that it generates terms essentially in normal form—turns out to be restricted to a limited set of types, including functions and products, but excluding sums. If one wishes to deal with sum types, separate normalisation is required. This is one reason why we do not consider normalisation as required by QDSL to be particularly onerous.

Here are further points of comparison between the two approaches.

- Both CDSL and QDSL can exploit notational conveniences in the host language. The example here exploits Haskell **do** notation; the embedding of SQL in F# by Cheney et al. (2013) expoits F# sequence notation. For EDSL, exploiting **do** notation just requires instantiating return and $(\gg\!=)$ correctly. For QDSL, it is also necessary for the normaliser to recognise and expand **do** notation and to substitute appropriate instances of return and $(\gg\!=)$.
- As this example shows, sometimes both EDSLs and QDSLs may require normalisation. Each EDSL usually has a distinct deep representation and so requires a distinct normaliser. In contrast, all QDSLs can share the representation of the quoted host language, and so can share a normaliser.

5.4 The deep embedding

We now review the usual approach to embedding a DSL into a host language by combining deep and shallow embedding. Much of this section reprises Svenningsson and Axelsson (2012).

Recall that a value of type $\mathit{Dp}\ \ a$ represents a term of type a, and is called a deep embedding.

```
data Dp a where
  LitB     :: Bool → Dp Bool
  LitI     :: Int → Dp Int
  LitF     :: Float → Dp Float
  If       :: Dp Bool → Dp a → Dp a → Dp a
  While    :: (Dp a → Dp Bool) →
                (Dp a → Dp a) → Dp a → Dp a
  Pair     :: Dp a → Dp b → Dp (a, b)
  Fst      :: Dp (a, b) → Dp a
  Snd      :: Dp (a, b) → Dp b
  Prim1    :: String → Dp a → Dp b
  Prim2    :: String → Dp a → Dp b → Dp c
  Arr      :: Dp Int → (Dp Int → Dp a) → Dp (Arr a)
  ArrLen   :: Dp (Arr a) → Dp Int
  ArrIx    :: Dp (Arr a) → Dp Int → Dp a
  Variable :: String → Dp a
```

The type above represents a low level, pure functional language with a straightforward translation to C. It uses higher-order abstract syntax (HOAS) to represent constructs with variable binding Pfenning and Elliot (1988).

The deep embedding has boolean, integer, and floating point literals, conditionals, while loops, pairs, primitives, arrays, and special-purpose constructs for variables and values. Constructs LitB, LitI, LitF build literals. Construct If builds a conditional. Construct While may require explanation. Rather than using side-effects, the while loop takes three arguments, a function from current state a to a boolean, and a function from current state a to new state a, and initial state a, and returns final state a. Constructs Pair, Fst, and Snd build pairs and extract the first and second component. Constructs $\mathit{Prim1}$ and $\mathit{Prim2}$ represent primitive operations, the string is the name of the operation. Construct ArrIx creates a new array from a length and a body that computes the array element for each index, construct ArrLen extracts the length from an array, and construct ArrIx fetches the element at a given index. Construct $\mathit{Variable}$ is used to generate C.

5.5 Class Syn

We introduce a type class Syn that allows us to convert shallow embeddings to and from deep embeddings.

```
class Syn a where
  type Internal a
  toDp   :: a → Dp (Internal a)
  fromDp :: Dp (Internal a) → a
```

Type $\mathit{Internal}$ is a GHC type family (Chakravarty et al. 2005). Functions toDp and fromDp translate between the shallow embedding a and the deep embedding $\mathit{Dp}\ (\mathit{Internal}\ a)$.

The first instance of Syn is Dp itself, and is straightforward.

```
instance Syn (Dp a) where
  type Internal (Dp a) = a
  toDp   = id
  fromDp = id
```

Our representation of a run-time Bool will have type $\mathit{Dp}\ \ \mathit{Bool}$ in both the deep and shallow embeddings, and similarly for Int and Float.

