QDSLs: Why its nicer to be quoted normally

Shayan Najd[1], Sam Lindley[1], Josef Svenningsson[2], and Philip Wadler[1]

[1] The University of Edinburgh
sh.najd@ed.ac.uk, sam.lindley@ed.ac.uk, philip.wadler@ed.ac.uk
[2] Chalmers University of Technology
josefs@chalmers.se

Abstract. We describe a technique for engineering domain-specific languages based on quotation and normalisation of quoted terms, which we dub QDSL. We compare our technique to a standard approach combining deep and shallow embedding, which we dub CDSL. We draw attention to the importance of normalisation and Gentzen's subformula property for QDSLs. We implement our system, and measure five benchmarks in QDSL and CDSL implementations of Feldspar.

1 Introduction

Good artists copy, great artists steal. — Picasso

Which shall it be, CDSL or QDSL?

Say you wish to use a domain-specific language embedded in a host language to generate code in a target language. One widely-used technique combines deep and shallow embedding, which we dub CDSL (Combined Domain Specific Language). Here we introduce a second technique based on quotation and normalisation of quoted terms, which dub QDSL (Quoted Domain Specific Language).

CDSL 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 concrete syntax, and the abstract syntax of its host language.

CDSL sometimes, but not always, avoids the need for normalisation. QDSL depends crucially on normalisation. Each CDSL has a different deep embedding, so when normalisation is required, a new normaliser needs to be written for each CDSL. In contrast, all QDSLs for a host language share the same deep embedding—namely, the abstact syntax of the host language—so they can share a single normaliser.

Both CDSL and QDSL steal types from the host language. Sometimes, it proves convenient to steal a type from the host even when we expect it never to appear in target. The most common example is when we exploit higher-order types in the host even when the target supports only first-order types. But other examples are plentiful; here we provide an example of using the *Maybe* type of the host even though we don't expect to provide that type in the target. We show how these situations are neatly handled by an application of Gentzen's subformula property, exploiting a result from logic in 1935 to advance computing eight decades later.

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

The QDSL technique (though not the name) was first proposed by Cheney et al. (2013), who used it to integrate SQL queries into F#. They conjectured that the technique applies more widely, and here we test that conjecture by applying QDSL to Feldspar, a DSL for signal processing in Haskell that generates C (Axelsson et al., 2010). Our technique depends on GHC Haskell typed quasi-quotations (Mainland, 2007).

One key aspect of QDSL—normalisation of quoted terms—appears only in Cheney et al. (2013) and here. However, another aspect of QDSL—viewing domain-specific languages as quoted terms—is widely used in other systems, including Lisp macros, F# and C# LINQ (Syme, 2006; Meijer et al., 2006), and Scala Lightweight Modular Staging (LMS) (Rompf and Odersky, 2010). In F# LINQ quotation and anti-quotation are explicit, as here, while in C# LINQ and Scala LMS quotation and anti-quotation are controlled by type inference.

Feldspar exploits a combination of deep and shallow embedding, here dubbed CDSL. The technique is clearly described by Svenningsson and Axelsson (2012), and further refined by Persson et al. (2011) and Svenningsson and Svensson (2013). Essentially the same technique is also applied in Obsidian (Svensson et al., 2011) and Nikola (Mainland and Morrisett, 2010).

In a landmark paper, Gentzen (1935) introduced the two formulations of logic most widely used today, natural deduction and sequent calculus. Gentzen's main technical result was to establish the *subformula* property: any proof may be put in a normal form where all formulas it contains are subformulas of its hypotheses or conclusion. Cheney et al. (2013) applied this result to ensure queries with higher-order components always simplify to first-order queries, easily translated to SQL. Similarly here, our QDSL source may refer to higher-order concepts or data types such as *Maybe*, while we ensure that these do not appear in the generated code. The idea is not limited to QDSL, and the conclusion sketches how to apply the same idea to CDSL.

We make the following contributions.

- We introduces the names CDSL and QDSL, and provide a concise description and comparison of the two techniques.
- We highlight the benefits of normalisation and the subformula property.
- We introduce 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 empirically evaluate CDSL and QDSL implementations of Feldspar on five benchmarks.

