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

We describe a new approach to implementing Domain-Specific Languages (DSLs), called Quoted DSLs (QDSLs), that is inspired by two old ideas: quasi-quotation, from McCarthy's Lisp of 1960, and the subformula principle of normal proofs, from Gentzen's natural deduction of 1935. QDSLs reuse facilities provided for the host language, since host and quoted terms share the same syntax, type system, and normalisation rules. QDSL terms are normalised to a canonical form, inspired by the subformula principle, which guarantees that one can use higher-order types in the source while guaranteeing first-order types 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. The two variants produce identical code.

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 principle

1. Introduction

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 Embedded DSLs (EDSLs), which may use shallow embedding, deep embedding, or a combination of the two. In this paper we aim to reinvigorate the idea of building DSLs via quotation, by introducing an approach we call Quoted DSLs (QDSLs). A key feature of QDSLs is normalisation to a canonical form, inspired by the subformula principle identified by Gentzen (1935) and named by Prawitz (1965).

Cheney *et al.* (2013) describe a DSL for language-integrated query in F# that translates into SQL. Their technique depends on quotation, normalisation of quoted terms, and the subformula principle—an approach which we here dub QDSL. They conjecture that other DSLs might benefit from the same technique, particularly those that perform staged computation, where host code at generation-time computes target code 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 introduce a sharpened subformula principle. We compare QDSL and EDSL variants of Feldspar and assess the trade-offs between the two approaches.

QDSL terms are represented in a host language by terms in quotations (or more properly, quasi-quotations), where domain-specific constructs are represented as constants (free variables) in quotations. For instance, the following Haskell code defines a function that converts coloured images to greyscale, using QDSL Feldspar as discussed throughout the paper:

$$
\begin{array}{l}
\mathit{greyscale} :: \mathit{Qt}\ (\mathit{Img} \to \mathit{Img}) \\
\mathit{greyscale} = [||\lambda \mathit{img} \to \\
\quad \$\$\mathit{mapImg} \\
\quad (\lambda r\ g\ b \to \\
\qquad \mathbf{let}\ q = \mathit{div}\ ((30 \times r) + (59 \times g) + (11 \times b))\ 100 \\
\qquad \mathbf{in}\ \$\$\mathit{mkPxl}\ q\ q\ q)\ \mathit{img}||]
\end{array}
$$

We use a typed variant of Template Haskell (TH), an extension of GHC (Mainland 2012). Quotation is indicated by $[||...||]$, anti-quotation by $\$\$(\cdots)$, and the quotation of a Haskell term of type a has type $\mathit{Qt}\ \ a$. The domain-specific constructs used in the code are addition, multiplication, and division. The anti-quotations $\$\$\mathit{mapImg}$ and $\$\$\mathit{mkPxl}$ denote splicing user-defined functions named mapImg and mkPxl, which themselves are defined as quoted terms: mapImg is a higher-order function that applies a function from pixels to pixels to transform an image, where each pixel consists of three RGB colour values, and new pixels are created by mkPxl (see Section 2.7).

By allowing DSL terms to be written in the syntax of a host language, a DSL can reuse the facilities of the host language, including its syntax, its type system, its normalisation rules, its name resolution and module system, and tools including parsers and editors. Reification—representing terms as data manipulated by the host language—is key to many DSLs, which subject the reified term to processing such as optimisation, interpretation, and code generation. In EDSLs, the ability to reify syntactic features varies depending on the host language and the implementation technique, while in QDSLs, terms are always reified via quotation (for a general review see Gill (2014)). In this sense QDSLs reuse host language syntax entirely, without any restrictions. Further, since host and quoted terms share the same abstract syntax, QDSLs can reuse the internal machinery of the host language.

Wholesale reuse of the host language syntax has clear appeal, but also poses challenges. When targetting a first-order DSL, how can we guarantee that higher-order functions do not appear in the

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...
http://dx.doi.org/10.1145/2847538.2847541

generated code? How can we ensure loop fusion for arrays? When targetting a flat SQL query, how can we guarantee that nested datatypes do not appear in the generated query? The answer to all three questions is provided by normalising quoted terms, which are then guaranteed to satisfy Gentzen's subformula principle.

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. The subformula principle 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. Applying the principle of Propositions as Types (Howard 1980; Wadler 2015), the subformula principle states that every lambda term can be put into a normal form where its subterms can only have types that are subformulas of the type of the term and the types of the free variables.

The fact that we have access to the type of every subterm in a normal term, has a significant benefit for reasoning about QDSLs: just by checking that subformulas of the type of a quoted term and its free variables satisfy a specific predicate, we can guarantee that the type of every single subterm in the normalised quoted term satisfies the predicate. For instance, if the predicate asks for absence of higher-order function types, then we have the guarantee that after normalisation all uses of higher-order functions are normalised away. The subformula principle enables lifting reasoning about normal terms to types: reasoning is independent of the operations in the normaliser; any semantic-preserving normaliser respecting the subformula principle suffices.

The subformula principle provides users of the DSL with useful guarantees, such as the following: (a) they may write higher-order terms while guaranteeing to generate first-order code; (b) they may write a sequence of loops over arrays while guaranteeing to generate code that fuses those loops; (c) they may write intermediate terms with nested collections while guaranteeing to generate code that operates on flat data. Items (a) and (b) are used in this paper, and are key to generating C; while items (a) and (c) are used by Cheney *et al.* (2013) in F# and are key to generating SQL.

There exist many approaches that overlap with QDSLs: the Spoofax language workbench of Kats and Visser (2010), the Lightweight Modular Staging (LMS) framework of Rompf and Odersky (2010) in Scala, the combined deep and shallow embedding of Svenningsson and Axelsson (2012), and the macro system of Racket (Flatt 2010), to name a few (see Section 6). We believe QDSLs occupy an interesting point in the design space of DSLs, and deserve to be studied in their own right. Our goal is to characterise the key ingredients of QDSLs, reveal trade-offs, and enable cross-fertilisation of ideas between QDSLs and other approaches.

The contributions of this paper are:

- To introduce QDSLs as an approach to building DSLs based on quotation, reuse of host language internal machinery, and normalisation of quoted terms to a canonical form inspired by the subformula principle, by presenting the design of a QDSL variant of Feldspar (Section 2).
- To measure QDSL and EDSL implementations of Feldspar, and show they offer comparable performance (Section 3).
- To formulate a sharpened version of the subformula principle and apply it to characterise when higher-order terms normalise to first-order form, and to present a proof-of-concept normalisation algorithm for call-by-value and call-by-need that preserve sharing (Section 4).
- To compare the QDSL variant of Feldspar with the deep and shallow embedding approach used in the EDSL variant of Feldspar, and show they offer trade-offs with regard to ease of use (Section 5).

