

Everything old is new again: Quoted Domain Specific Languages

Shayan Najd

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

Sam Lindley

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

Josef Svenningsson

Chalmers University of Technology
josefs@chalmers.se

Philip Wadler

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

Abstract

Fashions come, go, return. We describe a new approach to domain specific languages (DSLs), called Quoted DSLs (QDSLs), that resurrects two old ideas: quotation, from McCarthy's Lisp of 1960, and the subformula property, from Gentzen's natural deduction of 1935. Quoted terms allow the domain specific language to share the syntax and type system of the host language. Normalising quoted terms ensures the subformula property, which provides strong guarantees, e.g., that one can use higher-order or nested code in the source while guaranteeing first-order or flat code in the target, or using types to guide fusion. We give three examples of QDSLs: QFeldspar (a variant of Feldspar), P-LINQ for F#, and some uses of Scala LMS. We also provide a comparison between QDSLs and traditional Embedded DSL (EDSLs).

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

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

1. Introduction

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

Implementing domain-specific languages (DSLs) via quotation is one of the oldest ideas in computing, going back at least to McCarthy's Lisp, which was introduced in 1960 and had macros as early

as 1963. Today, a more fashionable technique is 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 a new approach which we dub Quoted DSLs (QDSLs). A key feature of QDSLs is the use of normalisation to ensure the subformula property, first proposed by Gentzen in 1935.

Imitation is the sincerest of flattery.

— Charles Caleb Colton

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

Here we test that conjecture by reimplementing the EDSL Feldspar Axelsson et al. (2010) as a QDSL. We describe the key features of the design, and show that the performance of the two versions is comparable. We argue that, from the user's point of view, the QDSL approach may sometimes offer a considerable simplification as compared to the EDSL approach. To back that claim, we describe the EDSL approach to Feldspar for purposes of comparison. The QDSL description occupies TODO:M pages, while the EDSL description requires TODO:N pages.

We also claim that Lightweight Modular in Staging (LMS) as developed by Scala has much in common with QDSL: it often uses a type-based form of quotation, and some DSLs implemented with LMS exploit normalisation of quoted terms using smart constructors. LMS is a flexible library offering a range of approaches to building DSLs, only some of which make use of type-based quotation or normalisation via smart-constructors; so our claim is that some LMS implementations use QDSL techniques, not that QDSL subsumes LMS. We suggest that LMS implementations of DSLs that use normalisation may benefit from the subformula property.

[TODO: work out which specific LMS DSLs to cite. Scala-to-SQL is one, what are the others?]

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

— Gerhard Gentzen

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.

ICFP 2015, August 31–September 2, 2015, Vancouver, Canada.
Copyright © 2015 ACM \$15.00.
<http://dx.doi.org/10.1145/...>

Our approach exploits the fact that normalised terms satisfy the subformula property, first introduced in the context of natural deduction by ?, improved by ?, and applicable to lambda calculus via the principle of Propositions as Types.

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

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

We thus give modern application to a theorem four-fifths of a century old.

The subformula property is akin to a generalised conservativity result. For instance, the third bullet point above corresponds to a standard conservativity result for databases, namely that nested queries are no more expressive than flat queries. [TODO: reference for conservativity result.]

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

Good artists copy, great artists steal.

— Picasso

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

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

In theory, an EDSL also steals the normalisation rules of its host language, by using evaluation in the host to normalise terms of the target. In Section 6 we give several examples comparing our QDSL and EDSL versions of Feldspar. In the first of these, it is indeed the case that the EDSL achieves by evaluation of host terms what the QDSL achieves by normalisation of quoted terms. However, in other cases, the EDSL must perform normalisation of the deep embedding corresponding to what the QDSL achieves by normalisation of quoted terms.

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

— Richard Feynman

The subformula property depends on normalisation, but normalisation may lead to an exponential blowup in the size of the normalised code. In particular, this occurs when there are nested conditional or case statements. We explain how the QDSL technique

can offer the user control over where normalisation does and does not occur, while still maintaining the subformula property.

