Everything old is new again: Quoted Domain Specific Languages

Shayan Najd  Sam Lindley  Josef Svenningsson  Philip Wadler

The University of Edinburgh  Chalmers University of Technology  The University of Edinburgh
{sh.najd,sam.lindley}@ed.ac.uk  josefs@chalmers.se  wadler@inf.ed.ac.uk

Abstract
We describe a new approach to domain specific languages (DSLs), called Quoted DSLs (QDSLs), that resurrects two old ideas: quotation, from McCarthy’s Lisp of 1960, and the subformula property, from Gentzen’s natural deduction of 1935. Quoted terms allow the DSL to share the syntax and type system of the host language. Normalising quoted terms ensures the subformula property, which guarantees that one can use higher-order 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.

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

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

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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Copyright © 2015 ACM ...... $15.00.
http://dx.doi.org/10.1145/...
The subformula property provides users of the DSL with useful guarantees, such as the following:

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

The first and second are used in this paper, and are key to generating C; while the first and third are used by Cheney et al. (2013) and are key to generating SQL.

The subformula property is closely related to conservativity. A conservativity result expresses that adding a feature to a system of logic, or to a programming language, does not make it more expressive. Consider intuitionistic logic with conjunction; conservativity states that adding implication to this logic proves no additional theorems that can be stated in the original logic. Such a conservativity result is an immediate consequence of the subformula property; since the hypotheses and conjunction of the proof only mention conjunction, any proof, even if it uses implication, can be put into a normal form that only uses conjunction. Equivalently, any lambda calculus term that mentions only pair types in its free variables and result, even if it uses functions, can be put in a normal form that only uses pairs. Such a result is related to the first bullet point above; see Proposition 4.3 in Section 4.

As another example, the third bullet point above corresponds to a standard conservativity result for databases, namely that nested queries are no more expressive than flat queries (Wong 1996). This conservativity result, as implied by the subformula property, 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 property holds only for terms in normal form. Previous work, such as Cheney et al. (2013) uses call-by-name normalisation algorithm that may cause computations to be repeated. Here we present call-by-value and call-by-need normalisation algorithms that guarantee to preserve sharing of computations. We also present a sharpened version of the subformula property, which we apply to characterise the circumstances under which a QDSL may guarantee to generate first-order code.

Good artists copy, great artists steal. — Picasso

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

In theory, an EDSL should also steal the syntax of its host language, but in practice the theft is often only partial. For instance, an EDSL such as Feldspar (Axelsson et al. 2010) or Nikola (Mainland and Morrisett 2010), when embedded in Haskell, can exploit overloading so that arithmetic operations in both languages appear identical, but the same is not true of comparison or conditionals. In QDSL, the syntax of the host and embedded languages is identical. For instance, this paper presents a QDSL variant of Feldspar, again in Haskell, where arithmetic, comparison, and conditionals are all represented by quoted terms, and hence identical to the host.

An EDSL may also steal the normalisation rules of its host language, using evaluation in the host to normalise terms of the target, but again the theft is often only partial. Section 5 compares QDSL and EDSL variants of Feldspar. In the first example, it is indeed the case that the EDSL achieves by evaluation of host terms what the QDSL achieves by normalisation of quoted terms. However, in other cases, the EDSL must perform normalisation of the deep embedding corresponding to how the QDSL normalises quoted terms.

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

— Richard Feynman

The subformula property depends on normalisation, but normalisation may lead to exponential blowup in the size of the normalised code when there are nested conditionals; and hyperexponential blowup in recondite cases involving higher-order functions. We explain how uninterpreted constants allow the user to control where normalisation does and does not occur, while still maintaining the subformula property. Future work is required to consider trade-offs between full normalisation as required for the subformula property and special-purpose normalisation as used in many DSLs; possibly a combination of both will prove fruitful.

Some researchers contend an essential property of a DSL which guarantees target code is that every type-correct term should successfully generate code in the target language. EDSL Feldspar satisfies this property; but neither P-LINQ of Cheney et al. (2013) nor QDSL Feldspar satisfy this property, since the user is required to eyeball quoted code to ensure it mentions only permitted operators. If this is thought too onerous, it is possible to ensure the property with additional preprocessing.

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

The contributions of this paper are:

- To introduce QDSLs as an approach to building DSLs based on quotation, normalisation of quoted terms, and the subformula property by presenting the design of a QDSL variant of Feldspar (Section 2).
- To measure QDSL and EDSL implementations of Feldspar, and show they offer comparable performance (Section 3).
- To present normalisation algorithms for call-by-value and call-by-need that preserve sharing, and to formulate a sharpened version of the subformula property and apply it to characterise when higher-order terms normalise to first-order form (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 tradeoffs 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:

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

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

\[
\begin{align*}
\text{instance } & \text{Rep Int} \\
\text{instance } & \text{Rep Float} \\
\text{instance } & (\text{Rep } a, \text{Rep } b) \Rightarrow \text{Rep } (a, b)
\end{align*}
\]

It is easy to add triples and larger tuples.