We do not code the target language using its constructs directly. Instead, for each constructor we define a corresponding "smart constructor" using class Syn.

```
true, false :: Dp Bool
true = LitB True
false = LitB False
(?) :: Syn a ⇒ Dp Bool → (a, a) → a
c ? (t, e) = fromDp (If c (toDp t) (toDp e))
while :: Syn a ⇒ (a → Dp Bool) → (a → a) → a → a
while c b i = fromDp (While (c ∘ fromDp)
                            (toDp ∘ b ∘ fromDp)
                            (toDp i))
```

Numbers are made convenient to manipulate via overloading.

```
instance Num (Dp Int) where
  a + b = Prim2 "(+)" a b
  a − b = Prim2 "(-)" a b
  a × b = Prim2 "(*)" a b
  fromInteger a = LitI (fromInteger a)
```

With this declaration, $1 + 2 :: \mathit{Dp}\ \mathit{Int}$ evaluates to

$$\mathit{Prim2}\ \texttt{"(+)"}\ (\mathit{LitI}\ 1)\ (\mathit{LitI}\ 2),$$

permitting code executed at generation-time and run-time to appear identical. A similar declaration works for $Float$.

Comparison also benefits from smart constructors.

$$(.{==}.) :: (Syn\ a, Eq\ (Internal\ a)) \Rightarrow a \to a \to Dp\ Bool$$
$$a\ .{==}.\ b = Prim2\ \texttt{"(==)"}\ (toDp\ a)\ (toDp\ b)$$

$$(.{<}.) :: (Syn\ a, Ord\ (Internal\ a)) \Rightarrow a \to a \to Dp\ Bool$$
$$a\ .{<}.\ b = Prim2\ \texttt{"(<)"}\ (toDp\ a)\ (toDp\ b)$$

Overloading cannot apply here, because Haskell requires $(==)$ return a result of type $Bool$, while $(.{==}.)$ returns a result of type $Dp\ Bool$, and similarly for $(.{<}.)$.

5.6 Embedding pairs

We set up a correspondence between host language pairs in the shallow embedding and target language pairs in the deep embedding.

$$\begin{array}{l}
\textbf{instance}\ (Syn\ a, Syn\ b) \Rightarrow Syn\ (a, b)\ \textbf{where} \\
\quad \textbf{type}\ Internal\ (a, b) = (Internal\ a, Internal\ b) \\
\quad toDp\ (a, b) = Pair\ (toDp\ a)\ (toDp\ b) \\
\quad fromDp\ p \quad = (fromDp\ (Fst\ p), fromDp\ (Snd\ p))
\end{array}$$

This permits us to manipulate pairs as normal, with (a, b), $fst\ a$, and $snd\ a$. (Argument p is duplicated in the definition of $fromDp$, which may require common subexpression elimination as discussed in Section 5.2.)

We have now developed sufficient machinery to define a for loop in terms of a $while$ loop.

$$\begin{array}{l}
for :: Syn\ a \Rightarrow Dp\ Int \to a \to (Dp\ Int \to a \to a) \to a \\
for\ n\ x\ b = snd\ (while\ (\lambda(i, x) \to i\ .{<}.\ n) \\
\qquad\qquad\qquad\qquad\quad (\lambda(i, x) \to (i + 1, b\ i\ x)) \\
\qquad\qquad\qquad\qquad\quad (0, x))
\end{array}$$

The state of the $while$ loop is a pair consisting of a counter and the state of the for loop. The body b of the for loop is a function that expects both the counter and the state of the for loop. The counter is discarded when the loop is complete, and the final state of the for loop returned.

Thanks to our machinery, the above definition uses only ordinary Haskell pairs. The condition and body of the $while$ loop pattern match on the state using ordinary pair syntax, and the initial state is constructed as a standard Haskell pair.

5.7 Embedding undefined

For the next section, which defines an analogue of the $Maybe$ type, it will prove convenient to work with types which have a distinguished value at each type, which we call $undef$.

