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:

```haskell
let q = div ((30 \times r) + (59 \times g) + (11 \times b)) 100
in $$\text{mapImg}$$ ($$\text{mkPxl}$$ q q q)$$
```

We use a typed variant of Template Haskell (TH), an extension of GHC (Mainland 2012). Quotation is indicated by $$[\cdots]$$, anti-quotation by $$\langle\cdots\rangle$$, and the quotation of a Haskell term of type $$a$$ has type $$\langle a \rangle$$. The domain-specific constructs used in the code are addition, multiplication, and division. The anti-quotations $$\text{mapImg}$$ and $$\text{mkPxl}$$ denote splicing user-defined functions named $$\text{mapImg}$$ and $$\text{mkPxl}$$, which themselves are defined as quoted terms: $$\text{mapImg}$$ 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 $$\text{mkPxl}$$ (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 targeting a first-order DSL, how can we guarantee that higher-order functions do not appear in the...
generated code? How can we ensure loop fusion for arrays? When

targeting a flat SQL query, how can we guarantee that nested
data types do not appear in the generated query? The answer to all
three questions is provided by normalising quoted terms, which are
therefore guaranteed to satisfy Gentzen’s subformula principle.

The subformulas of a formula are its subparts; for instance, the
subformulas of $A \rightarrow B$ 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 gen-
erate 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 # 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 embed-
ding of Venningsson and Axelfson (2012), and the macro system
of Racket (Platt 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 charac-
terise 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 5).

• 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 normalisa-
tion algorithm for call-by-value and call-by-need that preserve
sharing (Section 5).

• 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

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:

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

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.

instance Rep Int
instance Rep Float
instance (Rep a, Rep b) \Rightarrow Rep (a, b)

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 arbitrar-
y choose to assert that raising zero to a negative power yields zero.

Let’s begin with the “hello world” of program generation, the
power function. Since division by zero is undefined, we arbitrar-
y choose to assert that raising zero to a negative power yields zero.

$power :: Int \rightarrow Qt (Float \rightarrow Float)$

power n =
if n < 0 then
  [[|$\lambda x \rightarrow if x == 0 then 0 else 1 / ($n$ power n) x|]]
else if n == 0 then
  [[|$\lambda x \rightarrow 1$|]]
else if even n then
  [[|$\lambda x \rightarrow let y = $n power y$ in y * y$|]]
else
  [[|$\lambda x \rightarrow x * ($n power 2$) $|]]

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

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

[[|$\lambda x \rightarrow if x == 0 then 0 else 1 /
  (\lambda x \rightarrow let y = (\lambda x \rightarrow x) x
  (\lambda x \rightarrow let y = (\lambda x \rightarrow x x) y)
  in y x x$|]]

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

[[|$\lambda u \rightarrow if u == 0 then 0 else
  let v = u \times 1 in

26
let $w = u \times (v \times v)$ 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:

```c
float prog (float u) {
    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 2.5.

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 `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

```
save :: Rep a \Rightarrow a \to a
```

by rewriting it in the form `save L M`, where $L$ and $M$ are arbitrary terms. A use of `save` appears in Section 2.7. In a context with recursion, we take

```
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 `Nothing` in the exceptional case, and the second maps `Nothing` to a suitable value.

The `Maybe` type is a part of the Haskell standard Prelude.

```haskell
data Maybe a = Nothing | Just a
maybe :: b \to (a \to b) \to Maybe a \to b
return :: a \to Maybe a

\(\triangleright=\) :: Maybe a \to (a \to Maybe b) \to Maybe b
```

Here is the refactored code.

```
power' :: Int \to Qt (Float \to Maybe Float)
power' n =
    if n < 0 then
        |||\(\lambda x \to (f x) 0 || Nothing\) ||
        else do x \leftarrow $$\(\text{power'} (-n)\) x
        return (1 / y) ||
    else if n == 0 then
        |||\(\lambda x \to return 1\) ||
        else do x \leftarrow $$\text{power'} (n \div 2)\) x
        return (y \times y) ||
        else
do x \leftarrow $$\text{power'} (n - 1)\) x
        return (x \times y) ||

power'' :: Int \to Qt (Float \to Float)
power'' n =
    |||\(\lambda x \to maybe 0 (\lambda y \to $$\text{power'} n\) x)\)||
```