### 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 choose that raising zero to a negative power yields zero. Here is an optimised power function represented using QDSL:

\[
\begin{align*}
\text{power} :: & \text{Int} \rightarrow \text{Qt Float} \\
\text{power } n = & \begin{cases} 
\text{if } n < 0 \text{ then } & 0 \\
\text{else if } n \equiv 0 \text{ then } & 1 \\
\text{else if } n \text{ even then } & \begin{cases} 
\text{let } y = \text{div } (n - 1) \text{ in } \text{div } (x \times y) \\
\text{else } & (x \times y) \times (x \times y)
\end{cases}
\end{cases}
\end{align*}
\]

The typed quasi-quoting mechanism of Template Haskell is used to indicate which code executes at which time. Unquoted code executes at generation-time while quoted code executes at run-time. Quoting is indicated by \((\lfloor \ldots \rfloor)\) and unquoting by \(\mathsf{\$} (\cdots)\).

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

\[
\begin{align*}
\lfloor \lambda x \rightarrow & \text{ if } x \equiv 0 \text{ then } 0 \text{ else if } n \text{ even then } \begin{cases} 
\text{let } y = \text{div } (n - 1) \text{ in } \text{div } (x \times y) & \\
\text{else } & (x \times y) \times (x \times y)
\end{cases} \\
\text{else if } n \text{ even then } & \begin{cases} 
\text{let } y = \text{div } (n - 1) \text{ in } \text{div } (x \times y) \\
\text{else } & (x \times y) \times (x \times y)
\end{cases}
\end{align*}
\]

Normalising as described in Section\(^3\) with variables renamed for readability, yields the following:

\[
\begin{align*}
\lfloor \lambda u \rightarrow & \text{ if } u \equiv 0 \text{ then } 0 \text{ else let } v = u \times 1 \text{ in } \\
& \text{let } w = u \times (v \times v) \text{ in } \\
& 1 \div (w \times w)
\end{align*}
\]