It is straightforward to define a type class $Undef$, where type a belongs to $Undef$ if it belongs to Syn and it has an undefined value.

$$\begin{array}{l}
\textbf{class}\ Syn\ a \Rightarrow Undef\ a\ \textbf{where} \\
\quad undef :: a \\
\textbf{instance}\ Undef\ (Dp\ Bool)\ \textbf{where} \\
\quad undef = false \\
\textbf{instance}\ Undef\ (Dp\ Int)\ \textbf{where} \\
\quad undef = 0 \\
\textbf{instance}\ Undef\ (Dp\ Float)\ \textbf{where} \\
\quad undef = 0 \\
\textbf{instance}\ (Undef\ a, Undef\ b) \Rightarrow Undef\ (a, b)\ \textbf{where} \\
\quad undef = (undef, undef)
\end{array}$$

For example,

$$\begin{array}{l}
(/\#) \quad :: Dp\ Float \to Dp\ Float \to Dp\ Float \\
x\ /\#\ y = (y\ .{==}.\ 0)\ ?\ (undef, x\ /\ y)
\end{array}$$

behaves as division, save that when the divisor is zero it returns the undefined value of type $Float$, which is also zero.

Svenningsson and Axelsson (2012) claim that it is not possible to support $undef$ without changing the deep embedding, but here we have defined $undef$ entirely as a shallow embedding. (It appears they underestimated the power of their own technique!)

5.8 Embedding option

We now explain in detail the Opt type seen in Section 2.4.

The deep-and-shallow technique cleverly represents deep embeddding $Dp\ (a, b)$ by shallow embedding $(Dp\ a, Dp\ b)$. Hence, it is tempting to represent $Dp\ (Maybe\ a)$ by $Maybe\ (Dp\ a)$, but this cannot work, because $fromDp$ would have to decide at generation-time whether to return $Just$ or $Nothing$, but which to use is not known until run-time.

Indeed, rather than extending the deep embedding to support the type $Dp\ (Maybe\ a)$, Svenningsson and Axelsson (2012) prefer a different choice, that represents optional values while leaving Dp unchanged. Following their development, we represent values of type $Maybe\ a$ by the type $Opt'\ a$, which pairs a boolean with a value of type a. For a value corresponding to $Just\ x$, the boolean is true and the value is x, while for one corresponding to $Nothing$, the boolean is false and the value is $undef$. We define $some'$, $none'$, and opt' as the analogues of $Just$, $Nothing$, and $maybe$. The Syn instance is straightforward, mapping options to and from the pairs already defined for Dp.

$$\begin{array}{l}
\textbf{data}\ Opt'\ a = Opt'\ \{ def :: Dp\ Bool, val :: a \} \\
\textbf{instance}\ Syn\ a \Rightarrow Syn\ (Opt'\ a)\ \textbf{where} \\
\quad \textbf{type}\ Internal\ (Opt'\ a) = (Bool, Internal\ a) \\
\quad toDp\ (Opt'\ b\ x) = Pair\ b\ (toDp\ x) \\
\quad fromDp\ p \qquad\ = Opt'\ (Fst\ p)\ (fromDp\ (Snd\ p))
\end{array}$$

$$\begin{array}{lcl}
some' & :: & a \to Opt'\ a \\
some'\ x & = & Opt'\ true\ x \\
none' & :: & Undef\ a \Rightarrow Opt'\ a \\
none' & = & Opt'\ false\ undef \\
option' & :: & Syn\ b \Rightarrow b \to (a \to b) \to Opt'\ a \to b \\
option'\ d\ f\ o & = & def\ o\ ?\ (f\ (val\ o), d)
\end{array}$$

The next obvious step is to define a suitable monad over the type Opt'. The natural definitions to use are as follows:

$$\begin{array}{l}
return \quad :: a \to Opt'\ a \\
return\ x = some'\ x \\
(\gg\!=) \qquad :: (Undef\ b) \Rightarrow Opt'\ a \to (a \to Opt'\ b) \to Opt'\ b \\
o \gg\!= g \ \ = Opt'\ (def\ o\ ?\ (def\ (g\ (val\ o)), false)) \\
\qquad\qquad\qquad (def\ o\ ?\ (val\ (g\ (val\ o)), undef))
\end{array}$$

However, this adds type constraint $Undef\ b$ to the type of $(\gg\!=)$, which is not permitted. This need to add constraints often arises, and has been dubbed the constrained-monad problem (Hughes 1999; Sculthorpe et al. 2013; Svenningsson and Svensson 2013). We solve it with a trick due to Persson et al. (2011).