Evaluation and normalisation of `power (-6)` and `power'' (-6)` yield identical terms (up to renaming), and hence applying `qds` 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 `do` notation are expanded to applications of `\(\triangleright=\)`, as usual. Rather than taking `return`, `\(\triangleright=\)`, and `maybe` as uninterpreted constants (whose types have subformulas involving `Maybe`), we treat them as known definitions to be eliminated by the normaliser. Type `Maybe a` is a sum type, and is normalised as described in Section 4.4.

### 2.5 While

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

```
while :: Rep s \Rightarrow (s \to 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 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.
for :: Rep s ⇒ Qt (Int → s → (Int → s → s) → s)
for = [[[\lambda s_0 b → snd (while (\lambda (i, s) → i < n)
  ((\lambda i, s) → (i + 1, b i s))
  (0, s_0))]]]

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.

fib :: Qt (Int → Int)
fib = [[[\lambda n → fst ($$for n (0, 1)
  (\lambda (a, b) → (b, a + b)))]]]

Again, the subformula principle plays a key role. As explained
in Section 2.3 primitives of the language to be compiled, such as
(×) 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

while (\lambda s → ···) (\lambda s → ···) ···

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 sub-
terms with the type of its state. The restriction of state to repre-
sentable 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 alterna-
tive 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 sub-
terms 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.

type Arr a = Array Int a
data Vec a = Vec Int (Int → a)

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

instance Rep a ⇒ Rep (Arr a)

For arrays, we assume the following primitive operations.
mkArr :: Rep a ⇒ Int → (Int → a) → Arr a
lnArr :: Rep a ⇒ Arr a → Int
ixArr :: Rep a ⇒ Arr a → Int → a

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.

toVec :: Rep a ⇒ Qt (Arr a → Vec a)
toVec = [[[\lambda a → Vec (lnArr a (\lambda i → ixArr a i))]]
fromVec :: Rep a ⇒ Qt (Vec a → Arr a)
fromVec = [[[\lambda (Vec n g) → mkArr n g]]]

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

lnVec :: Qt (Vec a → Int)
lnVec = [[[\lambda (Vec l g) → l]]]
ixVec :: Qt (Vec a → Int → a)
ixVec = [[[\lambda (Vec l g) → g]]]
minim :: Ord a ⇒ Qt (a → a → a)
minim = [[[\lambda x y → if x < y then x else y]]]
zipVec :: Qt ((a → b → c) → Vec a → Vec b → Vec c)
zipVec = [[[\lambda (Vec m g) (Vec n h) →
  Vec ($$minim m n) (\lambda i → f (g i) (h i))]]]
sumVec :: (Rep a, Num a) ⇒ Qt (Vec a → a)
sumVec = [[[\lambda (Vec n g) → $$for n 0 (\lambda i → x + g i)]]
dotVec :: (Rep a, Num a) ⇒ Qt (Vec a → Vec a → a)
dotVec = [[[\lambda a v → $$sumVec ($$zipVec (×) a a)]]

normVec :: Qt (Vec Float → Float)
normVec = [[[\lambda v → sqrt ($$dotVec v v)]]]

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

[[$$normVec o $$toVec]]

the quoted term normalises to

[[$$\lambda a → sqrt (snd
  (while (\lambda s → fst s < lnArr a)
    (\lambda s → let i = fst s in
      (i + 1, snd s + (ixArr a i × ixArr a i))
    (0, 0)))]]]

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
elements 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.

memorise :: Rep a ⇒ Qt (Vec a → Vec a)
memorise = [[[$$toVec o save o $$fromVec]]]

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

blur :: Qt (Vec Float → Vec Float)
blur = [[[\lambda a → $$sumVec (λx y → sqrt (x × y))
  ($$appVec ($$uniVec 0) a)
  ($$appVec a ($$uniVec 0))]]]
computes the geometric mean of adjacent elements of a vector, then one may choose to compute either

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

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.