Section 6 describes related work, and Section 7 concludes.

Our QDSL and EDSL variants of Feldspar and benchmarks are available at `https://github.com/shayan-najd/QFeldspar`.

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, QDSL Feldspar, that follows the structure of the previous design closely, but using the methods of QDSL rather than EDSL. Section 5 compares the QDSL and EDSL designs.

2.1 The Top Level

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

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

Here 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 computes the function.

Not all types representable in Haskell are easily representable in 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.

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

2.2 A First Example

Let's begin with the "hello world" of program generation, the power function. Since division by zero is undefined, we arbitrarily chose to assert that raising zero to a negative power yields zero. The optimised power function represented using QDSL is as follows. For pedagogical purposes, we avoid techniques that further optimise this function but obscure its presentation.

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

Quotation is used to indicate which code executes at which time: anti-quoted code executes at generation-time while quoted code executes at run-time.

Invoking $qdsl\ (power\ (-6))$ generates code to raise a number to its -6th power. Evaluating $power\ (-6)$ yields the following:

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

Normalising as described in Section 4, with variables renamed for readability, yields the following:

$$\begin{array}{l} [||\lambda u \to \mathbf{if}\ u == 0\ \mathbf{then}\ 0\ \mathbf{else} \\ \quad \mathbf{let}\ v = u \times 1\ \mathbf{in} \end{array}$$

$\mathbf{let}\ w = u \times (v \times v)\ \mathbf{in}$
$1 \,/\, (w \times w)||]$

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 the subformula principle. From the normalised term it is easy to generate the final C code:

```
float prog (float u) {
  float w; float v; float r;
  if (u == 0.0) {
    r = 0.0;
  } else {
    v = (u * 1.0);
    w = (u * (v * v));
    r = (1.0f / (w * w));}
  return r;}
```

By default, we always generate a routine called `prog`; 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. In Haskell, an EDSL can use the same syntax for arithmetic operators, but must use a different syntax for equality tests and conditionals, as explained 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 QDSL Feldspar user be certain that such function applications do not render transformation to first-order code impossible or introduce additional runtime overhead? The answer is the subformula principle.

2.3 The Subformula Principle

Gentzen's subformula principle 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, the subformula principle guarantees that any term can be normalised so that the type of each of its subterms is a subformula of either the type of one of its free variables (corresponding to hypotheses) or of the term itself (corresponding to the conclusion). Here the subformulas of a type are the type itself and the subformulas of its parts, where the parts of $a \to b$ are a and b, the parts of (a, b) are a and b, and types Int and Float have no parts (see Theorem 4.2).

Further, it is easy to adapt the original proof to guarantee a sharpened subformula principle: any term can be normalised so that the type of each of its proper subterms is a proper subformula 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 subformulas of a type are all subformulas save for the type itself. In the example of the previous subsection, the sharpened subformula principle guarantees that after normalisation a closed term of type $\mathit{float} \to \mathit{float}$ will only have proper subterms of type float, which is indeed true for the normalised term (see Theorem 4.3).

The subformula principle depends on normalisation, but complete normalisation is not always possible or desirable. The extent of normalisation may be controlled by introducing uninterpreted constants. In particular, we introduce the uninterpreted constant

$$\mathit{save} :: \mathit{Rep}\ a \Rightarrow a \to a$$

of arity 1, which is equivalent to the identity function on representable types. Unfolding of an application $L\ M$ can be inhibited by rewriting it in the form $\mathit{save}\ \ L\ M$, where L and M are arbitrary terms. A use of save appears in Section 2.6. In a context with recursion, we take

$$\mathit{fix} :: (a \to a) \to a$$

as an uninterpreted constant.

2.4 A Second Example

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 of Haskell to refactor the code, by separating identification of 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 value.

The Maybe type is a part of the Haskell 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.

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

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 principle 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 $\mathbf{do}$ notation are expanded to applications of $(\gg\!=)$, as usual. Rather than taking return, $(\gg\!=)$, and maybe as uninterpreted constants (whose types have subformulas involving Maybe), we treat them as known definitions to be eliminated by the normaliser. Type $\mathit{Maybe}\ \ a$ is a sum type, and is normalised as described in Section 4.

2.5 While

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

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

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. So that we may compile while loops to C, the type of the state is constrained to representable types. We can define a for loop in terms of a while loop.

$$
\begin{aligned}
&\mathit{for} :: \mathit{Rep}\ s \Rightarrow \mathit{Qt}\ (\mathit{Int} \to s \to (\mathit{Int} \to s \to s) \to s)\\
&\mathit{for} = [||\lambda n\ s_0\ b \to \mathit{snd}\ (\mathit{while}\ (\lambda(i,s) \to i < n)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda(i,s) \to (i+1, b\ i\ s))\\
&\qquad\qquad\qquad\qquad\qquad\qquad (0, s_0))||]
\end{aligned}
$$

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. Here while, like snd and $(+)$, is a constant known to QDSL Feldspar, and so not enclosed in $$ anti-quotation.

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

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Qt}\ (\mathit{Int} \to \mathit{Int})\\
&\mathit{fib} = [||\lambda n \to \mathit{fst}\ (\$\$\mathit{for}\ n\ (0,1)\\
&\qquad\qquad\qquad\qquad (\lambda i\ (a,b) \to (b, a+b)))||]
\end{aligned}
$$

Again, the subformula principle 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 or constants of a given arity. As described in Section 4, we can ensure that after normalisation every occurrence 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 the bound variable s and the second and third ellipses all have the same type, that of the state of the while loop.

Unsurprisingly, and in accord with the subformula principle, 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 principle. 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. In particular, Maybe cannot appear in the state of a while loop, which is restricted to representable types. An alternative choice is possible, as we will see in the next section.

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 principle 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.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Arr}\ a = \mathit{Array}\ \mathit{Int}\ a\\
&\textbf{data}\ \mathit{Vec}\ a = \mathit{Vec}\ \mathit{Int}\ (\mathit{Int} \to a)
\end{aligned}
$$

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

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

For arrays, we assume the following primitive operations.

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

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. Array components must be representable.