Section 2 introduces and compares the CDSL and QDSL approaches. Section 3 reviews how CDSL works in detail, in the context of Feldspar. Section 4 describes how QDSL works in detail, reworking the examples of Section 3. Section 5 describes our normalisation algorithm. Section 6 describes our implementation and empirical evaluation. Section 7 summarises related work. Section 8 concludes.

2 Overview

2.1 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,

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

should result from instantiating *power* to (-6).

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

```
cdsl :: (Dp a → Dp b) → 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 *Float* → *Float* in the original becomes *Dp Float* → *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 **if** L **then** M **else** N for the former but M .==. N and L ? (M, N) for the latter.

Assuming x contains a value of type *Dp Float* denoting an object variable u of type float, evaluating *power* (−6) x yields following.

$$(u \mathbin{.==.} 0)\ ?\ (0, 1\ /\ (u * ((u * 1) * (u * 1))) * (u * ((u * 1) * (u * 1))))$$

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

v	$(u * 1)$
w	$u * (v * v)$
main	$(u \mathbin{.==.} 0)\ ?\ (0, 1\ /\ (w * w))$

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

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

```
qdsl :: Qt (a → b) → 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 *Float* → *Float* in the original becomes *Qt* (*Float* → *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 $[|| \cdots ||]$ and unquoting by $\$\$(\cdots)$. Here, by the mechanism of quoting, without any need for tricks, the syntax for code excecuted at both generation-time and run-time is identical for all constructs, including comparison and conditionals.

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 * 1)) x)) x))) x)) x||]
```

Normalising, 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||]
```

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

Comparison Here are some points of comparison between the two approaches.

- A function $a \to b$ is embedded in CDSL as $Dp\ a \to Dp\ b$, a function between representations, and in QDSL as $Qt\ (a \to b)$, 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 unnessary distraction or as drawing a useful distinction between generation-time and run-time. If one takes the former view, the type-based approach to quotation found in C# and Scala might be preferred.

– CDSL typically develops custom shallow and deep embeddings for each application, although these may follow a fairly standard pattern (as we review in Section 3). 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 be 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 2.2. 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.
– Once the deep embedding or the normalised quoted term is produced, generating the domain-specific code is similar for both approaches.

2.2 Second example

In the previous code, we arbitrarily chose that raising zero to a negative power yields zero. Say that we wish to exploit the *Maybe* type 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.

```
power' :: Int → Float → Maybe Float
power' n x =
  if n < 0 then
    if x == 0 then Nothing else 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 → Float → Float
power'' n x = maybe 0 (λx → x) (power' n x)
```

Here *sqr* is as before. The above uses

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

from the Haskell prelude. Type *Maybe* is declared as a monad, enabling the **do** notation, which translates into (>>=). The same C code as before should result from instantiating $power''$ to (-6).

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.

CDSL In CDSL, *Maybe* is represented by *Opt*. Here is the refactored code.

```
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)
```

Here *sqr* is as before. The above uses the functions

```
none   :: Opt a
return :: a → Opt a
(>>=)  :: Opt a → (a → Opt b) → Opt b
option :: (Syn a, Syn b) ⇒ b → (a → b) → Opt a → b
```

from the CDSL library. Details of the type *Opt* and the type class *Syn* are explained in Section 3.5. Type *Opt* is declared as a monad, enabling the **do** notation, which translates into (>>=). Invoking *cdsl* (*power''* (−6)) generates the same C code as the previous example.

In order to be easily represented in C, type *Opt a* is represented as a pair consisting of a boolean and the representation of the type *a*; in the case that corresponds to *Nothing*, the boolean is false and a default value of type *a* is provided. The CDSL term generated by evaluating *power* (−6) 0 is large and unscrutable:

```
(((fst ((x == (0.0)) ? ((((False) ? ((True), (False))), ((False) ? (undef,
undef))), ((True), ((1.0) / ((x * ((x * (1.0)) * (x * (1.0)))) * (x * ((x *
(1.0)) * (x * (1.0))))))))))) ? ((True), (False))) ? (((fst ((x == (0.0)) ?
((((False) ? ((True), (False))), ((False) ? (undef, undef))), ((True), ((1.0) /
((x * ((x * (1.0)) * (x * (1.0)))) * (x * ((x * (1.0)) * (x * (1.0))))))))))) ?
((snd ((x == (0.0)) ? ((((False) ? ((True), (False))), ((False) ? (undef,
undef))), ((True), ((1.0) / ((x * ((x * (1.0)) * (x * (1.0)))) * (x * ((x *
(1.0)) * (x * (1.0))))))))))), undef)), (0.0)))
```

Before, evaluating *power* yielded an CDSL term essentially in normal form, save for the need to use common subexpression elimination or observable sharing

to recover shared structure. However, this is not the case here. Rewrite rules including the following need to be repeatedly applied.