```haskell
class Imaging a where
  flipImage :: a img → a img
  flipH     :: a img → a img
  flipV     :: a img → a img
  mirror    :: a img → a img
```

Pixels are triples containing the value of each colour channel.

```haskell
mkImg :: Qt img → a img
mkImg = [[\#\$\text{flipImage}]]
```

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

```haskell
mapImg :: Qt img → a img
mapImg = [[\$\$\text{map}]]
```

### 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 a 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 \(\lambda x. c\). It replaces identifiers connected to the type Maybe, such as return, (\(\approx\)), 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.

### Lines of Haskell code

<table>
<thead>
<tr>
<th>QDSL Feldspar</th>
<th>EDSL Feldspar</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>Run</td>
<td>Compile</td>
</tr>
<tr>
<td>IPGray</td>
<td>16.96</td>
<td>0.01</td>
</tr>
<tr>
<td>IPBW</td>
<td>17.08</td>
<td>0.01</td>
</tr>
<tr>
<td>FFT</td>
<td>17.87</td>
<td>0.39</td>
</tr>
<tr>
<td>CRC</td>
<td>17.14</td>
<td>0.01</td>
</tr>
<tr>
<td>Window</td>
<td>17.85</td>
<td>0.02</td>
</tr>
</tbody>
</table>

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

- 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: (\(\approx\)), (\(\prec\)), (\(\geq\)), (\(\times\)), 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 into let bindings. Second, Dp terms are normalised using exactly the same rules used for normalising Qt terms, as described in Section 5. Third, Dp terms are optimised using \(\eta\) contraction for conditionals and arrays:

```haskell
if L then M else M \rightarrow M
```

```haskell
\text{makeArr} (\text{lenArr} M) (\text{ixArr} M) \rightarrow M
```

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 times faster than EDSL.
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 1. Let A, B, C
range over types, including base types (o), functions (A → B),
products (A × 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(T) for the set of free variables
of T, 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 Γ ⊢ 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 2. 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 ↦→ 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 λ-normal form (Planagan 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 η rules push eliminators
closer to introducers to enable β rules. Phase 3 “garbage collects”
unused terms as in the call-by-need lambda calculus (Marcet 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 ↦→i is confluent and strongly normalising: all ↦→i
reduction sequences on well-typed terms are finite.

The only non-trivial proof is for ↦→2, which can be proved via a
standard reducibility argument (see, for example, Lindley 2007).
If the target language includes general recursion, normalisation
should take the fixpoint operator as an uninterpreted constant.

The subformulas of a type are the type itself and its compo-
ments. For instance, the subformulas of A → 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 Ax. 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 A₁, . . . , Aₖ, and B (but not
A₁ \to \cdots \to Aₖ \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 M₁, . . . , Mₖ (but not c M₁ \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 Γ ⊢ 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 Γ ⊢ 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 sub-
formula principle. Given a subterm that is a constant application
c τ_i, where c has type A → 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 require-
ment. 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 → bool and s → 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 subfor-
mulas 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 → B has rank \( m + 1, n \)
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 ev-
ery subterm is either representable, or is of the form \( \lambda \ x \. \ N \) where
each bound variable and the body is of representable type. The fol-
lowing 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 \( \lambda \ x \. (L \ y) \) 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
example of a normalised term. By Theorem 5.4, 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 represents Svenningsson and Axelson [2012].

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

\[
\text{edsl} : (\text{Rep } a, \text{Rep } b) \Rightarrow (Dp a \rightarrow Dp b) \rightarrow C
\]

Here 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 Dp (A → B). Instead, to represent functions in Feldspar, host-level functions are used, i.e. terms of type Dp A → 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:

```
power :: Int → Dp Float → Dp Float
power n x =
  if n < 0 then
    x := 0 ? (0, 1 / power (-n) x)
  else if n == 0 then
    1
  else if even n then
    let y = power (n div 2) x in y * y
  else
    x * power (n - 1) x
```

Type Q (Float → Float) in the QDSL variant becomes the type Dp Float → Dp Float in the EDSL variant, meaning that 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. However, as explained later, comparison and conditionals appear differently depending on 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. However, as explained later, comparison and conditionals appear differently depending on whether they are to be executed at generation-time or run-time.

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

\[
(u := 0) ? (0, 1 / ((u × ((u × 1) × (u × 1))) × (u × ((u × 1) × (u × 1)))))
\]

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

\[
\begin{align*}
v & (u × 1) \\
w & u × (v × v) \\
top & (u := 0) ? (0, 1 / (w × w))
\end{align*}
\]

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.

```haskell
type Opt a
none \::\: Undef a \Rightarrow Opt a
some \::\: a \Rightarrow Opt a
return :: a \Rightarrow Opt a
(\( \gg= \)) :: Opt a \to (a \to Opt b) \to Opt b
option :: (Undef a, Undef b) \Rightarrow
   b \to (a \to b) \Rightarrow Opt a \to b
```

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.

```haskell
def power' (n - 1) x
  return (x * y)
```

```haskell
power'' n x =
  do y <- power' (n - 1) x
  return (x * y)
```

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

\[
\begin{align*}
((fst \ (x === 0.0)) \ ? (f (x \times (x \times 1.0) \times (x \times 1.0)) \times (x \times (x \times 1.0) \times (x \times 1.0))))) \ ? (True,
& \ False)) \ ? (f (x \times 0.0) \ ? (f (x \times 0.0)) \ ? (f (x \times 0.0)))) \ ? (snd
& ((False \ ? (True, False)), (False \ ? (undef, undef)))) \ ? (True,\ 1.0 / ((x \times (x \times 1.0) \times (x \times 1.0)) \times (x \times (x \times 1.0))))))) \ ? (snd
& ((False \ ? (True, False)), (False \ ? (undef, undef)))) \ ? (True,\ 1.0 / ((x \times (x \times 1.0) \times (x \times 1.0)) \times (x \times (x \times 1.0)))))))
\end{align*}
\]

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 5. 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 (Romph 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.