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

$$
\begin{aligned}
&\mathit{toVec} \quad\ :: \mathit{Rep}\ a \Rightarrow \mathit{Qt}\ (\mathit{Arr}\ a \to \mathit{Vec}\ a)\\
&\mathit{toVec} \quad\ = [||\lambda a \to \mathit{Vec}\ (\mathit{lnArr}\ a)\ (\lambda i \to \mathit{ixArr}\ a\ i)||]\\
&\mathit{fromVec} :: \mathit{Rep}\ a \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Arr}\ a)\\
&\mathit{fromVec} = [||\lambda(\mathit{Vec}\ n\ g) \to \mathit{mkArr}\ n\ g||]
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{lnVec} \quad :: \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Int})\\
&\mathit{lnVec} \quad = [||\lambda(\mathit{Vec}\ l\ g) \to l||]\\
&\mathit{ixVec} \quad :: \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Int} \to a)\\
&\mathit{ixVec} \quad = [||\lambda(\mathit{Vec}\ l\ g) \to g||]\\
&\mathit{minim} \ \ :: \mathit{Ord}\ a \Rightarrow \mathit{Qt}\ (a \to a \to a)\\
&\mathit{minim} \ \ = [||\lambda x\ y \to \textbf{if}\ x < y\ \textbf{then}\ x\ \textbf{else}\ y||]\\
&\mathit{zipVec} \ :: \mathit{Qt}\ ((a \to b \to c) \to \mathit{Vec}\ a \to \mathit{Vec}\ b \to \mathit{Vec}\ c)\\
&\mathit{zipVec} \ = [||\lambda f\ (\mathit{Vec}\ m\ g)\ (\mathit{Vec}\ n\ h) \to\\
&\qquad\qquad\quad \mathit{Vec}\ (\$\$\mathit{minim}\ m\ n)\ (\lambda i \to f\ (g\ i)\ (h\ i))||]\\
&\mathit{sumVec} :: (\mathit{Rep}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to a)\\
&\mathit{sumVec} = [||\lambda(\mathit{Vec}\ n\ g) \to \$\$\mathit{for}\ n\ 0\ (\lambda i\ x \to x + g\ i)||]\\
&\mathit{dotVec} \ :: (\mathit{Rep}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Vec}\ a \to a)\\
&\mathit{dotVec} \ = [||\lambda u\ v \to \$\$\mathit{sumVec}\ (\$\$\mathit{zipVec}\ (\times)\ u\ v)||]\\
\\
&\mathit{normVec} :: \mathit{Qt}\ (\mathit{Vec}\ \mathit{Float} \to \mathit{Float})\\
&\mathit{normVec} = [||\lambda v \to \mathit{sqrt}\ (\$\$\mathit{dotVec}\ v\ v)||]
\end{aligned}
$$

The fifth of these uses the for loop defined in Section 2.5.

Our generated program cannot accept Vec as input, since the Vec type is not representable, but it can accept Arr as input. For instance, if we invoke qdsl on

$$[||\$\$\mathit{normVec} \circ \$\$\mathit{toVec}||]$$

the quoted term normalises to

$$
\begin{aligned}
&[||\lambda a \to \mathit{sqrt}\ (\mathit{snd}\\
&\quad (\mathit{while}\ (\lambda s \to \mathit{fst}\ s < \mathit{lnArr}\ a)\\
&\qquad\quad (\lambda s \to \textbf{let}\ i = \mathit{fst}\ s\ \textbf{in}\\
&\qquad\qquad (i+1, \mathit{snd}\ s + (\mathit{ixArr}\ a\ i \times \mathit{ixArr}\ a\ i)))\\
&\qquad\quad (0, 0.0)))||]
\end{aligned}
$$

from which it is easy to generate C code.

The vector representation makes it easy to define any function where each vector element is computed independently, such as the examples above, vector append (appVec) and creating a vector of one element (uniVec), but is less well suited to functions with dependencies between elements, such as computing a running sum.

Types and the subformula principle help us to guarantee fusion. The subformula principle 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, in which case fusion will cause the elements to be recomputed. An alternative is to materialise the vector as an array with the following function.

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

Here we interpose save, as defined in Section 2.3 to forestall the fusion that would otherwise occur. For example, if

$$
\begin{aligned}
&\mathit{blur} :: \mathit{Qt}\ (\mathit{Vec}\ \mathit{Float} \to \mathit{Vec}\ \mathit{Float})\\
&\mathit{blur} = [||\lambda a \to \$\$\mathit{zipVec}\ (\lambda x\ y \to \mathit{sqrt}\ (x \times y))\\
&\qquad\qquad\qquad\qquad (\$\$\mathit{appVec}\ (\$\$\mathit{uniVec}\ 0)\ a)\\
&\qquad\qquad\qquad\qquad (\$\$\mathit{appVec}\ a\ (\$\$\mathit{uniVec}\ 0))||]
\end{aligned}
$$

computes the geometric mean of adjacent elements of a vector, then one may choose to compute either

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

with different trade-offs between recomputation and memory use. Strong guarantees for fusion in combination with $memorise$ give the programmer a simple interface which provides powerful optimisations combined with fine control over memory use.

Here we have applied the subformula principle to array fusion as based on "pull arrays" (Svenningsson and Axelsson 2012), but the same technique should also apply to other techniques that support array fusion, such as "push arrays" (Claessen *et al.* 2012).

2.7 Image Processing

We can use the operations defined for vectors to implement image processing algorithms. An image can be seen as a vector of vectors (of the same length) whose elements are the pixels (RGB). Length of the outer vector is the height of the image, and length of the inner vectors is the width of the image.

$$\begin{aligned}
&\mathbf{type}\ Img = Vec\ (Vec\ Pxl)\\
&mkImg :: Qt\ (Int \to Int \to (Int \to Int \to Pxl) \to Img)\\
&mkImg = [||\lambda h\ w\ ixf \to\\
&\quad Vec\ h\ (\lambda i \to Vec\ w\ (\lambda j \to ixf\ i\ j))||]\\
&hImg, wImg :: Qt\ (Img \to Int)\\
&hImg\ = [||\lambda img \to \$\$lnVec\ img||]\\
&wImg = [||\lambda img \to \$\$lnVec\ (\$\$ixVec\ img\ 0)||]\\
&getPxl :: Qt\ (Img \to Int \to Int \to Pxl)\\
&getPxl = [||\lambda v\ i\ j \to \$\$lnVec\ (\$\$ixVec\ v\ i)\ j||]
\end{aligned}$$

Pixels are triples containing the value of each colour channel.

$$\begin{aligned}
&\mathbf{type}\ Pxl = (Int, (Int, Int))\\
&mkPxl :: Qt\ (Int \to Int \to Int \to Pxl)\\
&mkPxl = [||\lambda r\ g\ b \to (r, (g, b))||]\\
&red, green, blue :: Qt\ (Pxl \to Int)\\
&red\quad\ = [||\lambda p \to fst\ p\qquad\qquad ||]\\
&green = [||\lambda p \to fst\ (snd\ p)\ ||]\\
&blue\ \ = [||\lambda p \to snd\ (snd\ p)||]
\end{aligned}$$

We can define a higher-order function for mapping over pixels, which are useful for defining algorithms such as $greyscale$.

$$\begin{aligned}
&mapImg :: Qt\ ((Int \to Int \to Int \to Pxl) \to Img \to Img)\\
&mapImg = [||\lambda f\ img \to\\
&\quad \$\$mkImg\ (\$\$hImg\ img)\ (\$\$wImg\ img)\\
&\quad (\lambda i\ j \to \mathbf{let}\ p = \$\$getPxl\ img\ i\ j\ \mathbf{in}\\
&\quad\quad f\ (\$\$red\ p)\ (\$\$green\ p)\ (\$\$blue\ p))||]
\end{aligned}$$

3. Implementation

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

The transformer from Qt to Dp performs the following steps.