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

TODO: some quotation suitable for contributions (or summary)

The contributions of this paper are:

- To introduce QDSLs as an approach to building DSLs based on quotation, normalisation of quoted terms, and the subformula property by presenting the design of a QDSL variant of Feldspar. (Section 2.)
- To compare QDSL and EDSL implementations of Feldspar, and show that they are of comparable length and offer comparable performance. (Section ??.)
- To explain the role of the subformula property in formulating DSLs, and to describe a normalisation algorithm suitable for call-by-value or call-by-need, which ensures the subformula property while not losing sharing of quoted terms. (Section 4.)
- To review the F# implementation of language-integrated query (Cheney et al. 2013) and the Scala LMS implementations of query and [TODO: what else?], and argue that these are instances of QDSL. (Section 5.)
- To argue that, from the user’s point of view, the QDSL variant of Feldspar is conceptually easier to understand than the EDSL variant of Feldspar, by a detailed comparison of the two (Section 6.)

Section 7 describes related work, and Section 8 concludes.

2. A QDSL variant of Feldspar

Feldspar is an EDSL for writing signal-processing software, that generates code in C (Axelsson et al. 2010). We present a variant, QFeldspar, that follows the structure of the previous design closely, but using the methods of QDSL rather than EDSL. We make a detailed comparison of the QDSL and EDSL designs in Section 6.

2.1 The top level

In QFeldspar, our goal is to translate a quoted term to C code, so we also assume a type *C* that represents code in C. The top-level function of QFeldspar has the type:

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

which generates a `main` function that takes an argument of type *a* and returns a result of type *b*.

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

```
instance Rep Int
instance Rep Float
instance (Rep a, Rep b) => Rep (a, b)
```

It is easy to add triples and larger tuples.

2.2 An introductory example

Let's begin by considering the “hello world” of program generation, the power function, raising a float to an arbitrary integer. We assume a type *Qt a* to represent a term of type *a*, its *quoted* representation. Since division by zero is undefined, we arbitrarily choose that raising zero to a negative power yields zero. Here is the power function represented using QDSL:

```
power :: Int → Qt (Float → Float)
power n =
  if n < 0 then
    [|λx → if x == 0 then 0
      else 1 / ($$(power (-n)) x)|]
  else if n == 0 then
    [|λx → 1|]
  else if even n then
    [|λx → $$$qr ($$(power (n div 2)) x)|]
  else
    [|λx → x × ($$(power (n - 1)) x)|]
qr :: Qt (Float → Float)
qr = [|λy → y × y|]
```

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

Evaluating *power* (-6) yields the following:

```
[|λx → if x == 0 then 0 else
  1 / (λx → (λy → y × y)
    ((λx → (x × ((λx → (λy → y × y)
      ((λx → (x × ((λx → 1) x))) x))) x))) x)|]
```

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

```
[|λu → if u == 0 then 0 else
  let v = u × 1 in
  let w = u × (v × v) in
  1 / (w × 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 Gentzen's subformula property. From the normalised term it is easy to generate the desired C code:

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

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

Depending on your point of view, quotation in this form of QDSL is either desirable, because it makes manifest the staging, or undesirable because it is too noisy. We return to this point in Section ?? . QDSL enables us to “steal” the entire syntax of the host language for our DSL. The EDSL approach can use the same syntax for arithmetic operators, but must use a different syntax for equality tests and conditionals, as we will see in Section 6.

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

2.3 The subformula property

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

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

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

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

2.4 Maybe

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

The *Maybe* type is a part of the standard prelude.

```
data Maybe a = Nothing | Just a
```

```
return      :: a → Maybe a
```

```
return      = Just
```

```
(>=)       :: Maybe a → (a → Maybe b) → Maybe b
```

```
m >= k      = case m of
  Nothing → Nothing
  Just x  → k x
```

```
maybe     :: b → (a → b) → Maybe a → b
```

```
maybe x g m = case m of
  Nothing → x
  Just y  → g y
```

Here is the refactored code.

```

power' :: Int → Qt (Float → Maybe Float)
power' n =
  if n < 0 then
    [[λx → if x == 0 then Nothing
      else do y ← $(power' (-n)) x
      return (1 / y)]]
  else if n == 0 then
    [[λx → return 1]]
  else if even n then
    [[λx → do y ← $(power' (n div 2)) x
      return ($sqr y)]]
  else
    [[λx → do y ← $(power' (n - 1)) x
      return (x × y)]]
power'' :: Int → Qt (Float → Float)
power'' n = [[λx → maybe 0 (λy → y) ($ (power' n) x)]]

```

Here *sqr* is as before. Evaluation and normalisation of *power* (−6) and *power''* (−6) yield identical terms (up to renaming), and hence applying *qdsl* to these yields identical C code.

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

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

2.5 While

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

$$while :: (Rep\ s) \Rightarrow Qt\ ((s \rightarrow Bool) \rightarrow (s \rightarrow s) \rightarrow s \rightarrow s)$$

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

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

```

for :: (Rep\ s) \Rightarrow Qt\ (Int \rightarrow s \rightarrow (Int \rightarrow s \rightarrow s) \rightarrow s)
for = [[λn s_0 b \rightarrow snd (while (λ(i, s) \rightarrow i < n)
  (λ(i, s) \rightarrow (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.