```haskell
data Dp a where
LitB :: Bool \to Dp Bool
LitI :: Int \to Dp Int
LitF :: Float \to Dp Float
If :: Dp Bool \to Dp a \to Dp a \to Dp a
While :: (Dp a \to Dp Bool) \to
   (Dp a \to Dp a) \to Dp a \to Dp a
Pair :: Dp a \to Dp b \to Dp (a, b)
Fst :: Rep b \Rightarrow Dp (a, b) \to Dp a
Snd :: Rep a \Rightarrow Dp (a, b) \to Dp b
Prim1 :: Rep a \Rightarrow String \to Dp a \to Dp b
Prim2 :: (Rep a, Rep b) \Rightarrow
   String \to Dp a \to Dp b \to Dp c
MkArr :: Dp Int \to (Dp Int \to Dp a) \to Dp (Arr a)
LnArr :: Rep a \Rightarrow Dp (Arr a) \to Dp Int
IxArr :: Dp (Arr a) \to Dp Int \to Dp a
Save :: Dp a \to Dp a
Let :: Rep a \Rightarrow Dp a \to (Dp a \to Dp b) \to Dp b
Variable :: String \to Dp a
```

Type \( Dp\ a \) 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
5.4 Class \textit{Syn}

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

\begin{verbatim}
class Rep (Internal a) ⇒ Syn a where
type Internal a

toDp  : a ↦ Dp (Internal a)
fromDp : Dp (Internal a) ↦ a
\end{verbatim}

Type \textit{Internal} is a GHC type family (Chakravarty et al., 2005). Functions \textit{toDp} and \textit{fromDp} translate between the shallow embedding \textit{a} and the deep embedding \textit{Dp (Internal \textit{a})}. The first instance of \textit{Syn} is \textit{Dp} itself, and is straightforward.

\begin{verbatim}
instance Rep a ⇒ Syn (Dp a) where
type Internal (Dp a) = a
toDp = id
fromDp = id
\end{verbatim}

Our representation of a run-time \textit{Bool} has type \textit{Dp Bool} in both the deep and shallow embeddings, and similarly for \textit{Int} and \textit{Float}. We do not code the target language using its constructs directly. Instead, for each constructor we define a corresponding “smart constructor” using class \textit{Syn}.

\begin{verbatim}
true, false :: Dp Bool
true = LitB True
false = LitB False
\end{verbatim}

\begin{verbatim}
(?) :: Syn a ⇒ Dp Bool → (a, a) → a
(?) (t, e) = fromDp (if e (toDp t) (toDp e))
\end{verbatim}

\begin{verbatim}
while :: Syn a ⇒ (a → Dp Bool) → (a → a) → a → a
while c b i s = fromDp (While (c o fromDp) (toDp o b o fromDp) (toDp i))
\end{verbatim}

Numbers are made convenient to manipulate via overloading.

\begin{verbatim}
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)
\end{verbatim}

With this declaration, 1 + 2 :: Dp Int evaluates to \textit{Prim2 "+" (LitI 1) (LitI 2)} permitting code executed at generation-time and run-time to appear identical. A similar declaration works for \textit{Float}.

Comparison also benefits from smart constructors.

\begin{verbatim}
(==) :: (Syn a, Eq (Internal a)) ⇒ a → a → Dp Bool
a == b = Prim2 "==" (toDp a) (toDp b)
\end{verbatim}

\begin{verbatim}
(<) :: (Syn a, Ord (Internal a)) ⇒ a → a → Dp Bool
a < b = Prim2 "<" (toDp a) (toDp b)
\end{verbatim}

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

5.5 Embedding Pairs

Host language pairs in the shallow embedding correspond to target language pairs in the deep embedding.

\begin{verbatim}
instance (Syn a, Syn b) ⇒ Syn (a, b) where
type Internal (a, b) = (Internal a, Internal b)
toDp (a, b) = Pair (toDp a) (toDp b)
fromDp p = (fromDp (Fst p), fromDp (Snd p))
\end{verbatim}

This permits us to manipulate pairs as normal, with \textit{a, b, fst \textit{a}, snd \textit{a}}, and \textit{Opt \textit{a}}. Argument \textit{p} is duplicated in the definition of \textit{fromDp}, which may require common subexpression elimination as discussed in Section 5.1.

We have now developed sufficient machinery to define a \textit{for} loop in terms of a \textit{while} loop.

\begin{verbatim}
for :: Syn a ⇒ Dp Int → a → (Dp Int → a → a) → a
for n s b = snd
(while (λ(i, s) → i <. n) (λ(i, s) → (i + 1, b i s)) (0, s0))
\end{verbatim}

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

Thanks to our machinery, the above definition uses only ordinary Haskell pairs. The condition and body of the \textit{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 \textit{Maybe} type, it will prove convenient to work with types which have a distinguished value at each type, which we call \textit{undef}. It is straightforward to define a type class \textit{Unde}, where type \textit{a} belongs to \textit{Unde} if it belongs to \textit{Syn} and has an undefined value.