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

```c
float prog (float u) {
    float v; 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 \(\text{prog}\); 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 Felspar 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 property.

### 2.3 The subformula property

Gentzen’s subformula property guarantees that any proof can be normalised so that the only formulas that appear within it are subformulas of one of the hypotheses or of the conclusion of the proof. Viewed through the lens of Propositions as Types, Gentzen’s subformula property 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 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 \(a \rightarrow b\) are \(a\) and \(b\), the parts of \((a, b)\) are \(a\) and \(b\), and types \(\text{Int}\) and \(\text{Float}\) have no parts. (See Proposition\(^4\).)

Further, it is easy to adapt the original proof to guarantee a sharpened subformula property: any term can be normalised so that the type of each of its proper subterms is a proper subtype of either the type of one of its free variables (corresponding to hypotheses) or the term itself (corresponding to the conclusion). Here the proper subterms of a term are all subterms save for free variables and the term itself, and the proper subformulas of a type are all subformulas save for the type itself. In the example of the previous subsection, the sharpened subformula property guarantees that after normalisation a closed term of type \(\text{float} \rightarrow \text{float}\) will only have proper subterms of type \(\text{float}\), which is indeed true for the normalised term. (See Proposition\(^6\).)

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

\[
\text{save} :: \text{Rep } a \Rightarrow a \rightarrow a
\]

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 \(\text{save } L\ M\), where \(L\) and \(M\) are arbitrary terms. A use of \(\text{save}\) appears in Section\(^2\). In a context with recursion, we take

\[
\text{fix} :: (a \rightarrow a) \rightarrow a
\]

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 \(\text{Maybe}\) 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 \(\text{power}\) into two functions \(\text{power}\'\) and \(\text{power}\'\), where the first returns \(\text{Nothing}\) in the exceptional case, and the second maps \(\text{Nothing}\) to a suitable value.

The \(\text{Maybe}\) type is a part of the Haskell standard Prelude.
data Maybe a = Nothing | Just a
maybe :: b → (a → b) → Maybe a → b
return :: a → Maybe a
(≫=) :: Maybe a → (a → Maybe b) → Maybe b

Here is the refactored code:

\[
\text{power}': \text{Int} \to \text{Qt} (\text{Float} \to \text{Maybe Float})
\]
\[
\text{power}' n =
\]
\[
\begin{cases}
  || \lambda x . \text{if} \ x \ 0 \ 0 \ \text{then} \ \text{Nothing} \\
  \quad \text{else} \ \text{do} \ y \ 0 \ \text{in} \ $(\text{power}' (-n)) \ x \ return \ (1 / y) ||
\end{cases}
\]
\[
\text{else if} \ n \ 0 \ 0 \ \text{then} \\
\quad || \lambda x . \text{return} \ 1 ||
\]
\[
\text{else if} \ n \ 0 \ n \ \text{then} \\
\quad || \lambda x . \text{do} \ y \ 0 \ $(\text{power}' (n \div 2)) \ x \ return \ (x \times y) ||
\]
\[
\text{else} \\
\quad || \lambda x . \text{return} \ 0 \ (\lambda y . \$(\text{power}' n) \ x) ||
\]
\[
\text{power}'' : \text{Int} \to \text{Qt} (\text{Float} \to \text{Float})
\]
\[
\text{power}'' n =
\]
\[
\begin{cases}
  || \lambda x . \text{maybe} \ 0 \ (\lambda y . \$(\text{power}' n) \ x) ||
\end{cases}
\]

Evaluation and normalisation of power (–6) and power'' (–6) yield identical terms (up to renaming), and hence applying qsdl to these yields identical C code.

The subformula property is key: because the final type of the result does not involve Maybe it is certain that normalisation will remove all its occurrences. Occurrences of do notation are expanded to applications of (≫=), as usual. Rather than taking return, (≫=), 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 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.

while :: (Rep s) ⇒ (s → Bool) → (s → s) → s → s

Rather than using side-effects, while takes three arguments: a predicate over the current state, of type s → Bool; a function from current state to new state, of type s → 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.

\[
\text{for} = ||\lambda n s_0 b \rightarrow \text{snd} (\text{while} (\lambda (i, s) . \ i < n) \left( \begin{array}{c}
  (\lambda (i, s) . \ (i + 1, b \ i s)) \\
  (0, s_0)
\end{array} \right)) ||
\]

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 $$ antiquotes.

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

\[
\text{fib} : \text{Qt} (\text{Int} \to \text{Int})
\]
\[
\text{fib} = \left( ||\lambda n \rightarrow \text{fst} (\$\text{for} \ n \ (0, 1) \ (\lambda (a, b) . \ (b, a + b))) || \right)
\]

2.6 Arrays

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

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

\[
\text{type} \ \text{Arr} \ a = \text{Array} \ \text{Int} \ a
\]

\[
\text{data} \ \text{Vec} a = \text{Vec} \ \text{Int} \ (\text{Int} \to a)
\]

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

\[
\text{instance} \ (\text{Rep} a) \Rightarrow \text{Rep} \ (\text{Arr} a)
\]

For arrays, we assume the following primitive operations.

\[
\text{mkArr} :: (\text{Rep} a) \Rightarrow \text{Int} \to (\text{Int} \to a) \rightarrow \text{Arr} a
\]
\[
\text{lnArr} :: (\text{Rep} a) \Rightarrow \text{Arr} a \rightarrow \text{Int}
\]
\[
\text{ixArr} :: (\text{Rep} a) \Rightarrow \text{Arr} a \rightarrow \text{Int} \rightarrow a
\]

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.

\[
\text{toVec} :: \text{Rep} a \Rightarrow \text{Qt} (\text{Arr} a \rightarrow \text{Vec} a)
\]
\[
\text{toVec} = ||\lambda a \rightarrow \text{Vec} (\text{lnArr} a) (\lambda i . \text{ixArr} a i) ||
\]
\[
\text{fromVec} :: \text{Rep} a \Rightarrow \text{Qt} (\text{Vec} a \rightarrow \text{Arr} a)
\]
\[
\text{fromVec} = ||\lambda (\text{Vec} n g) \rightarrow \text{mkArr} n g ||
\]

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

\[
\text{min} :: \text{Ord} a \Rightarrow \text{Qt} (a \rightarrow a \rightarrow a)
\]
\[
\text{min} = ||\lambda x y . \text{if} \ \ x < y \ \text{then} \ x \ \text{else} \ y ||
\]
The third of these uses the for loop defined in Section 2.3.

Our final function 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

$$[[ \lambda \alpha \rightarrow \text{sqrt} (\$\text{sumVec } v)]]$$

the quoted term normalises to

$$[[ \lambda \alpha \rightarrow \text{sqrt} (\$\text{toVec } v)]]$$

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

$$\text{memorise :: Rep a } \Rightarrow \text{ Qt (Vec a)} \Rightarrow \text{ Vec a}$$

$$\text{memorise } = [[ \lambda \alpha \rightarrow \text{save } (\$\text{sumVec } (\$\text{toVec } v) ) ] ]$$

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

$$\text{blur :: Qt (Vec Float } \Rightarrow \text{ Vec Float) }$$

$$\text{blur } = [[ \lambda \alpha \rightarrow \text{ZipVec } (\lambda x \rightarrow \text{sqrt} (x \times y)) (\$\text{appVec } (\$\text{uniVec } 0) \ a) (\$\text{appVec } a (\$\text{uniVec } 0)) ] ]$$

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

$$[[ \lambda \alpha \rightarrow \text{ZipVec } (\lambda x \rightarrow \text{sqrt} (x \times y)) (\$\text{appVec } (\$\text{uniVec } 0) \ a) (\$\text{appVec } a (\$\text{uniVec } 0)) ] ]$$

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 syntactically identical C code.

4. The subformula property

This section introduces a reduction rules for normalising terms that enforce the subformula property 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 (\(\tau\)), functions (\(\text{A } \rightarrow \text{B}\)),...
Lines of Haskell code

<table>
<thead>
<tr>
<th>Term</th>
<th>QDSL Feldspar</th>
<th>EDSL Feldspar</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>3970</td>
<td>452</td>
<td>2964</td>
</tr>
<tr>
<td>unique</td>
<td>1722</td>
<td>422</td>
<td>1224</td>
</tr>
<tr>
<td>total</td>
<td>5692</td>
<td>2442</td>
<td>4190</td>
</tr>
</tbody>
</table>

Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>QGray</th>