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

```

fib :: Qt\ (Int \rightarrow Int)
fib = [[λn \rightarrow $for n (λ(a, b) \rightarrow (b, a + b)) (0, 1) ]]

```

Again, the subformula property plays a key role. As explained in Section 2.3, primitives of the language to be compiled, such as (\times) and *while*, are treated as free variables of a given arity. As will

be explained in Section ??, we can ensure that after normalisation every occurrence of *while* has the form

$$while\ (\lambda s \rightarrow \dots)\ (\lambda s \rightarrow \dots)\ (\dots)$$

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

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

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

2.6 Arrays

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

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

```

type Arr a = Array Int a
data Vec a = Vec Int (Int \rightarrow a)

```

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

```
instance (Rep\ a) \Rightarrow Rep\ (Arr a)
```

For arrays, we assume the following primitive operations.

```

makeArr :: (Rep\ a) \Rightarrow Int \rightarrow (Int \rightarrow a) \rightarrow Arr a
lenArr  :: (Rep\ a) \Rightarrow Arr a \rightarrow Int
ixArr   :: (Rep\ a) \Rightarrow Arr a \rightarrow 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.

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

```

toArr :: Qt\ (Vec a \rightarrow Arr a)
toArr = [[λ (Vec n g) \rightarrow makeArr n (λx \rightarrow g x)]]
toVec :: Qt\ (Arr a \rightarrow Vec a)
toVec = [[λa \rightarrow Vec (lenArr a) (λi \rightarrow ixArr a i)]]

```

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.

```

zipVec :: Qt\ ((a \rightarrow b \rightarrow c) \rightarrow Vec a \rightarrow Vec b \rightarrow Vec c)
zipVec = [[λf (Vec m g) (Vec n h) \rightarrow
  Vec (m 'min' n) (λi \rightarrow f (g i) (h i))]]

```

```

sumVec :: (Rep a, Num a) => Qt (Vec a -> a)
sumVec = [[λ( Vec n g) -> $$for n 0 (λi x -> x + g i)]]
dotVec :: (Rep a, Num a) => Qt (Vec a -> Vec a -> a)
dotVec = [[λu v -> $$sumVec ($$zipVec (×) u v)]]
normVec :: Qt (Vec Float -> Float)
normVec =

```

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

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

```

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

```

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

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

```

memorise :: Syn a => Qt (Vec a -> Vec a)
memorise = [[λtoVec -> toArr]]

```

For example, if

```
blur :: Syn a => Vec a -> Vec a
```

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