- To simplify normalisation, in any context where a constant c is not fully applied, it replaces c with $\lambda\overline{x}.\, c\ \ \overline{x}$. 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 principle, using the rules of Section 4. The normaliser supports a limited set of types, including tuples, $Maybe$, and Vec.
- It performs simple type inference, which is used to resolve overloading. Overloading is limited to a fixed set of cases, including overloading arithmetic operators.
- 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 $while$, $makeArr$, $lenArr$, $ixArr$, and $save$.

An unfortunate feature of typed quotation in GHC is that the implementation discards all type information when creating the representation of a term. Thus, the translator from $Qt\ a$ to $Dp\ a$ is forced to re-infer all types for subterms, and for this reason we support only limited overloading, and we translate the $Maybe$ monad as a special case rather than supporting overloading for monad operations in general.

The backend performs three transformations over Dp terms before compiling to C. First, common subexpressions are recognised and transformed to **let** bindings. Second, Dp terms are normalised using exactly the same rules used for normalising Qt terms, as described in Section 4. Third, Dp terms are optimised using η contraction for conditionals and arrays:

$$\begin{aligned}
\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ M \quad &\mapsto \quad M\\
makeArr\ (lenArr\ M)\ (ixArr\ M) \quad &\mapsto \quad M
\end{aligned}$$

and a restricted form of linear inlining for **let** bindings that preserves the order of evaluation.

Figure 1 lists lines of code, benchmarks used, and performance results. The translator from Dp to C is shared by QDSL and EDSL Feldspar, and listed in a separate column. All five benchmarks run under QDSL and EDSL Feldspar generate identical C code, up to permutation of independent assignments, with identical compile and run times. The columns for QDSL and EDSL Feldspar give compile and run times for Haskell, while the columns for generated code give compile and run times for the generated C. QDSL compile times are slightly greater than EDSL. Most Haskell run times are too small to be meaningful, but FFT shows QDSL running four

Lines of Haskell code

	shared	unique	total
QDSL Feldspar	3970	1722	5962
EDSL Feldspar	3970	452	4422

Benchmarks

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

Performance

	QDSL Feldspar		EDSL Feldspar		Generated Code	
	Compile	Run	Compile	Run	Compile	Run
IPGray	16.96	0.01	15.06	0.01	0.06	0.39
IPBW	17.08	0.01	14.86	0.01	0.06	0.19
FFT	17.87	0.39	15.79	0.09	0.07	3.02
CRC	17.14	0.01	15.33	0.01	0.05	0.12
Window	17.85	0.02	15.77	0.01	0.06	0.27

Times in seconds; minimum time of ten runs.
Quad-core Intel i7-2640M CPU, 2.80 GHz, 3.7 GiB RAM.
GHC 7.8.3; GCC 4.8.2; Ubuntu 14.04 (64-bit).

Figure 1. Comparison of QDSL and EDSL Feldspar

times slower than EDSL, the slow-down being due to normalisation time (our normaliser was not designed to be particularly efficient).

4. The Subformula Principle

This section introduces reduction rules for normalising terms that enforce the subformula principle while preserving sharing. 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 constant applications are non-values, we represent literals as free variables. As usual, terms are taken as equivalent up to renaming of bound variables. We 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. Let Γ range over type environments, which pair variables with types, and write $\Gamma \vdash M : A$ to indicate that term M has type A under type environment Γ. Typing rules are standard. We omit them due to lack of space.

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.

We write $M \mapsto_i N$ to indicate that M reduces to N in phase i. 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. Reductions are 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 (Bondorf and Danvy 1991), naming subterms, 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 two kinds of reduction: β rules apply when an introduction (construction) is immediately followed by an elimination (deconstruction), and κ rules push eliminators closer to introducers to enable β rules. Phase 3 "garbage collects" unused terms as in the call-by-need lambda calculus (Maraist *et al.* 1998; Ariola and Felleisen 1997). 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.

THEOREM 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 a term are the term itself and its components. For instance, the subterms of $\lambda x.\, N$ are itself and the subterms of N and the subterms of $L\ M$ are itself and the subterms of L and M. The *proper subterms* of a term are all its subterms other than the term itself.

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

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

and its subformulas are itself and $A_1, \ldots, A_k$, and B (but not $A_i \to \cdots \to A_k \to B$ 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 principle.

THEOREM 4.2 (Subformula Principle). *If $\Gamma \vdash M : A$ and M is in normal form, then every subterm of M has a type that is a subformula of either A, a type in Γ, or 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 principle, below), and that our reduction rules introduce **let**, in order to ensure sharing is preserved.

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

THEOREM 4.3 (Sharpened Subformula Principle). *If $\Gamma \vdash M : A$ and M is in normal form, then every proper subterm of M that is not a free variable or a subterm of a constant application has a type that is a proper subformula of either A or a type in Γ.*

We believe we are the first to formulate the sharpened version.

The sharpened subformula principle says nothing about the types of subterms of constant applications, but such types are immediately apparent by recursive application of the sharpened subformula principle. 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 free variable of M_i 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, we require that every top-level term passed to *qdsl* is suitable for translation to C after normalisation, and any DSL translating to a *first-order* language must impose a similar requirement. 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 property required is not hard to formulate in a general way, and is easy to ensure by applying the sharpened subformula principle.

Take the representable types to be any set closed under subformulas that does not include function types. 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 $\max(m + 1, n)$ where m is the rank of A and n is the rank of B, while a term of representable type has rank 0. We say a term is *first-order* when every subterm is either representable, or is of the form $\lambda\overline{x}.\, N$ where each bound variable and the body is of representable type. The following characterises translation to a first-order language.

THEOREM 4.4 (First-Order Subformula Principle). *Let M be a term of rank 1, in which every free variable has rank 0 and every constant rank at most 2. Then M normalises to a first-order term.*

The theorem follows immediately by observing that any term L of rank 1 can be rewritten to the form $\lambda\overline{y}.\, (L\ \overline{y})$ where each bound variable and the body has representable type, and then normalising and applying the sharpened subformula principle.