<th>IPBW</th>
<th>FFT</th>
<th>CRC</th>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average array in a sliding window</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPGray</td>
<td>Image Processing (Grayscale)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPBW</td>
<td>Image Processing (Black and White)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Window</td>
<td>Average array in a sliding window</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>QDSL Feldspar</th>
<th>EDSL Feldspar</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>16.36</td>
<td>16.34</td>
<td>16.39</td>
</tr>
<tr>
<td>Run</td>
<td>0.03</td>
<td>0.02</td>
<td>0.71</td>
</tr>
<tr>
<td>Compile</td>
<td>0.39</td>
<td>0.19</td>
<td>0.12</td>
</tr>
<tr>
<td>Run</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>Compile</td>
<td>0.39</td>
<td>0.19</td>
<td>0.12</td>
</tr>
<tr>
<td>Window</td>
<td>15.13</td>
<td>15.13</td>
<td>15.12</td>
</tr>
<tr>
<td>Average</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</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

The only non-trivial proof is for \( \rightarrow_2 \), 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 \( A \rightarrow B \) are itself and the subformulas of \( A \) and \( B \). The proper subformulas of a type are all its subformulas other than the type itself.

The subterms of term are the term itself and its components. For instance, the subterms of \( \lambda x. N \) are itself and the subterms of \( N \) and the subterms of \( L \) 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

\[ c : A_1 \Rightarrow \cdots \Rightarrow A_k \Rightarrow B \]

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

\[ c \ M_1 \cdots M_k \]

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

Terms in normal form satisfy the subformula property.

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

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

Normalisation may lead to an exponential or worse blow up in the size of a term, for instance when there are nested case expressions. The benchmarks in Section 4 do not suffer from blow up, but it may be a problem in some contexts. Normalisation may be controlled by introduction of uninterpreted constants, as in Section 2.5. Further work is needed to understand when complete normalisation is desirable and when it is problematic.

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

**Proposition 4.3** (Sharpened subformula). 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 \( A \) or a proper subformula of a type in \( \Gamma \).

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

The sharpened subformula property says nothing about the types of subterms of constant applications, but such types are immediately apparent by recursive application of the sharpened subformula property. Given a subterm that is a constant application \( c \ M \), where \( c \) has type \( \mathcal{X} \rightarrow B \), then the subterm itself has type \( B \), each subterm \( M_i \) has type \( A_i \), and every proper subterm of \( M \) that is not a free variable of \( M \) 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 \( s \rightarrow \text{Bool} \) and \( s \rightarrow s \), 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 property.

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 \( A \rightarrow B \) has rank \( \min (m + 1, n) \) where \( m \) is the rank of \( A \) and \( n \) is the rank of \( B \), while a term of representable type has rank 0. We say a term is first-order when every subterm is either representable, or is of the form \( \lambda x. N \) where each bound variable and the body is of representable type.

The following characterises translation to a first-order language:

**Proposition 4.4 (First-order).** Consider a term of rank 1, where every free variable has rank 0 and every constant has rank at most 2. Then the term normalises to a term that is first-order.

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

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 the proposition, 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:

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

Here \( \text{Dp} \ a \) 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. As before, \( C \) represents code in C, and type class \( \text{Rep} \) restricts to representable types.

5.1 A first example

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

\[
\text{power} :: \text{Int} \rightarrow \text{Dp} \text{Float} \rightarrow \text{Dp} \text{Float}
\]

\[
\text{power} \ n \ x = \begin{cases} 
\text{if} \ n < 0 \text{ then} \\
\text{else if} \ \text{even} \ n \text{ then} \\
\text{else}
\end{cases}
\]

\[
\begin{align*}
&\text{if} \ n \leq 0 \text{ then} \\
&\begin{cases} 
\text{x ::= } 0 ? (0, 1 / \text{power} (\neg n) \ x) \\
\text{1} \\
\text{y := power} \ (\text{div}\ 2) \ x \ \text{in} \ y \times y \\
\text{else}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
x \times \text{power} \ (\text{div}\ n) \ x
\end{align*}
\]

Type \( Q :: \text{Float} \rightarrow \text{Float} \) in the QDSL variant becomes the type \( \text{Dp} \text{Float} \rightarrow \text{Dp} \text{Float} \) in the EDSL variant, meaning that \( \text{power} \ n \) 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
From the above, it is easy to generate the final C code, which is by default a sharing structure. Applying common-subexpression elimination, for instance via observable sharing, permits recovering the sharing structure.

\[
\begin{align*}
\lambda u \to (u \equiv .0) & \equiv (0, \ 1 / ((u \times ((u \times 1) \times (u \times 1))) \times (u \times ((u \times 1) \times (u \times 1))))
\end{align*}
\]

Applying common-subexpression elimination, for instance via observable sharing, permits recovering the sharing structure.

\[
\begin{align*}
v & \equiv (u \times 1) \\
w & \equiv (u \times (v \times w)) \\
top & \equiv (u \equiv .0) ? (0, 1 / (w \times w))
\end{align*}
\]

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 representation.

- QDSL enables the host and embedded languages to appear identical. In contrast, Haskell, EDSL requires some term forms, such as comparison and conditionals, to differ between the host and embedded languages. Other languages, notably Scala Virtualised [Rompf et al. 2013], may support more general overloading that allows comparison and conditionals to be identical.

- QDSL requires syntax to separate quoted and unquoted terms. In contrast, EDSL permits the 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 with the specific domain. In contrast, EDSL typically develops custom shallow and deep embeddings for each application; a notable exception is the LMS and Delite frameworks for Scala, which provide a deep embedding shared across several disparate DSLs [Sujeeth et al. 2013].

- QDSL yields an unwieldy term that requires normalisation. In contrast, EDSL yields the term in normalised form in this case, though there are other situations where a normaliser is required (see Section 5.2).

- QDSL requires traversing the quoted term to ensure it only mentions permitted identifiers. In contrast, EDSL guarantees that if a term has the right type it will translate to the target. If the requirement to eyeball code to ensure only permitted identifiers are used is considered too onerous, it should be easy to build a preprocessor that checks this property. For example, in Haskell, it is possible to incorporate such a preprocessor using MetaHaskell [Mainland 2012].

- 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.

- Once the deep embedding or the normalised quoted term is produced, generating the domain-specific code 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 \( \Rightarrow \) are similar to before.

**Type**

- `type Opt a
- none :: Undef a \Rightarrow Opt a
- some :: a \to Opt a
- return :: a \to Opt a
- (\Rightarrow) :: Opt a \to (a \to Opt b) \to Opt b
- option :: (Undef a, Undef b) \Rightarrow b \to (a \to b) \to 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.