```
[[λblur -> blur]] or [[λblur -> memorise -> blur]]
```

with different trade-offs between recomputation and memory usage. Strong guarantees for fusion in combination with *memorise* gives the programmer a simple interface which provides powerful optimisation combined with fine control over memory usage.

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

3. Implementation

The original Feldspar generates values of an algebraic type (called *Dp a* in Section 6), with constructs that represent *while* and manifest arrays similar to those above. A backend then compiles values of type *Exp a* to C code. QFeldspar provides a transformer from *Qt a* to *Exp a*, and shares the Feldspar backend.

The transformer from *Qt* to *Exp* performs the following steps.

- In any context where a constant *c* is not fully applied, it replaces *c* with $\lambda \bar{x}..$ It replaces identifiers connected to the type *Maybe*, such as *return*, $(\gg=)$, and *maybe*, by their definitions.

- It normalises the term to ensure the subformula property, using the rules of Section 4. The normaliser does not support all Haskell data types, but does support tuples, and the types *Maybe* and *Vec*.
- It traverses the term, converting *Qt* to *Dp*. It checks that only permitted primitives appear in *Qt*, and translates these to their corresponding representation in *Dp*. Permitted primitives include: $(=)$, $(<)$, $(+)$, (\times) , and similar, plus *while*, *makeArr*, *lenArr*, and *ixArr*.

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

We measured the behaviour of five benchmark programs.

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

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

Measurements were done on a PC with a quad-core Intel i7-2640M CPU running at 2.80 GHz and 3.7 GiB of RAM, with GHC Version 7.8.3 and GCC version 4.8.2, running on Ubuntu 14.04 (64-bit).

4. The subformula property

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

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

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

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

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

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

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

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

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

Every term has a normal form.

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

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

The *subformulas* of a type are the type itself and its components. For instance, the subformulas of $A \rightarrow B$ 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 $L M$ are itself and the subterms of L and M .

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

$$c : A_1 \rightarrow \dots \rightarrow A_k \rightarrow B$$

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

$$cM_1 \dots M_k$$

and its subterms are itself and M_1, \dots, M_k (but not $cM_1 \dots M_j$ for $j < k$). Free variables are equivalent to constants of arity zero.

Terms in normal form satisfy the subformula property.

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

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

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

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

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

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

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

In Section 2, an important property is that every top-level term passed to *qdsl* is suitable for translation to C after normalisation. Here we are interested in C as a *first-order* language. The exact property required is somewhat subtle. One might at first guess the required property is that every subterm is representable, in the sense introduced in Section ??, 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 $s \rightarrow Bool$ and

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

Figure 2. Types, Terms, and Values

Phase 1 (let-insertion)

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

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

Phase 2 (symbolic evaluation)

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

$(\kappa.\text{let})$	$G[\text{let } x = P \text{ in } N]$	\mapsto_2	$\text{let } x = P \text{ in } G[N], \quad x \notin FV(G)$
$(\kappa.\text{case})$	$G[\text{case } z \text{ of } \{\text{inl } x. M; \text{inr } y. N\}]$	\mapsto_2	$\text{case } z \text{ of } \{\text{inl } x. G[M]; \text{inr } y. G[N]\}, \quad x, y \notin FV(G)$
$(\beta.\rightarrow)$	$(\lambda x. N) V$	\mapsto_2	$N[x := V]$
$(\beta.\times_1)$	$\text{fst } (V, W)$	\mapsto_2	V
$(\beta.\times_2)$	$\text{snd } (V, W)$	\mapsto_2	W
$(\beta.+_1)$	$\text{case } (\text{inl } V) \text{ of } \{\text{inl } x. M; \text{inr } y. N\}$	\mapsto_2	$M[x := V]$
$(\beta.+_2)$	$\text{case } (\text{inr } W) \text{ of } \{\text{inl } x. M; \text{inr } y. N\}$	\mapsto_2	$N[y := W]$
$(\beta.\text{let})$	$\text{let } x = V \text{ in } N$	\mapsto_2	$N[x := V]$

Phase 3 (garbage collection)

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

Figure 3. Normalisation rules

$s \rightarrow s$, where the state s is representable. Fortunately, the exact property required is not hard to formulate in a general way, and is easy to ensure by applying the sharpened subformula property.

We introduce a variant of the usual notion of *rank* of a type, with respect to a notion of representability. A term of type $A \rightarrow B$ 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.

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

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

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

5. Other examples of QDSLs

5.1 F# P-LINQ

5.2 Scala LMS

6. A comparison of QDSL and EDSL

6.1 First example

Let's begin by considering 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.

```
power :: Int -> Float -> Float
power n x =
  if n < 0 then
    if x == 0 then 0 else 1 / power (-n) x
  else if n == 0 then
    1
  else if even n then
    sqr (power (n div 2) x)
  else
    x * power (n - 1) x
sqr :: Float -> Float
sqr x = x * x
```

Our goal is to generate code in the programming language C. For example, instantiating *power* to (-6) should result in the following:

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

6.1.1 CDSL

For CDSL, we assume a type $Dp \ a$ to represent a term of type a , its *deep* representation. Function

$$\text{cdsl} :: (\text{Type } a, \text{Type } b) \Rightarrow (Dp \ a \rightarrow Dp \ b) \rightarrow C$$

generates a `main` function corresponding to its argument, where C is a type that represents C code. Here is a solution to our problem using CDSL.