In QDSL Feldspar, while is a constant with type of rank 2 and other constants have types of rank 1. Section 2.6 gives an

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

Figure 2. Types and Terms

Phase 1 (let-insertion)

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

$$(\mathit{let}) \quad F[M] \quad \mapsto_1 \quad \mathbf{let}\ x = M\ \mathbf{in}\ F[x], \quad x \text{ fresh}, M \text{ not a value}$$

Phase 2 (symbolic evaluation)

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

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

Phase 3 (garbage collection)

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

Figure 3. Normalisation Rules

example of a normalised term. By Theorem 4.4, each subterm has a representable type (boolean, integer, float, or a pair of an integer and float) or is a lambda abstraction with bound variables and body of representable type; and it is this property which ensures it is easy to generate C code from the term.

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 trade-offs between the QDSL and EDSL approaches. Much of this section reprises Svenningsson and Axelsson (2012).

The top-level function of EDSL Feldspar has the type:

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

Here $\mathit{Dp}\ a$ is the deep representation of a term of type a. The deep representation is described in detail in Section 5.3 below, and is chosen to be easy to translate to C. Since Feldspar is first-order, there is no constructor for terms of type $\mathit{Dp}\ (A \to B)$. Instead, to represent functions in Feldspar, host-level functions are used, i.e. terms of type $\mathit{Dp}\ A \to \mathit{Dp}\ B$. As before, type C represents code in C, and type class Rep denotes representable types.

5.1 A First Example

Here is the power function of Section 2.2, written as an EDSL:

$$
\begin{array}{l}
\mathit{power} :: \mathit{Int} \to \mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float} \\
\mathit{power}\ n\ x = \\
\quad \mathbf{if}\ n < 0\ \mathbf{then} \\
\qquad x \mathrel{.{=}{=}.} 0\ ?\ (0, 1\ /\ \mathit{power}\ (-n)\ x) \\
\quad \mathbf{else\ if}\ n == 0\ \mathbf{then} \\
\qquad 1 \\
\quad \mathbf{else\ if}\ \mathit{even}\ n\ \mathbf{then} \\
\qquad \mathbf{let}\ y = \mathit{power}\ (n\ \mathrm{div}\ 2)\ x\ \mathbf{in}\ y \times y \\
\quad \mathbf{else} \\
\qquad x \times \mathit{power}\ (n-1)\ x
\end{array}
$$

Type $Q\ (\mathit{Float} \to \mathit{Float})$ in the QDSL variant becomes the type $\mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float}$ 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 the EDSL variant, 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 $\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N$ for the former but $M \mathrel{.{=}{=}.} N$ and $L\ ?\ (M, N)$ for the latter.

Invoking $\mathit{edsl}\ (\mathit{power}\ (-6))$ generates code to raise a number to its -6 power. Evaluating $\mathit{power}\ (-6)\ u$, where u is a term representing a variable of type $\mathit{Dp}\ \mathit{Float}$, yields the following:

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

Applying Common-Subexpression Elimination (CSE) permits recovering sharing structure.

$$
\begin{array}{r|l}
v & (u \times 1) \\
w & u \times (v \times v) \\
\text{top} & (u \mathrel{.{=}{=}.} 0)\ ?\ (0, 1\ /\ (w \times w))
\end{array}
$$

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

Here are points of comparison between the two approaches.

- A function $a \to b$ is embedded in QDSL as $Qt\ (a \to b)$, a representation of a function, and in EDSL as $Dp\ a \to Dp\ b$, a function between representations.
- QDSL enables host and embedded languages to appear identical. In contrast, in Haskell, EDSL requires some term forms, such as comparison and conditionals, to differ between host and embedded languages. Other languages, notably Scala Virtualised (Rompf *et al.* 2013), support more general overloading that allows even comparison and conditionals to be identical.
- QDSL requires syntax to separate quoted and anti-quoted terms. In contrast, EDSL permits host and embedded languages to intermingle seamlessly. Depending on your point of view, explicit quotation syntax may be considered an unnecessary 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.
- 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 by domain. In contrast, EDSL typically develops custom shallow and deep embeddings for each application; a notable exception is the LMS/Delite framework (Sujeeth *et al.* 2014) for Scala, which shares a deep embedding across several disparate DSLs (Sujeeth *et al.* 2013).
- QDSL always yields a term that requires normalisation. In contrast, EDSL uses smart constructors to yield a term already in normal form, though in some cases a normaliser is still required (see Section 5.2).
- Since QDSLs may share the same quoted terms across a range of applications, the cost of building a normaliser or a preprocessor might be amortised across multiple QDSLs for a single language. In the conclusion, we consider the design of a tool for building QDSLs that uses a shared normaliser and preprocessor.
- Back-end code generation is similar for both approaches.

5.2 A Second Example

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

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

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

Type class $Undef$ is explained in Section 5.6, and details of type Opt are given in Section 5.7.

Here is the refactored code.

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

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

$(((fst\ ((x == 0.0)\ ?\ (((False\ ?\ (True, False)), (False\ ?\ (undef, undef))), (True, (1.0\ /\ ((x \times ((x \times 1.0) \times (x \times 1.0))) \times (x \times ((x \times 1.0) \times (x \times 1.0)))))))))\ ?\ (True, False))\ ?\ (((fst\ ((x == 0.0)\ ?\ (((False\ ?\ (True, False)), (False\ ?\ (undef, undef))), (True, (1.0\ /\ ((x \times ((x \times 1.0) \times (x \times 1.0))) \times (x \times ((x \times 1.0) \times (x \times 1.0)))))))))\ ?\ ((snd\ ((x == 0.0)\ ?\ (((False\ ?\ (True, False)), (False\ ?\ (undef, undef))), (True, (1.0\ /\ ((x \times ((x \times 1.0) \times (x \times 1.0))) \times (x \times ((x \times 1.0) \times (x \times 1.0))))))))), undef)), 0.0))$

Before, evaluating $power$ yielded a term essentially in normal form. However, to obtain a normal form here, rewrite rules must be repeatedly applied, as described in Section 3. After applying these rules, common subexpression elimination yields the same structure, and ultimately the same C code as in the previous subsection.

Here we have described normalisation via rewriting, but some EDSLs achieve normalisation via smart constructors, which ensure deep terms are always in normal form (Rompf 2012); the two techniques are roughly equivalent.