```plaintext
power' :: Int \to Dp Float \to Opt (Dp Float)
power' a x =
  if n < 0 then
    (x \equiv .0) ? (none,
      do y <- power' \((-n\) x
        return (1 / y))
    else if n == 0 then
      return 1
    else if even n then
      do y <- power' \(n \div 2\) x
        return (y \times y)
    else
      do y <- power' \(n - 1\) x
        return (x \times y)

power'' :: Int \to Dp Float \to Dp Float
power'' n x = option 0 \((\lambda y \to y)\) \((power' n x)\)
```

The term of type `Dp Float` generated by evaluating \( (\lambda x \rightarrow x) \) is large and unscrutable:

\[
(((\text{fst} ((x \equiv 0.0)) \ ? (((\text{False} \ ? \ (\text{True}, \text{False})) \ ? \ (\text{False} \ \text{undef}, \text{undef}))) \ ? \ (\text{True}, \text{undef})) \ ? (((\text{False} \ ? \ (\text{True}, \text{False})) \ ? \ (\text{False} \ \text{undef}, \text{undef}))) \ ? \ (\text{True}, \text{undef})) \ ? (((\text{False} \ ? \ (\text{True}, \text{False})) \ ? \ (\text{False} \ \text{undef}, \text{undef}))) \ ? \ (\text{True}, \text{undef})) \ ? (((\text{False} \ ? \ (\text{True}, \text{False})) \ ? \ (\text{False} \ \text{undef}, \text{undef}))) \ ? \ (\text{True}, \text{undef}) \ ? \ (\text{undef}, \text{undef}))) \ ? \ (\text{true}, \text{undef}))\)\)
\]

Before, evaluating `power` yielded a term essentially in normal form. However, here rewrite rules need to be repeatedly applied, as described in Section 3. After applying these rules, common subexpression elimination yields the same structure as in the previous subsection, from which the same C code is generated.

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.
Here are points of comparison between the two approaches.

- Both QDSL and EDSL can exploit notational conveniences in the host language. The example here exploits Haskell do notation; the embedding of SQL in F# by Cheney et al. (2013) exploits F# sequence notation. For EDSL, exploiting do notation just requires instantiating return and (⇒) correctly. For QDSL, it is also necessary for the translator to recognise and expand do notation and to substitute appropriate instances of return and (⇒).

As this example shows, sometimes both QDSL and EDSL may require normalisation. As mentioned previously, for QDSLs the cost of building a normaliser might be amortised across several applications. In contrast, each EDSL usually has a distinct deep representation and so requires a distinct normaliser.

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.