```
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
    sqr (power (n div 2) x)
  else
    x × power (n - 1) x
sqr :: Dp Float → Dp Float
sqr y = y × y
```

Invoking `cdsl (power (-6))` generates the C code above.

Type $\text{Float} \rightarrow \text{Float}$ in the original becomes $\text{Dp Float} \rightarrow \text{Dp Float}$ in the CDSL solution, meaning that `power n` accepts a representation of the argument and returns a representation of that argument raised to the n 'th power.

In CDSL, the body of the code remains almost—but not quite!—identical to the original. Clever encoding tricks, explained later, permit declarations, function calls, arithmetic operations, and numbers to appear the same whether they are to be executed at generation-time or run-time. However, as explained later, comparison and conditionals appear differently depending on whether they are to be executed at generation-time or run-time, using $M == N$ and $\text{if } L \text{ then } M \text{ else } N$ for the former but $M .==. N$ and $L ? (M, N)$ for the latter.

Evaluating `power (-6)` yields the following:

$$\lambda u \rightarrow (u .==. 0) ? (0, 1 / ((u \times ((u \times 1) \times (u \times 1))) \times (u \times ((u \times 1) \times (u \times 1)))))$$

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

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

It is easy to generate the final C code from this structure.

6.1.2 QDSL

By contrast, for QDSL, we assume a type $Qt\ a$ to represent a term of type a , its *quoted* representation. Function

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

[TODO: Should be as follows]

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

generates a `main` function corresponding to its argument, where C is as before. Here is a solution to our problem using QDSL.

```
power :: Int → Qt (Float → Float)
power n =
  if n < 0 then
    [||λx → if x == 0 then 0 else 1 / ($$(power (-n)) x)||]
  else if n == 0 then
    [||λx → 1||]
  else if even n then
    [||λx → $$sqr ($$(power (n div 2)) x)||]
  else
    [||λx → x × ($$(power (n - 1)) x)||]
sqr :: Qt (Float → Float)
sqr = [||λy → y × y||]
```

Invoking `qdsl (power (-6))` generates the C code above.

Type $\text{Float} \rightarrow \text{Float}$ in the original becomes $Qt\ (\text{Float} \rightarrow \text{Float})$ in the QDSL solution, meaning that `power n` returns a quotation of a function that accepts an argument and returns that argument raised to the n 'th power.

In QDSL, the body of the code changes more substantially. The typed quasi-quoting mechanism of Template Haskell is used to indicate which code executes at which time. Unquoted code executes at generation-time while quoted code executes at run-time. Quoting is indicated by `[||...||]` and unquoting by `$(...)`. Here, by the mechanism of quoting, without any need for tricks, the syntax for code executed at both generation-time and run-time is identical for all constructs, including comparison and conditionals.

Evaluating `power (-6)` yields the following:

$$\begin{aligned} & [||\lambda x \rightarrow \text{if } x == 0 \text{ then } 0 \text{ else} \\ & \quad 1 / (\lambda x \rightarrow (\lambda y \rightarrow y \times y) \\ & \quad \quad ((\lambda x \rightarrow (x \times ((\lambda x \rightarrow (\lambda y \rightarrow y \times y) \\ & \quad \quad \quad ((\lambda x \rightarrow (x \times ((\lambda x \rightarrow 1) x))) x))) x))) x ||] \end{aligned}$$

Normalising, with variables renamed for readability, yields code equivalent to the following:

$$\begin{aligned} & [||\lambda u \rightarrow \text{if } u == 0 \text{ then } 0 \text{ else} \\ & \quad \text{let } v = u \times 1 \text{ in} \\ & \quad \text{let } w = u \times (v \times v) \text{ in} \\ & \quad 1 / (w \times w) ||] \end{aligned}$$

It is easy to generate the final C code from the normalised term.

6.1.3 Comparison

Here are some points of comparison between the two approaches.

- A function $a \rightarrow b$ is embedded in CDSL as $\text{Dp } a \rightarrow \text{Dp } b$, a function between representations, and in QDSL as $Qt\ (a \rightarrow b)$, a representation of a function.
- CDSL requires some term forms, such as comparison and conditionals, to differ between the host and embedded languages. In contrast, QDSL enables the host and embedded languages to appear identical.
- CDSL permits the host and embedded languages to intermingle seamlessly. In contrast, QDSL requires syntax to separate quoted and unquoted terms, which (depending on your point of view) may be considered as an 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.
- CDSL typically develops custom shallow and deep embeddings for each application, although these may follow a fairly standard pattern (as we review in Section ??). In contrast, QDSL may share the same representation for quoted terms across a range of applications; the quoted language is the host language, and does not vary with the specific domain.
- CDSL loses sharing, which must later be recovered by either common subexpression elimination or applying a technique such as observable sharing. In contrast, QDSL preserves sharing throughout.
- CDSL yields the term in normalised form in this case, though there are other situations where a normaliser is required (see Section 6.2.) In contrast, QDSL yields an unwieldy term that requires normalisation. However, just as a single representation of QDSL terms suffices across many applications, so does a single normaliser—it can be built once and reused many times.

- ## 5.2 Second example