We introduce a second continuation-passing style (cps) type Opt, defined in terms of the representation type Opt'. It is straightforward to define $Monad$ and Syn instances for the cps type, operations to lift the representation type to cps and to lower cps to the representation type, and to lift $some$, $none$, and $option$ from the representation type to the cps type. The $lift$ operation is closely related to the $(\gg\!=)$ operation we could not define above; it is properly typed, thanks to the type constraint on b in the definition of $Opt\ a$.

$$\begin{array}{l}
\textbf{newtype}\ Opt\ a = \\
\quad O\ \{ unO :: \forall b. Undef\ b \Rightarrow ((a \to Opt'\ b) \to Opt'\ b) \}
\end{array}$$

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Monad}\ \mathit{Opt}\ \mathbf{where} \\
\quad \mathit{return}\ x \quad = O\ (\lambda g \rightarrow g\ x) \\
\quad m \ggg k \quad = O\ (\lambda g \rightarrow \mathit{unO}\ m\ (\lambda x \rightarrow \mathit{unO}\ (k\ x)\ g))
\end{array}
$$

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Undef}\ a \Rightarrow \mathit{Syn}\ (\mathit{Opt}\ a)\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Internal}\ (\mathit{Opt}\ a) = (\mathit{Bool}, \mathit{Internal}\ a) \\
\quad \mathit{fromDp} = \mathit{lift} \circ \mathit{fromDp} \\
\quad \mathit{toDp} \quad = \mathit{toDp} \circ \mathit{lower}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{lift} & :: & \mathit{Opt}'\ a \rightarrow \mathit{Opt}\ a \\
\mathit{lift}\ o & = & O\ (\lambda g \rightarrow \mathit{Opt}'\ (\mathit{def}\ o\ ?\ (\mathit{def}\ (g\ (\mathit{val}\ o)), \mathit{false})) \\
 & & \qquad\qquad\qquad (\mathit{def}\ o\ ?\ (\mathit{val}\ (g\ (\mathit{val}\ o)), \mathit{undef})))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{lower} & :: & \mathit{Undef}\ a \Rightarrow \mathit{Opt}\ a \rightarrow \mathit{Opt}'\ a \\
\mathit{lower}\ m & = & \mathit{unO}\ m\ \mathit{some}'
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{none} & :: & \mathit{Undef}\ a \Rightarrow \mathit{Opt}\ a \\
\mathit{none} & = & \mathit{lift}\ \mathit{none}'
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{some} & :: & a \rightarrow \mathit{Opt}\ a \\
\mathit{some}\ a & = & \mathit{lift}\ (\mathit{some}'\ a)
\end{array}
$$

$$
\begin{array}{l}
\mathit{option} \qquad :: (\mathit{Undef}\ a, \mathit{Undef}\ b) \Rightarrow \\
\qquad\qquad\qquad b \rightarrow (a \rightarrow b) \rightarrow \mathit{Opt}\ a \rightarrow b \\
\mathit{option}\ d\ f\ o = \mathit{option}'\ d\ f\ (\mathit{lower}\ o)
\end{array}
$$

These definitions support the EDSL code presented in Section 5.3.

5.9 Embedding vector

Array programming is central to the intended application domain of Feldspar. In this section, we extend our EDSL to handle arrays.

Recall that values of type Array are created by construct Arr, while ArrLen extracts the length and ArrIx fetches the element at the given index. Corresponding to the deep embedding Array is a shallow embedding Vec.

$$
\mathbf{data}\ \mathit{Vec}\ a = \mathit{Vec}\ (\mathit{Dp}\ \mathit{Int})\ (\mathit{Dp}\ \mathit{Int} \rightarrow a)
$$

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Syn}\ a \Rightarrow \mathit{Syn}\ (\mathit{Vec}\ a)\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Internal}\ (\mathit{Vec}\ a) = \mathit{Array}\ \mathit{Int}\ (\mathit{Internal}\ a) \\
\quad \mathit{toDp}\ (\mathit{Vec}\ n\ g) \quad = \mathit{Arr}\ n\ (\mathit{toDp} \circ g) \\
\quad \mathit{fromDp}\ a \qquad\quad = \mathit{Vec}\ (\mathit{ArrLen}\ a) \\
\qquad\qquad\qquad\qquad\quad (\lambda i \rightarrow \mathit{fromDp}\ (\mathit{ArrIx}\ a\ i))
\end{array}
$$