```
data Dp a where
  LitB :: Bool → Dp Bool
  LitI :: Int → Dp Int
  LitF :: Float → Dp Float
  If :: Dp Bool → Dp a → Dp a → Dp a
  While :: (Dp a → Dp fbool) → (Dp a → Dp a) → Dp a → Dp a
  Pair :: Dp a → Dp b → Dp (a, b)
  Fst :: Rep b ⇒ Dp (a, b) → Dp a
  Snd :: Rep a ⇒ Dp (a, b) → Dp b
  Prim1 :: Rep a ⇒ String → Dp a → Dp b
  Prim2 :: (Rep a, Rep b) ⇒ String → Dp a → Dp b → Dp c
  MkArr :: Dp Int → (Dp Int → Dp a) → Dp (Arr a)
  LnArr :: Rep a ⇒ Dp (Arr a) → Dp Int
  IsArr :: Dp (Arr a) → Dp Int → Dp a
  Save :: Dp a → Dp a
  Let :: Rep a ⇒ Dp a → (Dp a → Dp b) → Dp b
  Variable :: String → Dp a
```

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. 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 IsArr correspond to the array operations in Section 2.6. Save corresponds to save in Section 2.5. 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.

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

Type Internal is a GHC type family. 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.

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

Our representation of a run-time Bool will have 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.

```
true, false :: Dp Bool
true = LitB True
false = LitB False

(?) :: Syn a ⇒ Dp Bool → (a, a) → a
  c ? (t, e) = fromDp (If (c (toDp t)) (toDp e))
  while :: Syn a ⇒ (a → Dp Bool) → (a → a) → a
  while c b i = fromDp (While (c o fromDp)
    (toDp o b o fromDp)
    (toDp i))
```

Numbers are made convenient to manipulate via overloading.

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

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

```
  Prim2 (+) (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.

```
.(==) :: (Syn a, Eq (Internal a)) ⇒ a → a → Dp Bool
a == b = Prim2 (=) (toDp a) (toDp b)
.(<) :: (Syn a, Ord (Internal a)) ⇒ a → a → Dp Bool
a < b = Prim2 (<) (toDp a) (toDp b)
```

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

Here is how to compute the minimum of two values.

```
  minm :: (Syn a, Ord (Internal a)) ⇒ a → a → a
  minm x y = (x <. y) ? (x, y)
```

5.5 Embedding pairs

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

```
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))
```

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 5.5.
We have now developed sufficient machinery to define a for loop in terms of a while loop.

\[
\text{for} :: \text{Syn a} \Rightarrow \text{Dp Int} \to a \to (\text{Dp Int} \to a \to a) \to a \\
\text{for} n \ s_0 \ b = \text{snd} (\text{while} (\lambda (i, s) \to i < n) \\
(\lambda (i, s) \to (i + 1, b \ i s)) \\
(0, s_0))
\]

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.

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 \textit{undef}. It is straightforward to define a type class \textit{UnDef}, where type \(a\) belongs to \textit{UnDef} if it belongs to \textit{Syn} and it has an undefined value.

\[
\begin{align*}
\text{class Syn a} & \Rightarrow \text{UnDef a where} \\
\text{undef} & : a
\end{align*}
\]

\[
\text{instance UnDef (Dp Bool) where} \\
\text{undef} = \text{false}
\]

\[
\text{instance UnDef (Dp Int) where} \\
\text{undef} = \text{0}
\]

\[
\text{instance UnDef (Dp Float) where} \\
\text{undef} = \text{0}
\]

\[
\text{instance (UnDef a, UnDef b) \Rightarrow UnDef (a, b) where} \\
\text{undef} = (\text{undef}, \text{undef})
\]

For example,

\[
(\#) :: \text{Dp Float} \to \text{Dp Float} \to \text{Dp Float}
\]

\[
x \# y \Rightarrow (x + y) ? (\text{undef}, x / y)
\]

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

\textit{Svenningsson 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 2.4.