\begin{verbatim}
class Syn a ⇒ Unde a where unde :: a
\end{verbatim}

\begin{verbatim}
instance Unde (Dp Bool) where unde = false
instance Unde (Dp Int) where unde = 0
instance Unde (Dp Float) where unde = 0
instance (Unde a, Unde b) ⇒ Unde (a, b) where
unde = (unde, unde)
\end{verbatim}

For example,

\begin{verbatim}
(/#) :: Dp Float → Dp Float → Dp Float
x /# y = (y === 0) ? (undef, x / y)
\end{verbatim}

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

[Venningsson and Axelsson, 2012] claim that it is not possible to support \textit{undef} without changing the deep embedding, but here we have defined \textit{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 \textit{Opt} type seen in Section 5.2.

The deep-and-shallow technique represents deep embedding \textit{Dp (a, b)} by shallow embedding \textit{(Dp \textit{a, Dp \textit{b}})}. Hence, it is tempting to represent \textit{Dp (Maybe \textit{a})} by \textit{Maybe (Dp \textit{a})}, but this cannot work, because \textit{fromDp} would have to decide at generation-time whether to return \textit{Just} or \textit{Nothing}, but which to use is not known until run-time.
Instead, Svenningsson and Axelsson [2012] represent values of type \( \text{Maybe}\ a \) by the type \( \text{Opt}\ a \), which pairs a boolean with a value of type \( a \). For a value corresponding to \textit{Just} \( x \), the boolean is true and the value is \( x \), while for one corresponding to \textit{Nothing}, the boolean is false and the value is \textit{undef}. We define \textit{some'}, \textit{none'}, and \textit{option'} as the analogues of \textit{Just}, \textit{Nothing}, and \textit{maybe}. The \textit{Syn} instance is straightforward, mapping options to and from the pairs already defined for \textit{Dp}.

\begin{verbatim}
data Opt' a = Opt' { def :: Dp Bool, val :: a } instance Syn a ⇒ Syn (Opt' a) where
  type Internal (Opt' a) = (Bool, Internal a)
  toDp (Opt' b x) = Pair b (toDp x)
  fromDp p = Opt' (fst p) (fromDp (snd p))

  some' :: a ⇒ Opt' a
  some' x = Opt' true x

  none' :: Undef a ⇒ Opt' a
  none' = Opt' false undef

  option' :: Syn b ⇒ b → (a → b) → Opt' a → b
  option' d f o = def o ? (f (val o), d)
\end{verbatim}

The next obvious step is to define a suitable monad over the type \( \text{Opt}' \). The natural definitions to use are as follows:

\begin{verbatim}
return :: a ⇒ Opt' a
return x = some' x

(⇒) :: (Undef b ⇒ Opt' a ⇒ (a → Opt' b) → Opt' b)
o ⇒ g = Opt' (def o ? (def (g (val o)), false))
  (def o ? (val (g (val o)), undef))
\end{verbatim}

However, this adds type constraint \textit{Undef} \( b \) to the type of \( (⇒) \), 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, \( \text{Opt} \), defined in terms of \( \text{Opt}' \). It is straightforward to define \textit{Monad} and \textit{Syn} instances, operations to lift the representation type, operations to lift and lower one type to the other, and operations to lift \textit{some}, \textit{none}, and \textit{option} to the CPS type. The \textit{lift} operation is closely related to the \( (⇒) \) operation we could not define above; it is properly typed, thanks to the type constraint on \( b \) in the definition of \( \text{Opt} a \).

\begin{verbatim}
newtype Opt a =
  O \{ unO :: ∀b. Undef b ⇒ ((a → Opt' b) → Opt' b) \}

instance Monad Opt where
  return x = O (λg → g x)
  m ⇒ k = O (λx → unO m (λx → unO (k x) g))

instance Unfold a ⇒ Syn (Opt a) where
  type Internal (Opt a) = (Bool, Internal a)
  fromDp = lift ∘ fromDp
  toDp = toDp ∘ lower

  lift :: Opt a → Opt a
  lift o =
    O (λg → Opt' (def o ? (def (g (val o)), false))
          (def o ? (val (g (val o)), undef)))

  lower :: Undef a ⇒ Opt a → Opt' a
  lower m = unO m some'

  none :: Undef a ⇒ Opt a
  none = lift none'

  some :: a ⇒ Opt a
  some a = lift (some' a)