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Functor}\ \mathit{Vec}\ \mathbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{Vec}\ n\ g) = \mathit{Vec}\ n\ (f \circ g)
\end{array}
$$

The constructor Vec resembles the constructor Arr, but the former constructs a high-level representation of the array and the latter an actual array. It is straightforward to make Vec an instance of $\mathit{Functor}$.

It is straightforward to define operations on vectors, including combining corresponding elements of two vectors, summing the elements of a vector, dot product of two vectors, and norm of a vector.

$$
\begin{array}{l}
\mathit{zipVec} \ \ :: (\mathit{Syn}\ a, \mathit{Syn}\ b) \Rightarrow \\
\qquad\qquad\quad (a \rightarrow b \rightarrow c) \rightarrow \mathit{Vec}\ a \rightarrow \mathit{Vec}\ b \rightarrow \mathit{Vec}\ c \\
\mathit{zipVec}\ f\ (\mathit{Vec}\ m\ g)\ (\mathit{Vec}\ n\ h) \\
\qquad = \mathit{Vec}\ (m\ \text{`}\mathit{min}\text{`}\ n)\ (\lambda i \rightarrow f\ (g\ i)\ (h\ i))
\end{array}
$$

$$
\begin{array}{l}
\mathit{sumVec} :: (\mathit{Syn}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Vec}\ a \rightarrow a \\
\mathit{sumVec}\ (\mathit{Vec}\ n\ g) \\
\qquad = \mathit{for}\ n\ 0\ (\lambda i\ x \rightarrow x + g\ i)
\end{array}
$$

$$
\begin{array}{l}
\mathit{dotVec} \quad :: (\mathit{Syn}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Vec}\ a \rightarrow \mathit{Vec}\ a \rightarrow a \\
\mathit{dotVec}\ u\ v = \mathit{sumVec}\ (\mathit{zipVec}\ (\times)\ u\ v)
\end{array}
$$

$$
\begin{array}{l}
\mathit{normVec} \ \ :: \mathit{Vec}\ \mathit{Float} \rightarrow \mathit{Dp}\ \mathit{Float} \\
\mathit{normVec}\ v = \mathit{sqrt}\ (\mathit{dotVec}\ v\ v)
\end{array}
$$

The vector representation makes it easy to define any functions when each vector element is computed independently, including drop, take, $\mathit{reverse}$, vector concatentation, and the like, but is less well suited to functions with dependencies between elements, such as computing a running sum.

An important consequence of the style of definition we have adopted is that it provides lightweight fusion. The definition of dotVec would not produce good C code if it first computed $\mathit{zipVec}\ (\times)\ u\ v$, put the result into an intermediate vector w, and then computed $\mathit{sumVec}\ w$. Fortunately, it does not. Assume u is $\mathit{Vec}\ m\ g$ and v is $\mathit{Vec}\ n\ h$. Then we can simplify $\mathit{dotVec}\ u\ v$ as follows:

$$
\begin{array}{ll}
 & \mathit{toVec}\ u\ v \\
\rightsquigarrow & \mathit{sumVec}\ (\mathit{zipVec}\ (\times)\ u\ v) \\
\rightsquigarrow & \mathit{sumVec}\ (\mathit{zipVec}\ (\times)\ (\mathit{Vec}\ m\ g)\ (\mathit{Vec}\ n\ h) \\
\rightsquigarrow & \mathit{sumVec}\ (\mathit{Vec}\ (m\ \text{`}\mathit{min}\text{`}\ n)\ (\lambda i \rightarrow g\ i \times h\ i) \\
\rightsquigarrow & \mathit{for}\ (m\ \text{`}\mathit{min}\text{`}\ n)\ (\lambda i\ x \rightarrow x + g\ i \times h\ i)
\end{array}
$$

Indeed, we can see that by construction that whenever we combine two primitives the intermediate vector is always eliminated, a stronger guarantee than provided by conventional optimising compilers.