Hence, an advantage of the EDSL approach—that it generates terms essentially in normal form—turns out to apply sometimes but not others. It appears to often work for functions and products, but to fail for sums. In such situations, separate normalisation is required. This is one reason why we do not consider normalisation as required by QDSL to be particularly onerous.

5.3 The Deep Embedding

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{data}\ Dp\ a\ \textbf{where}} \\
\quad LitB & :: Bool \to Dp\ Bool \\
\quad LitI & :: Int \to Dp\ Int \\
\quad LitF & :: Float \to Dp\ Float \\
\quad If & :: Dp\ Bool \to Dp\ a \to Dp\ a \to Dp\ a \\
\quad While & :: (Dp\ a \to Dp\ Bool) \to \\
 & \quad\ (Dp\ a \to Dp\ a) \to Dp\ a \to Dp\ a \\
\quad Pair & :: Dp\ a \to Dp\ b \to Dp\ (a, b) \\
\quad Fst & :: Rep\ b \Rightarrow Dp\ (a, b) \to Dp\ a \\
\quad Snd & :: Rep\ a \Rightarrow Dp\ (a, b) \to Dp\ b \\
\quad Prim1 & :: Rep\ a \Rightarrow String \to Dp\ a \to Dp\ b \\
\quad Prim2 & :: (Rep\ a, Rep\ b) \Rightarrow \\
 & \quad\ String \to Dp\ a \to Dp\ b \to Dp\ c \\
\quad MkArr & :: Dp\ Int \to (Dp\ Int \to Dp\ a) \to Dp\ (Arr\ a) \\
\quad LnArr & :: Rep\ a \Rightarrow Dp\ (Arr\ a) \to Dp\ Int \\
\quad IxArr & :: Dp\ (Arr\ a) \to Dp\ Int \to Dp\ a \\
\quad Save & :: Dp\ a \to Dp\ a \\
\quad Let & :: Rep\ a \Rightarrow Dp\ a \to (Dp\ a \to Dp\ b) \to Dp\ b \\
\quad Variable & :: String \to Dp\ a
\end{array}
$$

Type Dp 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 Elliott 1988). Our code obeys the invariant that we only write $Dp\ a$ when $Rep\ a$ holds, that is, when type a is representable.

The deep embedding has boolean, integer, and floating point literals, conditionals, while loops, pairs, primitives, arrays, and special-purpose constructs to disable normalisation, for let binding, and for variables. Constructs $LitB$, $LitI$, $LitF$ build literals; If builds a conditional. $While$ corresponds to $while$ in Section 2.5; $Pair$, Fst, and Snd build and decompose pairs; $Prim1$ and $Prim2$ represent primitive operations, where the string is the name of the operation; $MkArr$, $LnArr$, and $IxArr$ correspond to the array operations in Section 2.6; $Save$ corresponds to $save$ in Section 2.3; Let corresponds to let binding, and $Variable$ is used when translating HOAS to C code.

5.4 Class Syn

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

$$\begin{array}{l}
\textbf{class}\ Rep\ (Internal\ a) \Rightarrow Syn\ a\ \textbf{where} \\
\quad \textbf{type}\ Internal\ a \\
\quad toDp \quad :: a \to Dp\ (Internal\ a) \\
\quad fromDp :: Dp\ (Internal\ a) \to a
\end{array}$$

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

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

$$\begin{array}{l}
\textbf{instance}\ Rep\ a \Rightarrow Syn\ (Dp\ a)\ \textbf{where} \\
\quad \textbf{type}\ Internal\ (Dp\ a) = a \\
\quad toDp \quad = id \\
\quad fromDp = id
\end{array}$$

Our representation of a run-time $Bool$ has type $Dp\ \ Bool$ 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.

$$\begin{array}{l}
true, false :: Dp\ Bool \\
true = LitB\ True \\
false = LitB\ False \\
(?) :: Syn\ a \Rightarrow Dp\ Bool \to (a, a) \to a \\
c\ ?\ (t, e) = fromDp\ (If\ \ c\ (toDp\ t)\ (toDp\ e)) \\
while :: Syn\ a \Rightarrow (a \to Dp\ Bool) \to (a \to a) \to a \to a \\
while\ c\ b\ i = fromDp \\
\quad (While\ (c \circ fromDp)\ (toDp \circ b \circ fromDp)\ (toDp\ i))
\end{array}$$

Numbers are made convenient to manipulate via overloading.

$$\begin{array}{l}
\textbf{instance}\ Num\ (Dp\ Int)\ \textbf{where} \\
\quad a + b = Prim2\ \texttt{"(+)"}\ a\ b \\
\quad a - b = Prim2\ \texttt{"(-)"}\ a\ b \\
\quad a \times b = Prim2\ \texttt{"(*)"}\ a\ b \\
\quad fromInteger\ \ a = LitI\ (fromInteger\ \ a)
\end{array}$$

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

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

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

Comparison also benefits from smart constructors.

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

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

Here is how to compute the minimum of two values.

$$\begin{array}{l}
minim \qquad :: (Syn\ a, Ord\ (Internal\ a)) \Rightarrow a \to a \to a \\
minim\ x\ y = (x\ .{<}.\ y)\ ?\ (x, y)
\end{array}$$

5.5 Embedding Pairs

Host language pairs in the shallow embedding correspond to 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 5.1.

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

$$\begin{array}{l}
for :: Syn\ a \Rightarrow Dp\ Int \to a \to (Dp\ Int \to a \to a) \to a \\
for\ n\ s_0\ b = snd \\
(while\ (\lambda(i, s) \to i\ .{<}.\ n)\ (\lambda(i, s) \to (i + 1, b\ i\ s))\ (0, s_0))
\end{array}$$

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.

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 an ordinary pair.

5.6 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 has an undefined value.

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

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!)

5.7 Embedding Option

We now explain in detail the Opt type seen in Section 5.2.

The deep-and-shallow technique represents deep embedding $Dp\ \ (a, b)$ 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.

Instead, Svenningsson and Axelsson (2012) 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 $option'$ as the analogues of $Just$, $Nothing$, and $maybe$. The Syn instance is straightforward, mapping options to and from the pairs already defined for Dp.