```

power' :: Int → Float → Maybe Float
power' n x =
  if n < 0 then
    if x == 0 then Nothing else do y ← pow
  else if n == 0 then
    return 1
  else if even n then
    do y ← power' (n div 2) x; return (sqr y)
  else
    do y ← power' (n - 1) x; return (x × y)
power'' :: Int → Float → Float
power'' n x = maybe 0 (λy → y) (power' n x)

```

```

data Maybe a = Nothing | Just a
return :: a → Maybe a
(≫) :: Maybe a → (a → Maybe b) → Maybe b
maybe :: b → (a → b) → Maybe a → b

```

In this case, the refactored function is arguably clumsier than the original, but clearly it is desirable to support this form of refactoring in general.

6.2.1 CDSL

```

power' :: Int → Dp Float → Opt (Dp Float)
power' n x =
  if n < 0 then
    (x .==. 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 (sqr y)
  else
    do y ← power' (n − 1) x; return (x × y)
power''      :: Int → Dp Float → Dp Float
power'' n x = option 0 (λy → y) (power' n x)

```

$$\begin{aligned} \text{none} &:: \text{Opt } a \\ \text{return} &:: a \rightarrow \text{Opt } a \\ (\gg) &:: \text{Opt } a \rightarrow (a \rightarrow \text{Opt } b) \rightarrow \text{Opt } b \\ \text{option} &:: (\text{Syn } a, \text{Syn } b) \Rightarrow b \rightarrow (a \rightarrow b) \rightarrow \text{Opt } a \rightarrow b \end{aligned}$$

as a monad, enabling the **do** notation, just as for *Maybe* above. Invoking *cdsl* (*power''* (−6)) generates the same C code as the previous example.

$$\begin{aligned} &(((fst((x = (0.0))) ? (((False) ? ((True), (False))), ((False) ? (undef, \\ undef))), ((True), ((1.0) / ((x \times (x \times (1.0)) \times (x \times (1.0)))) \times (x \times ((x \\ (1.0)) \times (x \times (1.0)))))))) ? ((True), (False))) ? (((fst((x = (0.0))) ? \\ (((False) ? ((True), (False))), ((False) ? (undef, undef))), ((True), ((1.0) \\ ((x \times ((x \times (1.0)) \times (x \times (1.0)))) \times (x \times ((x \times (1.0)) \times (x \times (1.0)))))), \\ ((snd((x = (0.0))) ? (((False) ? ((True), (False))), ((False) ? (undef, \\ undef))), ((True), ((1.0) / ((x \times (x \times (1.0)) \times (x \times (1.0)))) \times (x \times ((x \\ (1.0)) \times (x \times (1.0))))))))), undef)), (0.0))) \end{aligned}$$
$$\begin{array}{ll}
fst\ (M, N) & \rightsquigarrow M \\
snd\ (M, N) & \rightsquigarrow N \\
fst\ (L\ ?\ (M, N)) & \rightsquigarrow L\ ?\ (fst\ M, fst\ N) \\
snd\ (L\ ?\ (M, N)) & \rightsquigarrow L\ ?\ (snd\ M, snd\ N) \\
True\ ?\ (M, N) & \rightsquigarrow M \\
False\ ?\ (M, N) & \rightsquigarrow N \\
(M, N)\ ?\ (P, Q) & \rightsquigarrow L\ ?\ (M\ ?\ (P, Q))\ ?\ (N\ ?\ (P, Q)) \\
L\ ?\ (M, N) & \rightsquigarrow L\ ?\ (M[L := True], N[L := False])
\end{array}$$

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

6.2.2 QDSL