The type class Syn enables conversion between types Arr and Vec. Hence for EDSL, unlike QDSL, explicit calls toVec and $\mathit{fromVec}$ are not required. Invoking $\mathit{edsl}\ \mathit{normVec}$ produces the same code as in Section 2.6.

As with QDSL, there are some situations where fusion is not beneficial. We may materialise a vector as an array with the following function.

$$
\begin{array}{l}
\mathit{memorise} :: \mathit{Syn}\ a \Rightarrow \mathit{Vec}\ a \rightarrow \mathit{Vec}\ a \\
\mathit{memorise}\ (\mathit{Vec}\ n\ g) = \\
\quad = \mathit{Vec}\ n\ (\lambda i \rightarrow \mathit{fromDp}\ (\mathit{ArrIx}\ (\mathit{Arr}\ n\ (\mathit{toDp} \circ g))\ i))
\end{array}
$$

The above definition depends on common subexpression elimination to ensure $\mathit{Arr}\ n\ (\mathit{toDp} \circ g)$ is computed once, rather than once for each element of the resulting vector. If

$$
\mathit{blur} :: \mathit{Syn}\ a \Rightarrow \mathit{Vec}\ a \rightarrow \mathit{Vec}\ a
$$

averages adjacent elements of a vector, then one may choose to compute either

$$
\mathit{blur} \circ \mathit{blur} \quad \text{or} \quad \mathit{blur} \circ \mathit{memorise} \circ \mathit{blur}
$$

with different trade-offs between recomputation and memory usage.

EDSL silently converts between representation, while QDSL forces all conversions to be written out; following the pattern that EDSL is more compact, while QDSL is more explicit. For QDSL it is the subformula property which guarantees that all intermediate uses of Vec are eliminated, while for EDSL this is established by operational reasoning on the behaviour of the type Vec.

6. Related work

Domain specific languages are becoming increasingly popular as a way to deal with software complexity. Yet, they have a long and rich history (Bentley 1986).

In this paper we have, like many other DSL writers, used Haskell as it has proven to be very suitable for *embedding* domain specific languages (Gill 2014). Examples include Reid et al. (1999); Hudak (1997); Bjesse et al. (1998).

In this paper we have specifically built on the technique of combining deep and shallow embeddings (Svenningsson and Axelsson 2012) and contrasted it with our new QDSL technique. Languages which have used this technique include Feldspar (Axelsson et al. 2010), Obsidian (Svensson et al. 2011), Nikola (Mainland and Morrisett 2010), Hydra (Giorgidze and Nilsson 2011) and Meta-Repa (Ankner and Svenningsson 2013).

The vector type used in this paper is one of several types which enjoy fusion in the CDSL framework. Other examples include push arrays (Claessen et al. 2012) and sequential arrays and streams as used in Feldspar (Feldspar 2015).

The loss of sharing when implementing embedded DSLs was identified by ODonnell (1993) in the context of embedded circuit descriptions. Claessen and Sands (1999) proposed to introduce a little bit of impurity in Haskell, referred to as *observable sharing* to be able to recover from the loss of sharing. Later, Gill (2009) proposed a somewhat safer way of recover sharing, though still ultimately relying on impurity.

A proposition-as-types principle for quotation as a modal logic was proposed by Davies and Pfenning (1996, 2001). As they note in that paper, their technique has close connections to two-level languages (Nielson and Nielson 2005).

An early use of quotation in programming is Lisp (McCarthy 1960), and perhaps the first application of quotation to domain-specific languages is Lisp macros (Hart 1963).

The underlying idea for QDSLs was established by Cheney et al. (2013).

7. Conclusion

We have compared EDSLs and QDSLs, arguing that QDSLs offer competing expressiveness and efficiency. EDSLs often (but not always) mimic the syntax of the host language, and often (but not always) perform normalisation in the host languages, while QDSLs (always) steal the syntax of the host language, and (always) ensure the subformula property, at the cost of requiring a normaliser, one per host language.