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

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

Hence, an advantages of the 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.

QDSL In QDSL, type *Maybe* is represented by itself. 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 (λx → x) ($$(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 $Qt\ (\mathit{powerQ''}\ (-6))$ yields a term of similar complexity to the term yielded by the CDSL. Normalisation by the rules discussed in Section 5 reduces the term to the same form as before, which in turn generates the same C as before.

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 Cheney et al. (2013) expoited 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.

3 MiniFeldspar as a CDSL

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 Svenningsson and Axelsson (2012).

3.1 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 oa 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 ($<\!\!*\!\!>$).

$$
\begin{array}{ll}
(<\!\!*\!\!>) & :: (a \to b) \to a \to b \\
f <\!\!*\!\!> x & = \mathit{seq}\ x\ (f\ x) \\
\mathit{eval} & :: \mathit{Dp}\ a \to a \\
\mathit{eval}\ (\mathit{LitI}\ i) & = i \\
\mathit{eval}\ (\mathit{LitF}\ x) & = x \\
\mathit{eval}\ (\mathit{LitB}\ b) & = b \\
\mathit{eval}\ (\mathit{If}\ c\ t\ e) & = \mathbf{if}\ \mathit{eval}\ c\ \mathbf{then}\ \mathit{eval}\ t\ \mathbf{else}\ \mathit{eval}\ e \\
\mathit{eval}\ (\mathit{While}\ c\ b\ i) & = \mathit{evalWhile}\ (\mathit{evalFun}\ c)\ (\mathit{evalFun}\ b) <\!\!*\!\!> \mathit{eval}\ i \\
\mathit{eval}\ (\mathit{Pair}\ a\ b) & = (,) <\!\!*\!\!> \mathit{eval}\ a <\!\!*\!\!> \mathit{eval}\ b \\
\mathit{eval}\ (\mathit{Fst}\ p) & = \mathit{fst} <\!\!*\!\!> \mathit{eval}\ p \\
\mathit{eval}\ (\mathit{Snd}\ p) & = \mathit{snd} <\!\!*\!\!> \mathit{eval}\ p \\
\mathit{eval}\ (\mathit{Prim1}\ \_\ f\ a) & = f <\!\!*\!\!> \mathit{eval}\ a \\
\mathit{eval}\ (\mathit{Prim2}\ \_\ f\ a\ b) & = f <\!\!*\!\!> \mathit{eval}\ a <\!\!*\!\!> \mathit{eval}\ b \\
\mathit{eval}\ (\mathit{Arr}\ n\ g) & = \mathit{array}\ (0, n')\ [(i, \mathit{eval}\ (g\ (\mathit{LitI}\ i))) \mid i \leftarrow [0\,..\,n']] \\
 & \quad \mathbf{where}\ n' = \mathit{eval}\ n - 1 \\
\mathit{eval}\ (\mathit{ArrLen}\ a) & = u - l + 1\ \mathbf{where}\ (l, u) = \mathit{bounds}\ (\mathit{eval}\ a) \\
\mathit{eval}\ (\mathit{ArrIx}\ a\ i) & = \mathit{eval}\ a\,!\,\mathit{eval}\ i \\
\mathit{eval}\ (\mathit{Value}\ v) & = v \\
\mathit{evalFun} & :: (\mathit{Dp}\ a \to \mathit{Dp}\ b) \to a \to b \\
\mathit{evalFun}\ f\ x & = (\mathit{eval} \circ f \circ \mathit{Value}) <\!\!*\!\!> x \\
\mathit{evalWhile} & :: (a \to \mathit{Bool}) \to (a \to a) \to a \to a \\
\mathit{evalWhile}\ c\ b\ i & = \mathbf{if}\ c\ i\ \mathbf{then}\ \mathit{evalWhile}\ c\ b <\!\!*\!\!> b\ i\ \mathbf{else}\ i
\end{array}
$$