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

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

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

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

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

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

$$
\begin{array}{l}
\mathbf{newtype}\ Opt\ a = \\
\quad O\ \{unO :: \forall b. Undef\ b \Rightarrow ((a \to Opt'\ b) \to Opt'\ b)\} \\
\mathbf{instance}\ Monad\ Opt\ \mathbf{where} \\
\quad return\ x = O\ (\lambda g \to g\ x) \\
\quad m \ggg= k \ = O\ (\lambda g \to unO\ m\ (\lambda x \to unO\ (k\ x)\ g)) \\
\mathbf{instance}\ Undef\ a \Rightarrow Syn\ (Opt\ a)\ \mathbf{where} \\
\quad \mathbf{type}\ Internal\ (Opt\ a) = (Bool, Internal\ a) \\
\quad fromDp = lift \circ fromDp \\
\quad toDp \quad\ = toDp \circ lower \\
lift :: Opt'\ a \to Opt\ a \\
lift\ o = \quad O\ (\lambda g \to Opt'\ (def\ o\ ?\ (def\ (g\ (val\ o)), false)) \\
\qquad\qquad\qquad\qquad\qquad (def\ o\ ?\ (val\ (g\ (val\ o)), undef))) \\
lower :: Undef\ a \Rightarrow Opt\ a \to Opt'\ a \\
lower\ m = unO\ m\ some' \\
none :: Undef\ a \Rightarrow Opt\ a \\
none = lift\ none' \\
some :: a \to Opt\ a \\
some\ a = lift\ (some'\ a) \\
option :: (Undef\ a, Syn\ b) \Rightarrow b \to (a \to b) \to Opt\ a \to b \\
option\ d\ f\ o = option'\ d\ f\ (lower\ o)
\end{array}
$$

These definitions support the EDSL code presented in Section 5.2.

5.8 Embedding Vector

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

$$
\begin{array}{l}
\mathbf{data}\ Vec\ a = Vec\ (Dp\ Int)\ (Dp\ Int \to a) \\
\mathbf{instance}\ Syn\ a \Rightarrow Syn\ (Vec\ a)\ \mathbf{where} \\
\quad \mathbf{type}\ Internal\ (Vec\ a) = Array\ Int\ (Internal\ a) \\
\quad toDp\ (Vec\ n\ g) \quad = MkArr\ n\ (toDp \circ g) \\
\quad fromDp\ a \qquad\ = Vec\ (LnArr\ a)\ (fromDp \circ IxArr\ a) \\
\mathbf{instance}\ Functor\ Vec\ \mathbf{where} \\
\quad fmap\ f\ (Vec\ n\ g) = Vec\ n\ (f \circ g)
\end{array}
$$

Constructor Vec resembles 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 $Functor$.

It is easy 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.

$$
\begin{array}{l}
zipVec \quad :: (Syn\ a, Syn\ b) \Rightarrow \\
\qquad\qquad\qquad (a \to b \to c) \to Vec\ a \to Vec\ b \to Vec\ c \\
zipVec\ f\ (Vec\ m\ g)\ (Vec\ n\ h) \\
\qquad\quad = Vec\ (m\ `minim`\ n)\ (\lambda i \to f\ (g\ i)\ (h\ i)) \\
sumVec :: (Syn\ a, Num\ a) \Rightarrow Vec\ a \to a \\
sumVec\ (Vec\ n\ g) = for\ n\ 0\ (\lambda i\ x \to x + g\ i) \\
dotVec \quad\ :: (Syn\ a, Num\ a) \Rightarrow Vec\ a \to Vec\ a \to a \\
dotVec\ u\ v = sumVec\ (zipVec\ (\times)\ u\ v) \\
normVec \quad :: Vec\ (Dp\ Float) \to Dp\ Float \\
normVec\ v = sqrt\ (dotVec\ v\ v)
\end{array}
$$

Invoking $edsl$ on $normVec \circ toVec$ generates C code to normalise a vector. A top-level function of type $(Syn\ a, Syn\ b) \Rightarrow (a \to b) \to C$ would insert the $toVec$ coercion automatically.

This style of definition again provides fusion. For instance:

$$
\begin{array}{cl}
& dotVec\ (Vec\ m\ g)\ (Vec\ n\ h) \\
= & sumVec\ (zipVec\ (\times)\ (Vec\ m\ g)\ (Vec\ n\ h)) \\
= & sumVec\ (Vec\ (m\ `minim`\ n)\ (\lambda i \to g\ i \times h\ i)) \\
= & for\ (m\ `minim`\ n)\ (\lambda i\ x \to x + g\ i \times h\ i)
\end{array}
$$

Indeed, we can see that by construction that whenever we combine two primitives the intermediate vector is always eliminated.

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

As with QDSL, sometimes fusion is not beneficial. We may materialise a vector as an array with the following function.

$$
\begin{array}{l}
memorise :: Syn\ a \Rightarrow Vec\ a \to Vec\ a \\
memorise = fromDp \circ Save \circ toDp
\end{array}
$$

Here we interpose $Save$ to forestall the fusion that would otherwise occur. For example, if

$$
\begin{array}{l}
blur :: Syn\ a \Rightarrow Vec\ a \to Vec\ a \\
blur\ v = zipVec\ (\lambda x\ y \to sqrt\ (x \times y)) \\
\qquad\qquad\qquad (appVec\ a\ (uniVec\ 0)) \\
\qquad\qquad\qquad (appVec\ (uniVec\ 0)\ a)
\end{array}
$$

computes the geometric mean of adjacent elements of a vector, then one may compute either $blur \circ blur$ or $blur \circ memorise \circ blur$ with different trade-offs between recomputation and memory use.

QDSL forces all conversions to be written out, while EDSL silently converts between representations; following the pattern that QDSL is more explicit, while EDSL is more compact. For QDSL it is the subformula principle 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. Discussion and Related Work

DSLs have a long and rich history (Bentley 1986). 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 work of Cheney *et al.* (2013) on a QDSL for language integrated query is extended with information flow security by Schoepe *et al.* (2014) and to nested results by Cheney *et al.* (2014b). Related work combines language-integrated query with effect types (Cooper 2009; Lindley and Cheney 2012). Cheney *et al.* (2014a) compare approaches based on quotation and effects. Suzuki *et al.* (2015) adapt the idea to an EDSL using the finally-tagless approach, which supports user-extensible syntax and optimisations, extending the core system with grouping and aggregation.

Davies and Pfenning (2001) also suggest quotation as a foundation for staged computation (Taha and Sheard 2000; Eckhardt *et al.* 2007), 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 the utility of normalising quoted terms, although they do not mention the subformula principle. As they note, their technique has close connections to two-level languages (Nielson and Nielson 1992) and partial evaluation (Jones *et al.* 1993).

The .NET Language-Integrated Query (LINQ) framework as used in C# and F# (Meijer *et al.* 2006; Syme 2006), and the Lightweight Modular Staging (LMS) framework as used in Scala (Rompf and Odersky 2010), exhibit 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). Scala LMS exploits techniques found in both QDSL (quotation and normalisation) and EDSL (combining shallow and deep embeddings), see Rompf *et al.* (2013), and exploits reuse to allow multiple DSLs to share infrastructure, see Sujeeth *et al.* (2013).