The subformula property may have applications in DSLs other that QDSLs. For instance, after Section 5.8 of this paper was drafted, it occurred to us that a different approach to options in EDSL would be to extend type Dp with constructs for type $Maybe$. So long as type $Maybe$ does not appear in the input or output of the program, a normaliser that ensures the subformula property could guarantee that C code for such constructs need never be generated.

As we noted in the introduction, rather than build a special-purpose tool for each QDSL, it should be possible to design a single tool for each host language. Our next step is to design Haskell QDSL, with the following features.

- Full-strength type inference for the terms returned from typed quasi-quotations, restoring type information currently discarded by GHC.
- Based on the above, full support for type classes and overloading within quasi-quotation.
- The user may choose either an ADT or GADT representation of the term returned by typed quasi-quotation, whichever is more convenient.
- A normaliser to ensure the subformula property, which works with any datatype declared in Haskell.
- The user may supply a type environment indicating which constants (or free variables) may appear in typed quasi-quotations.

Such a tool could easily subsume the special-purpose translator from Qt to Dp described at the beginning of Section 3, and lift most of its restrictions. For instance, the current prototype is restricted to the $Maybe$ monad, while the envisioned tool will work with any monad.

Molière's Monsieur Jourdain was bemused to discover he had been speaking prose his whole life. Similarly, many of us have used QDSLs for years, if not by that name. DSL via quotation is the heart of Lisp macros, Microsoft LINQ, and Scala LMS, to name but three. We hope that by naming the concept and drawing attention to the central benefits of normalisation and the subformula propety, we may help the concept to flower further for decades to come.

Acknowledgement This work was funded by EPSRC Grant EP/K034413/1. Shayan Najd is a recipient of the Google Europe Fellowship in Programming Technology, and this research is supported in part by this Google Fellowship. Josef Svenningsson is a SICSA Visiting Fellow and is funded by a HiPEAC collaboration grant and by the Swedish Foundation for Strategic Research, under grant RawFP.

8. References

J. Ankner and J. Svenningsson. An EDSL approach to high performance Haskell programming. In *Haskell*, 2013.

E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A Domain Specific Language for Digital Signal Processing Algorithms. In *MEMOCODE*, 2010.

J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ICFP*, 1998.

M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL*, 2005.

J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.

K. Claessen and D. Sands. Observable sharing for functional circuit description. In *ASIAN*. Springer, 1999.

K. Claessen, M. Sheeran, and B. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP*, 2012.

Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Jr. Guy Steele, editor, *Principles of Programming Languages (POPL)*, pages 258–270, St. Petersburg Beach, Florida, 1996. ACM.

Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

Feldspar. Feldspar github repository, 2015. https://github.com/Feldspar/feldspar-language.

Cormac Flanagan, Amr Sabryand Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation (PLDI)*, pages 237–247. ACM, 1993.

Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

A. Gill. Domain-specific languages and code synthesis using Haskell. *Commun. ACM*, 57(6), 2014.

A. Gill. Type-safe observable sharing in Haskell. In *Haskell*, 2009.

G. Giorgidze and H. Nilsson. Embedding a functional hybrid modelling language in Haskell. In *IFL*. Springer, 2011.

Timothy P. Hart. MACRO definitions for LISP. Technical Report AIM-057, MIT, 1963.

W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–491. Academic Press, 1980. The original version was circulated privately in 1969.

P. Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3, 1997.

J. Hughes. Restricted data types in Haskell. In *Haskell*, 1999.

S. Lindley. Extensional rewriting with sums. In *TLCA*, 2007.

G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Haskell*, 2010.

J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *JFP*, 8(03):275–317, 1998.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4): 184–195, 1960.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*, volume 34. Cambridge university press, 2005.

J. ODonnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming*. Springer, 1993.

A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *IFL*, 2011.

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, 1988.

Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almqvist and Wiksell, Stockholm, 1965.

A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *ICSE*, 1999.

N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *ICFP*, 2013.

J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.

J. Svenningsson and B. Svensson. Simple and compositional reification of monadic embedded languages. In *ICFP*, 2013.

J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *IFL*. 2011.

Philip Wadler. Propositions as types. *Communications of the ACM*, 2015. To appear.

Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Principles of Database Systems (PODS)*, pages 26–36. ACM, 1993.