The deep-and-shallow technique represents deep embedding \textit{Dp} \((a, b)\) by shallow embedding \((\text{Dp a, Dp b})\). Hence, it is tempting to represent \textit{Dp} \((\text{Maybe a})\) by \textit{Maybe} \((\text{Dp a})\), 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, \textit{Svenningsson and Axelsson} [2012] represent values of type \textit{Maybe a} by the type \textit{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}.

\[
\text{data Opt' a} = \text{Opt'} \{ \text{def} :: \text{Dp Bool}, \text{val} :: a \}
\]

\[
\text{instance Syn a} \Rightarrow \text{Syn (Opt' a) where}
\]

\[
\begin{align*}
\text{type Syn a} & \Rightarrow \text{Syn (Opt' a) where} \\
\text{toDp (Opt' a x)} & = (\text{Pair} b (\text{toDp x})) \\
\text{fromDp p} & = \text{Opt' (Fst p) (fromDp (Snd p))}
\end{align*}
\]

\[
\begin{align*}
\text{some'} & :: a \to \text{Opt' a} \\
\text{some'} x & = \text{Opt' true x} \\
\text{none'} & :: \text{UnDef a} \Rightarrow \text{Opt' a} \\
\text{none'} & = \text{Opt' false undef} \\
\text{option'} & :: \text{Syn b} \Rightarrow (a \to b) \Rightarrow \text{Opt' a} \Rightarrow b \\
\text{option'} d f a & = (f (\text{val a}), d)
\end{align*}
\]

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

\[
\begin{align*}
\text{return} & :: a \to \text{Opt' a} \\
\text{return x} & = \text{some' x} \\
\text{(>>=)} & :: (\text{UnDef b} \Rightarrow \text{Opt' a} \Rightarrow (a \to \text{Opt' b}) \to \text{Opt' b}) \\
\text{o >>= g} & = \text{Opt' (def o ? (def (g (val a)), false))} \\
& \quad (\text{def o ? (val (g (val a)), undef))}
\end{align*}
\]

However, this adds type constraint \textit{UnDef b} to the type of \((\gg=)\), 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]). We solve it with a trick due to Persson et al. [2011].

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

\[
\begin{align*}
\text{newtype Opt a} & = \text{O (\text{unO} :: \text{vb. UnDef b} \Rightarrow ((a \to \text{Opt' b}) \to \text{Opt' b})}
\end{align*}
\]

\[
\begin{align*}
\text{instance Monad Opt where} \\
\text{return x} & = \text{O (\lambda g \to g x)} \\
\text{m >>= k} & = \text{O (\lambda g \to \text{unO m} \ (\lambda x \to \text{unO (k x g))}}
\end{align*}
\]

\[
\text{instance UnDef a} \Rightarrow \text{Syn (Opt a) where}
\]

\[
\begin{align*}
\text{type Internal (Opt a) b} & = (\text{Bool, Internal a}) \\
\text{toDp} & = \text{lift o fromDp} \\
\text{fromDp} & = \text{toDp o lower}
\end{align*}
\]

\[
\begin{align*}
\text{lift o} & :: \text{O (\lambda g \to \text{Opt' (def o ? (def (g (val a)), false)) (def o ? (val (g (val a)), undef))}}
\end{align*}
\]

\[
\begin{align*}
\text{lower} & :: \text{UnDef a} \Rightarrow \text{Opt a} \Rightarrow \text{Opt' a} \\
\text{lower m} & = \text{unO m some'} \\
\text{none} & :: \text{UnDef a} \Rightarrow \text{Opt a} \\
\text{none} & = \text{lift none'} \\
\text{some} & :: a \Rightarrow \text{Opt a} \\
\text{some} & = \text{lift (some' a)} \\
\text{option} & :: (\text{UnDef a, Syn b}) \Rightarrow (a \to b) \Rightarrow \text{Opt a} \Rightarrow b \\
\text{option} d f a & = \text{option' d f (lower a)}
\end{align*}
\]

These definitions support the EDSL code presented in Section 5.2.

5.8 Embedding vector

Recall that values of type \textit{Array} are created by construct \textit{ MkArr}, while \textit{lnArr} extracts the length and \textit{izArr} fetches the element at the given index. Corresponding to the deep embedding \textit{Array} is a shallow embedding \textit{Vec}.

\[
\begin{align*}
\text{data Vec a} & = \text{Vec (Dp Int) (Dp Int \to a)}
\end{align*}
\]
instance Syn a ⇒ Syn (Vec a) where
type Internal (Vec a) = Array Int (Internal a)
toDp (Vec n g) = Arr n (toDp o g)
fromDp a = Vec (LnArr a) (fromDp o IzArr a)

instance Functor Vec where
fmap f (Vec n g) = Vec n (f o g)

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
straightforward to make Vec a straightforward to make

\[
\begin{align*}
\text{zipVec} & : (\text{Syn a}, \text{Syn b}) \Rightarrow (a \rightarrow b) \rightarrow \text{Vec a} \rightarrow \text{Vec b} \rightarrow \text{Vec c} \\
\text{zipVec} f \ (\text{Vec m g}) \ (\text{Vec n h}) &= \text{Vec} \ (m \cdot \text{minim} \ n) \ (\lambda i \rightarrow f \ (g \ i) \ (h \ i)) \\
\text{sumVec} & : (\text{Syn a}, \text{Num a}) \Rightarrow \text{Vec a} \rightarrow a \\
\text{sumVec} \ (\text{Vec n g}) &= \text{for} \ n \ 0 \ (\lambda i \rightarrow x + g \ i) \\
\text{dotVec} & : (\text{Syn a}, \text{Num a}) \Rightarrow \text{Vec a} \rightarrow \text{Vec a} \rightarrow a \\
\text{dotVec} \ a \ v \ w &= \text{sumVec} \ (\text{zipVec} \ (\times) \ a \ v) \\
\text{normVec} & : \text{Vec} \ (\text{Dp Float}) \rightarrow \text{Dp Float} \\
\text{normVec} \ v &= \text{sqrt} \ (\text{dotVec} \ v \ v)
\end{align*}
\]