```

power' :: Int → Qt (Float → Maybe Float)
power' n =
  if n < 0 then
    [|λx → if x == 0 then Nothing else
      do y ← $$ (power' (-n)) x; return (1 / y)|]
  else if n == 0 then
    [|λx → return 1|]
  else if even n then
    [|λx → do y ← $$ (power' (n div 2)) x; return ($$sqr y)|]
  else
    [|λx → do y ← $$ (power' (n - 1)) x; return (x × y)|]
power'' :: Int → Qt (Float → Float)
power'' n = [|λx → maybe 0 (λy → y) ($$ (power' n) x)|]

```

Here *sqr* is as before, and *Nothing*, *return*, (\gg), and *maybe* are as in the Haskell prelude, and provided for use in quoted terms by the QDSL library.

Evaluating *power''* (−6) yields a term of similar complexity to the term yielded by the CDSL. Normalisation by the rules discussed in Section 4 reduces the term to the same form as before, which in turn generates the same C as before.

6.2.3 Comparison

Here are further points of comparison between the two approaches.

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

We now review the usual approach to embedding a DSL into a host language by combining deep and shallow embedding. As a running example, we will use MiniFeldspar, an embedded DSL for generating signal processing software in C. Much of this section reprises ?.

6.3 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 Bool) → (Dp a → Dp a) → Dp a → Dp a
Pair   :: Dp a → Dp b → Dp (a, b)
Fst    :: Dp (a, b) → Dp a
Snd    :: Dp (a, b) → Dp b
Prim1  :: String → (a → b) → Dp a → Dp b
Prim2  :: String → (a → b → c) → Dp a → Dp b → Dp c
Arr     :: Dp Int → (Dp Int → Dp a) → Dp (Array Int a)
ArrLen  :: Dp (Array Int a) → Dp Int
ArrIx   :: Dp (Array Int a) → Dp Int → Dp a
Variable :: String → Dp a
Value  :: a → Dp a

```

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

Our CDSL has boolean, integer, and floating point literals, conditionals, while loops, pairs, primitives, arrays, and special-purpose constructs for variables and values. Constructs *LitB*, *LitI*, *LitF* build literals. Construct *If* builds a conditional. Construct *While* may require explanation. Rather than using side-effects, the while loop takes three arguments, a function from current state *a* to a boolean, and a function from current state *a* to new state *a*, and initial state *a*, and returns final state *a*. Constructs *Pair*, *Fst*, and *Snd* build pairs and extract the first and second component. Constructs *Prim1* and *Prim2* represent primitive operations, the string is

the name of the operation (used in printing or to generate C) and the function argument computes the primitive (used in evaluation). Construct *ArrIx* creates a new array from a length and a body that computes the array element for each index, construct *ArrLen* extracts the length from an array, and construct *ArrIx* fetches the element at a given index. Construct *Variable* is used in printing and in generating C, construct *Value* is used in the evaluator.

The exact semantics is given by *eval*. It is a strict language, so we define an infix strict application operator ($\langle * \rangle$).

```

(<*>)      :: (a → b) → a → b
f <*> x     = seq x (f x)

eval       :: Dp a → a
eval (LitI i)   = i
eval (LitF x)   = x
eval (LitB b)   = b
eval (If c t e) = if eval c then eval t else eval e
eval (While c b i) = evalWhile (evalFun c) (evalFun b) <*> eval i
eval (Pair a b)  = (,) <*> eval a <*> eval b
eval (Fst p)     = fst <*> eval p
eval (Snd p)     = snd <*> eval p
eval (Prim1 _ f a) = f <*> eval a
eval (Prim2 _ f a b) = f <*> eval a <*> eval b
eval (Arr n g)    = array (0, n') [(i, eval (g (LitI i))) | i <- [0..n']]
                where n' = eval n - 1
eval (ArrLen a)   = u - l + 1 where (l, u) = bounds (eval a)
eval (ArrIx a i)  = eval a ! eval i
eval (Value v)    = v

evalFun      :: (Dp a → Dp b) → a → b
evalFun f x   = (eval ∘ f ∘ Value) <*> x

evalWhile    :: (a → Bool) → (a → a) → a → a
evalWhile c b i = if c i then evalWhile c b <*> b i else i

```

Function *eval* plays no role in generating C, but may be useful for testing.

6.4 Class Syn

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

```

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

```

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

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

```

instance 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 -> 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.

```

minim :: Ord a => Dp a -> Dp a -> Dp a
minim m n = (m <. n) ? (m, n)

```

6.5 Embedding pairs

We set up a correspondence between host language pairs in the shallow embedding and 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 6.1.)

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

```

for :: Syn a => Dp Int -> a -> (Dp Int -> a -> a) -> a
for n x b = snd (while (\(i, x) -> i <. n) (\(i, x) -> (i + 1, b i x)) (0, x))

```

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

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

6.6 Embedding undefined

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

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

```

class Syn a => Undef a where
  undef :: a
instance Undef (Dp Bool) where
  undef = false

```

```

instance Undef (Dp Int) where
  undef = 0

```

```

instance Undef (Dp Float) where
  undef = 0

```

```

instance (Undef a, Undef b) => Undef (a, b) where
  undef = (undef, undef)

```

For example,