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

3.2 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 ∘ fromDp) (toDp ∘ b ∘ fromDp) (toDp i))
```

Numbers are made convenient to manipulate via overloading.

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

With this declaration, 1+2::*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)
```

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

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

Here is how to compute the minimum of two values.

$$
\begin{array}{ll}
minim & :: Ord\ a \Rightarrow Dp\ a \to Dp\ a \to Dp\ a \\
minim\ m\ n & = (m\ .<.\ n)\ ?\ (m, n)
\end{array}
$$

3.3 Embedding pairs

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

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

This permits us to manipulate pairs as normal, with (a, b), $fst\ a$, and $snd\ a$. (Argument p is duplicated in the definition of *fromDp*, which may require common subexpression elimination as discussed in Section 2.1.)

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

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

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

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

3.4 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.

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

3.5 Embedding option

We now explain in detail the *Option* type seen in Section 2.2.

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

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

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

We introduce a second continuation-passing style (cps) type Opt, defined in terms of the representation type Opt'. It is straightforward to define *Monad* and *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
```

$$
\begin{aligned}
&\mathit{none} &&= \mathit{lift}\ \mathit{none}' \\
&\mathit{option} &&:: (\mathit{Undef}\ a, \mathit{Undef}\ b) \Rightarrow b \to (a \to b) \to \mathit{Opt}\ a \to b \\
&\mathit{option}\ d\ f\ o &&= \mathit{option}'\ d\ f\ (\mathit{lower}\ o)
\end{aligned}
$$

These definitions support the CDSL code presented in Section 2.2.

3.6 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*.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Vec}\ a = \mathit{Vec}\ (\mathit{Dp}\ \mathit{Int})\ (\mathit{Dp}\ \mathit{Int} \to a) \\
&\textbf{instance}\ \mathit{Syn}\ a \Rightarrow \mathit{Syn}\ (\mathit{Vec}\ a)\ \textbf{where} \\
&\quad \textbf{type}\ \mathit{Internal}\ (\mathit{Vec}\ a) = \mathit{Array}\ \mathit{Int}\ (\mathit{Internal}\ a) \\
&\quad \mathit{toDp}\ (\mathit{Vec}\ n\ g) &&= \mathit{Arr}\ n\ (\mathit{toDp} \circ g) \\
&\quad \mathit{fromDp}\ a &&= \mathit{Vec}\ (\mathit{ArrLen}\ a)\ (\lambda i \to \mathit{fromDp}\ (\mathit{ArrIx}\ a\ i)) \\
&\textbf{instance}\ \mathit{Functor}\ \mathit{Vec}\ \textbf{where} \\
&\quad \mathit{fmap}\ f\ (\mathit{Vec}\ n\ g) &&= \mathit{Vec}\ n\ (f \circ g)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{zipWithVec} :: (\mathit{Syn}\ a, \mathit{Syn}\ b) \Rightarrow (a \to b \to c) \to \mathit{Vec}\ a \to \mathit{Vec}\ b \to \mathit{Vec}\ c \\
&\mathit{zipWithVec}\ f\ (\mathit{Vec}\ m\ g)\ (\mathit{Vec}\ n\ h) = \mathit{Vec}\ (\mathit{minim}\ m\ n)\ (\lambda i \to f\ (g\ i)\ (h\ i)) \\
&\mathit{sumVec} &&:: (\mathit{Syn}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Vec}\ a \to a \\
&\mathit{sumVec}\ (\mathit{Vec}\ n\ g) &&= \mathit{for}\ n\ 0\ (\lambda i\ x \to x + g\ i)
\end{aligned}
$$

Computing $\mathit{zipWithVec}\ f\ u\ v$ combines vectors u and v pointwise with f, and computing $\mathit{sumVec}\ v$ 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 concatentation, and the like, but it may be harder to do so when there are dependencies between elements, as in computing a running sum.

3.7 Fusion

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