Invoking

\[
[\text{normVec} \circ \text{toVec}]
\]

generates C code to normalise a vector. If we used a top-level
function of type \((\text{Syn a}, \text{Syn b}) \Rightarrow (a \rightarrow b) \rightarrow C\), then it would
insert the \text{toVec} coercion automatically.

This style of definition again provides fusion. For instance:

\[
\begin{align*}
\text{dotVec} \ (\text{Vec m g}) \ (\text{Vec n h}) &= \text{sumVec} \ (\text{zipVec} \ (\times) \ (\text{Vec m g}) \ (\text{Vec n h})) \\
\text{sumVec} \ (\text{Vec} \ (m \cdot \text{minim} \ n) \ (\lambda i \rightarrow g \ i \ i \ i) \ (\lambda i \rightarrow x + g \ i \ i \ i)) &= \text{for} \ (m \cdot \text{minim} \ n) \ (\lambda i \rightarrow x + g \ i \ i \ i) \\
\text{normVec} \ v &= \text{sqrt} \ (\text{dotVec} \ v \ v)
\end{align*}
\]

Indeed, we can see that by construction that whenever we combine
the beginning of Section 3, and lift most of its restrictions.

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
\text{toVec} and \text{fromVec} are not required. Invoking \text{edsl normVec produces the same C code as in Section 6}.

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

\[
\begin{align*}
\text{memorise} & : \text{Syn a} \Rightarrow \text{Vec a} \rightarrow \text{Vec a} \\
\text{memorise} &= \text{fromDp} \circ \text{Save} \circ \text{toDp}
\end{align*}
\]

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

\[
\begin{align*}
\text{blur} & : \text{Syn a} \Rightarrow \text{Vec a} \rightarrow \text{Vec a} \\
\text{blur} \ v &= \text{zipVec} \ (\lambda x \ y \rightarrow \text{sqrt} \ (x \times y)) \\
& \quad (\text{app} \ \text{Vec a} \ (\text{uniVec} \ 0)) \\
& \quad (\text{app} \ (\text{uniVec} \ 0) \ a)
\end{align*}
\]

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.

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 property 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. 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).

This paper uses Haskell, which has been widely used for ED-

SLs (Hudak 1997). We constrain QDSL with an EDSL technique

that combines deep and shallow embedding, as described by

Svenningsson and Axelsen (2012), and as used in several Haskell ED-

SLs including Feldspar (Axelsen et al. 2010), Obsidian (Svens-

son et al. 2011), Nikola (Mainland and Morrisett 2010), Hydra

(Giorgidze and Nilsson 2011), and Meta-Repa (Ankner and Sven-

ningsson 2013).

O’Donnell (1993) identified loss of sharing in the context of em-

bedded circuit descriptions. Claessen and Sands (1999) extended

Haskell to support observable sharing. Gill (2009) proposes library

features that support sharing without need to extend the language.

A proposition-as-types principle for quotation as a modal logic

was proposed by Davies and Pfenning (2001). As they note, their

technique has close connections to two-level languages (Nielson

and Nielson 2005) and partial evaluation (Jones et al. 1993).

Other approaches to DSL that make use of quotation include

C# and F# versions of LINQ (Meijer et al. 2006; Syme 2006) and

Scala Lightweight Modular Staging (LMS) (Rompf and Odersky

2010). Scala LMS exploits techniques found in both QDSL (quo-

tation 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 (Sujeth et al. 2013).

The underlying idea for QDSLs was established for F# LINQ

by Cheney et al. (2013), and extended 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.

7. Conclusion

A good idea can be much better than a new one.
― Gerard Berry

We have compared QDSLs and EDSLs, arguing that QDSLs

offer competing expressiveness and efficiency.

The subformula property may have applications in DSLs other

that 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 \text{Dp} with constructs for type \text{Maybe}. So long as type

\text{Maybe} does not appear in the input or output of the program, a

normaliser that ensures the subformula property could guarantee

that C code for such constructs need never be generated.

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 quasi-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 \text{Qt} to \text{Dp} described at

the beginning of Section 6 and lift most of its restrictions.

These forty years now I’ve been speaking in prose without

knowing it!
― Molière

Like Molière’s Monsieur Jourdain, many of us have used QD-

SLs for years, if not by that name. DSL via quotation lies at the
References


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


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


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