The subformula principle 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. As an example, a conservativity result holds for databases which states that nested queries are no more expressive than flat queries (Wong 1996). This conservativity result, as implied by the subformula principle, is used by 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 principle depends on normalisation, but normalisation may lead to exponential blowup in the size of the normalised code when there are nested conditionals; and hyperexponential blowup in complex cases involving higher-order functions. We have shown how uninterpreted constants allow the user to control where normalisation does and does not occur, while still maintaining the subformula principle. Future work is required to consider trade-offs between full normalisation as required for the subformula principle and special-purpose normalisation as used in many DSLs; possibly a combination of both will prove fruitful.

A desirable property of a DSL is that every type-correct term should successfully generate code in the target language. QDSL implementations may perform type checking at compile-time via the quotation machinery, or at run-time together with code generation. For implementation convenience, QFeldspar lies somewhere between the two: type-checking is done by Template Haskell, but under Haskell's typing context, leaving type-checks at run-time to ensure that only permitted operators are mentioned in quotations.

EDSL techniques are closely connected to reduction-free normalisation in the form of normalisation by evaluation (Dybjer and Filinski 2000; Lindley 2005) or type-directed partial evaluation (Danvy 1996). We are currently formalising the connections.

This paper uses Haskell, which has been widely used for EDSLs (Hudak 1997). We contrast QDSL with an EDSL technique that combines deep and shallow embedding (Svenningsson and Axelsson 2012), as used in several Haskell EDSLs including 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).

O'Donnell (1993) identified loss of sharing in the context of embedded circuit descriptions. Claessen and Sands (1999) proposes an extension of Haskell to support observable sharing. Gill (2009) proposes library features that support sharing using stable names extension(Peyton Jones *et al.* 2000). Thanks to quotations, sharing in QDSLs can be recovered by retrieving abstract syntax of quoted terms; no additional technique is required.

We have successfully implemented QDSLs in Template Haskell and previously in F# (Cheney *et al.* 2013). More generally, QDSLs can be implemented in any statically typed language that supports some form of quotation and splicing along with a means for analysing the abstract syntax and types of quoted terms.

7. Conclusion

We have compared QDSLs and EDSLs, arguing that QDSLs offer competing expressiveness and efficiency.

The subformula principle may have applications in DSLs other than QDSLs. For instance, after Section 5.7 of this paper was drafted, it occurred to us that a different approach would be to extend type Dp with constructs for type $Maybe$. As long as type $Maybe$ does not appear in the input or output of the program, a normaliser that enjoys the subformula principle can guarantee that C code for such constructs need never be generated.

Rather than building 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 a QDSL library for Haskell that restores the type information for quotations currently discarded by GHC and uses this to support type classes and overloading in full, and to supply a more general normaliser. Such a tool would subsume the special-purpose translator from Qt to Dp described at the beginning of Section 3, and lift most of its restrictions.

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 lies at the heart of Lisp macros, Microsoft LINQ, and Scala LMS, to name but three. By naming the concept and drawing attention to the benefits of normalisation and the subformula principle, we hope to help the concept to prosper for years to come.

Acknowledgements Thanks to Nada Amin, Emil Axelsson, James Cheney, Ryan Newton, and Tiark Rompf for comments on an earlier draft. Najd was funded by a Google Europe Fellowship in Programming Technology. Svenningsson held a SICSA Visiting Fellowship and was funded by a HiPEAC collaboration grant, and by the Swedish Foundation for Strategic Research under grant RawFP. Lindley and Wadler were funded by EPSRC Grant EP/K034413/1.

References

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

Z. Ariola and M. Felleisen. The call-by-need lambda calculus. *JFP*, 7(03), 1997.

E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE*. IEEE, 2010.

J. Bentley. Programming pearls: Little languages. *CACM*, 29(8), 1986.

A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *SCP*, 16(2), 1991.

M. M. T. Chakravarty, G. Keller, S. L. P. Jones, and S. Marlow. Associated types with class. In *POPL*. ACM, 2005.

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

J. Cheney, S. Lindley, G. Radanne, and P. Wadler. Effective quotation: relating approaches to language-integrated query. In *PEPM*. ACM, 2014.

J. Cheney, S. Lindley, and P. Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD*, pages 1027–1038. ACM, 2014.

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*. ACM, 2012.

E. Cooper. The script-writer's dream. In *DBPL*. ACM, 2009.

O. Danvy. Type-directed partial evaluation. In *POPL*. ACM, 1996.

R. Davies and F. Pfenning. A modal analysis of staged computation. *JACM*, 48(3), 2001.

P. Dybjer and A. Filinski. Normalization and partial evaluation. In *APPSEM*. Springer, 2000.

J. Eckhardt, R. Kaiabachev, E. Pašalić, K. Swadi, and W. Taha. Implicitly heterogeneous multi-stage programming. *New Generation Computing*, 25(3), 2007.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*. ACM, 1993.

M. Flatt. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010.

G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1), 1935.

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

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

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

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

W. Howard. The formulae-as-types notion of construction. In *To H. B. Curry*. Academic Press, 1980.

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

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

N. Jones, C. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall, 1993.

L. Kats and E. Visser. The spoofax language workbench. In *OOPSLA*. ACM, 2010.

S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*. ACM, 2012.

S. Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, University of Edinburgh, 2005.

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

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

G. Mainland. Explicitly heterogeneous metaprogramming with Meta-Haskell. In *ICFP*. ACM, 2012.

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

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *CACM*, 3(4), 1960.

E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*. ACM, 2006.

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991.

F. Nielson and H. Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

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

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

S. Peyton Jones, S. Marlow, and C. Elliott. Stretching the storage manager: Weak pointers and stable names in Haskell. In *IFL*. Springer, 2000.

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

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

T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 2010.

T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: linguistic reuse for deep embeddings. In *HOSC*. Springer, 2013.

T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

D. Schoepe, D. Hedin, and A. Sabelfeld. SeLINQ: tracking information across application-database boundaries. In J. Jeuring and M. M. T. Chakravarty, editors, *ICFP*. ACM, 2014.

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

A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*. Springer, 2013.

A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: a compiler architecture for performance-oriented embedded domain-specific languages. *TECS*, 13(4s), 2014.

K. Suzuki, O. Kiselyov, and Y. Kameyama. Finally, safely-extensible and efficient language integrated query. In *PEPM*. ACM, 2015.

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

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

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

D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML*. ACM, 2006.

W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *TCS*, 248(1), 2000.

P. Wadler. Propositions as types. *CACM*, 2015.

L. Wong. Normal forms and conservative extension properties for query languages over collection types. *JCSS*, 52(3), 1996.