```

(/#) :: Dp Float -> Dp Float -> Dp Float
x /# y = (y ==. 0) ? (undef, x / y)

```

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

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

6.7 Embedding option

We now explain in detail the `Option` type seen in Section 6.2.

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

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

```

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)

```

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

```

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))

```

However, this adds type constraint `Undef b` to the type of `(>=)`, which is not permitted. This need to add constraints often arises,

and has been dubbed the constrained-monad problem (Hughes 1999; Sculthorpe et al. 2013; Svenningsson and Svensson 2013). We solve it with a trick due to Persson et al. (2011).

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

```

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 (λg → unO m (λx → unO (k x) g))
instance Undef 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'
  some :: a → Opt a
  some a = lift (some' a)
  none :: Undef a ⇒ Opt a
  none = lift none'
  option :: (Undef a, Undef b) ⇒ b → (a → b) → Opt a → b
  option d f o = option' d f (lower o)

```

These definitions support the CDSL code presented in Section 6.2.

6.8 Embedding vector

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

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

```

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) = Arr n (toDp ∘ g)
  fromDp a = Vec (ArrLen a) (λi → fromDp (ArrIx a i))
instance Functor Vec where
  fmap f (Vec n g) = Vec n (f ∘ g)

```

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

Here are some primitive operations on vectors

```

zipWithVec :: (Syn a, Syn b) ⇒ (a → b → c) → Vec a → Vec b → Vec c
zipWithVec f (Vec m g) (Vec n h) = Vec (minim m n) (λi → f (g i) (h i))
sumVec :: (Syn a, Num a) ⇒ Vec a → a
sumVec (Vec n g) = for n 0 (λi x → x + g i)

```

Computing *zipWithVec f u v* combines vectors *u* and *v* point-wise with *f*, and computing *sumVec v* sums the elements of vector *v*.

We can easily define any function over vectors where each vector element is computed independently, including *drop*, *take*, *reverse*, vector concatenation, and the like, but it may be harder to do so when there are dependencies between elements, as in computing a running sum.

6.9 Fusion

Using our primitives, it is easy to compute the scalar product of two vectors.

```

scalarProd :: (Syn a, Num a) ⇒ Vec a → Vec a → a
scalarProd u v = sumVec (zipWithVec (×) u v)

```

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

```

scalarProd u v
  ⇓ sumVec (zipWith (×) u v)
  ⇓ sumVec (zipWith (×) (Vec m g) (Vec n h))
  ⇓ sumVec (Vec (min m n) (λi → g i × h i))
  ⇓ for (min m n) (λi x → x + g i × h i)

```

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

If we define

```

norm :: Vec (Dp Float) → Dp Float
norm v = scalarProd v v

```

invoking *cdsl norm* yields C code that accepts an array. The coercion from array to *Vec* is automatically inserted thanks to the *Syn* class.

There are some situations where fusion is not beneficial, notably when an intermediate vector is accessed many times fusion will cause the elements to be recomputed. An alternative is to materialise the vector in memory with the following function.

```

memorise :: Syn a ⇒ Vec a → Vec a
memorise (Vec n g) = Vec n (λi → fromDp (ArrIx (Arr n (toDp ∘ g)) i))

```

The above definition depends on common subexpression elimination to ensure *Arr n (toDp ∘ g)* is computed once, rather than once for each element of the resulting vector.

For example, if

```

blur :: Syn a ⇒ Vec a → Vec a

```

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

```

blur (blur v) or blur (memorise (blur v))

```

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

Related work

[TODO: Section 7 of ESOP submission]
[TODO: Citations for quotation, macros, early DSLs in Lisp]

8. Conclusion

[TODO: Section 8 of ESOP submission.]

9. References

- E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A Domain Specific Language for Digital Signal Processing Algorithms. In *MEMOCODE*, 2010.
- M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL*, 2005.
- J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.
- J. Hughes. Restricted data types in Haskell. In *Haskell*, 1999.
- S. Lindley. Extensional rewriting with sums. In *TLCA*, 2007.
- J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *JFP*, 8(03):275–317, 1998.
- A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *IFL*, 2011.
- F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, 1988.
- Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almqvist and Wiksell, Stockholm, 1965.
- N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *ICFP*, 2013.
- J. Svenningsson and B. Svensson. Simple and compositional reification of monadic embedded languages. In *ICFP*, 2013.