\end{verbatim}

option :: (Undef a, Syn b) ⇒ b → (a → b) → Opt a → b
  option d f o = option’ d f (lower a)

These definitions support the EDSL code presented in Section 5.2.

5.8 Embedding Vector
Recall that values of type \( \text{Array} \) are created by construct \( \text{MkArr} \), while \( \text{LnArr} \) extracts the length and \( \text{IzArr} \) fetches the element at the given index. Corresponding to the deep embedding \( \text{Arr} \) is a shallow embedding \( \text{Vec} \).

\begin{verbatim}
data Vec a = Vec (Dp Int) (Dp Int → a)

instance Syn a ⇒ Syn (Vec a) where
  type Internal (Vec a) = Array Int (Internal a)
  toDp (Vec n g) = MkArr n (toDp ∘ g)
  fromDp a = Vec (LnArr a) (fromDp ∘ IzArr a)

instance Functor Vec where
  fmap f (Vec n g) = Vec n (f ∘ g)
\end{verbatim}

Constrctor \( \text{Vec} \) resembles \( \text{Arr} \), but the former constructs a high-level representation of the array and the latter an actual array. It is straightforward to make \( \text{Vec} \) an instance of \textit{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{verbatim}
zipVec :: (Syn a, Syn b) ⇒
  (a → b → c) → Vec a → Vec b → Vec c

zipVec f (Vec m g) (Vec n h) = Vec (m 'minim' n) (λi → f (g i) (h i))

sumVec :: (Syn a, Num a) ⇒
  Vec a → a

sumVec (Vec n g) = for n (λi → x + g i)

dotVec :: (Syn a, Num a) ⇒
  Vec a → Vec a → a

dotVec u v = sumVec (zipVec (×) u v)

normVec ::
  Vec (Dp Float) → Dp Float

normVec v = sqrt (dotVec v v)
\end{verbatim}

Invoking \textit{edsl} on \textit{normVec} ∘ \textit{toVec} generates \textit{C} code to normalise a vector. A top-level function of type \( (\text{Syn} a, \text{Syn} b) ⇒ (a → b) → C \) would insert the \textit{toVec} coercion automatically.

This style of definition again provides fusion. For instance:

\begin{verbatim}
dotVec (Vec m g) (Vec n h) =
  sumVec (zipVec (×) (Vec m g) (Vec n h))

sumVec (Vec (m 'minim' n) (λi → g i × h i)) =
  for (m 'minim' n) (λi → x + g i × h i)
\end{verbatim}

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

The type class \( \text{Syn} \) enables conversion between types \( \text{Arr} \) and \( \text{Vec} \). Hence for EDSL, unlike QDSL, explicit calls to \textit{toVec} and \textit{fromVec} are not required. Invoking \textit{edsl} \( \text{normVec} \) produces the same \textit{C} code as in Section 5.6.

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

\begin{verbatim}
memorise :: Syn a ⇒ Vec a → Vec a
memorise = fromDp ∘ Save ∘ toDp
\end{verbatim}

Here we interpose \textit{Save} to forestall the fusion that would otherwise occur. For example, if

\begin{verbatim}
blur :: Syn a ⇒ Vec a → Vec a
blur v = zipVec (λx y → sqrt (x × y))
  (appVec a (uniVec 0))
  (appVec (uniVec 0) a)
\end{verbatim}
computes the geometric mean of adjacent elements of a vector, then one may compute either \( \text{blur} \circ \text{blur} \) or \( \text{blur} \circ \text{memorise} \circ \text{blur} \) 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 \( \text{Vec} \) are eliminated, while for EDSL this is established by operational reasoning on the behaviour of the type \( \text{Vec} \).

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 (Yaha and Sheard 2000; Eckhardt et al. 2007), and note a propositions-as-types connection between quotation and a modal logic; our type \( Qt \) corresponds to their type \( \Box o \). 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 (Nelson 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; Sync 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 by type annotation, 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 (Yaha and Sheard 2000; Eckhardt et al. 2007), and note a propositions-as-types connection between quotation and a modal logic; our type \( Qt \) corresponds to their type \( \Box o \). 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 (Nelson and Nielson 1992) and partial evaluation (Jones et al. 1993).

Acknowledgements Thanks to Nada Amin, Emil Axelson, 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.