$$
\begin{aligned}
&\mathit{scalarProd} &&:: (\mathit{Syn}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Vec}\ a \to \mathit{Vec}\ a \to a \\
&\mathit{scalarProd}\ u\ v &&= \mathit{sumVec}\ (\mathit{zipWithVec}\ (*)\ u\ v)
\end{aligned}
$$

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.

$$\begin{aligned}
& scalarProd\ u\ v \\
\rightsquigarrow\ & sumVec\ (zipWith\ (*)\ u\ v) \\
\rightsquigarrow\ & sumVec\ (zipWith\ (*)\ (Vec\ m\ g)\ (Vec\ n\ h) \\
\rightsquigarrow\ & sumVec\ (Vec\ (min\ m\ n)\ (\lambda i \rightarrow g\ i * h\ i) \\
\rightsquigarrow\ & for\ (min\ m\ n)\ (\lambda i\ x \rightarrow x + g\ i * h\ i)
\end{aligned}$$

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

$$\begin{aligned}
& norm\ \ :: Vec\ Float \rightarrow Float \\
& norm\ v = scalarProd\ v\ v
\end{aligned}$$

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.

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

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

For example, if

$$blur :: Syn\ a \Rightarrow Vec\ a \rightarrow Vec\ a$$

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

$$blur\ (blur\ v) \quad \text{or} \quad blur\ (memorise\ (blur\ v))$$

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.

4 MiniFeldspar as a QDSL

In eight pages, we explained how to combine deep and shallow embedding to produce a library that allows all CDSL code in Section 2 to run. Now, in three pages, we will cover what is required to achieve the same effect in QDSL.

To aid comparison, our CDSL and QDSL implementations both represent target code as terms of type *Dp*. Terms of *Qt* are normalised to ensure the subformula property and then translated to equivalent terms of type *Dp*. The postprocessor that converts *Dp* to C code is shared among both implementations.

4.1 While and for

For CDSL, we had a deep construct and corresponding "smart constructor".

$$\begin{aligned}&\mathit{While} :: (\mathit{Dp}\ a \to \mathit{Dp}\ \mathit{Bool}) \to (\mathit{Dp}\ a \to \mathit{Dp}\ a) \to \mathit{Dp}\ a \to \mathit{Dp}\ a\\ &\mathit{while}\ \ :: \mathit{Syn}\ a \Rightarrow (a \to \mathit{Dp}\ \mathit{Bool}) \to (a \to a) \to (a \to a)\end{aligned}$$

For QDSL, we follow the same pattern, but it's even simpler:

$$\mathit{while} :: (a \to \mathit{Bool}) \to (a \to a) \to (a \to a)$$

This identifier is declared in the QDSL library with the appropriate type, and conversion from *while* to *While* occurs when translating *Qt* to *Dp*.

The definition of the *for* loop is given by:

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

The code is similar in structure to that for CDSL, except the type is simpler, quasi-quotes surround the body, and $(.<.)$ in CDSL is replaced by $(<)$ in QDSL.

4.2 Embedding maybe

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

$$\begin{aligned}&\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a\\ &\mathit{return} && :: a \to \mathit{Maybe}\ a\\ &\mathit{return} && = \mathit{Just}\\ &(\gg\!=) && :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b\\ &m \gg\!= k && = \mathbf{case}\ m\ \mathbf{of}\ \{\mathit{Nothing} \to \mathit{Nothing}; \mathit{Just}\ x \to k\ x\}\\ &\mathit{maybe} && :: b \to (a \to b) \to \mathit{Maybe}\ a \to b\\ &\mathit{maybe}\ x\ g\ m && = \mathbf{case}\ m\ \mathbf{of}\ \{\mathit{Nothing} \to x; \mathit{Just}\ y \to g\ y\}\end{aligned}$$

The QDSL processor permits the constructors *Nothing* and *Just*, and replaces occurrences of *return*, $(\gg\!=)$, and *maybe* by the above definitions, prior to normalisation. The normaliser performs simplifications including the rules below.

$$\begin{aligned}&\mathbf{case}\ \mathit{Nothing}\ \mathbf{of}\ \{\mathit{Nothing} \to M; \mathit{Just}\ x \to N\} \rightsquigarrow M\\ &\mathbf{case}\ \mathit{Just}\ L\ \mathbf{of}\ \{\mathit{Nothing} \to M; \mathit{Just}\ x \to N\} \rightsquigarrow N[x := L]\end{aligned}$$

This is adequate to eliminate all occurrences of the *Maybe* type from the code in Section 2.2 after normalisation. 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.

4.3 Embedding array

The *Vec* type is defined as follows.

$$\textbf{data}\ \mathit{Vec}\ a = \mathit{Vec}\ \mathit{Int}\ (\mathit{Int} \to a)$$

The QDSL processor permits the constructor *Arr*, and performs simplifications including the rule below.

$$\textbf{case}\ \mathit{Vec}\ L\ M\ \textbf{of}\ \{\ \mathit{Vec}\ n\ g \to N\ \} \rightsquigarrow N[n := L, g := M]$$

This is adequate to eliminate all occurrences of the *Vec* type from the code below after normalisation. Again, the subformula property is key: because the final type of the result does not involve *Vec*, it is certain that normalisation will remove all its occurrences.

Here are some primitive operators on vectors, in QDSL style.

$$\begin{aligned}
&\mathit{zipWithVec} :: \mathit{Qt}\ ((a \to b \to c) \to \mathit{Vec}\ a \to \mathit{Vec}\ b \to \mathit{Vec}\ c)\\
&\mathit{zipWithVec} = [||\lambda f\ (\mathit{Vec}\ m\ g)\ (\mathit{Vec}\ n\ h) \to\\
&\qquad\qquad \mathit{Vec}\ (\$\$\mathit{minim}\ m\ n)\ (\lambda i \to f\ (g\ i)\ (h\ i))||]\\
&\mathit{sumVec} \quad :: \mathit{Num}\ a \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to a)\\
&\mathit{sumVec} \quad = [||\lambda(\mathit{Vec}\ n\ g) \to \$\$\mathit{for}\ n\ 0\ (\lambda i\ x \to x + g\ i)||]
\end{aligned}$$

This is identical to the previous code, save for additions of quotation and anti-quotation.

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

$$\begin{aligned}
&\mathit{scalarProd} :: \mathit{Num}\ a \Rightarrow \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Vec}\ a \to a)\\
&\mathit{scalarProd} = [||\lambda u\ v \to \$\$\mathit{sumVec}\ (\$\$\mathit{zipWithVec}\ (*)\ u\ v)||]
\end{aligned}$$

With CDSL, evaluation in the host language achieves fusion. With QDSL, normalisation in the QDSL processor achieves fusion. As noted above, the subformula property is sufficient to guarantee fusion is achieved.

Ultimately, we need to be able to manipulate actual arrays. For this purpose, we provide three constants analogous to three constructs of the type *Dp*.

$$\begin{aligned}
&\mathit{arr} \quad\ :: \mathit{Int} \to (\mathit{Int} \to a) \to \mathit{Array}\ \mathit{Int}\ a\\
&\mathit{arrLen} :: \mathit{Array}\ \mathit{Int}\ a \to \mathit{Int}\\
&\mathit{arrIx} \quad :: \mathit{Array}\ \mathit{Int}\ a \to \mathit{Int} \to a
\end{aligned}$$

These identifiers are declared in the QDSL library with the appropriate types, and conversion from *arr*, *arrLen*, *arrIx* to *Arr*, *ArrLen*, *ArrIx* occurs when translating *Qt* to *Dp*.

We convert between vectors and arrays as follows.

$$
\begin{aligned}
&\mathit{toArr} &&:: \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Array}\ \mathit{Int}\ a)\\
&\mathit{toArr} &&= [||\lambda(\mathit{Vec}\ n\ g) \to \mathit{arr}\ n\ g||]\\
&\mathit{fromArr} &&:: \mathit{Qt}\ (\mathit{Array}\ \mathit{Int}\ a \to \mathit{Vec}\ a)\\
&\mathit{fromArr} &&= [||\lambda a \to \mathit{Vec}\ (\mathit{arrLen}\ a)\ (\lambda i \to \mathit{arrIx}\ a\ i)||]
\end{aligned}
$$

If we define

$$
\begin{aligned}
&\mathit{norm} &&:: \mathit{Qt}\ (\mathit{Vec}\ \mathit{Float} \to \mathit{Float})\\
&\mathit{norm}\ v &&= \mathit{scalarProd}\ v\ v
\end{aligned}
$$

invoking $\mathit{qdsl}\ (\mathit{norm} \circ \mathit{fromArr})$ yields C code that accepts an array. In contrast to CDSL, the coercion must be inserted by hand. The advantage is that whereas for CDSL we must resort to operational reasoning to determine which instances of *Vec* are eliminated, with QDSL we can apply the subformula property. We know that all occurrences of *Vec* must be eliminated since it appears as neither an input nor output of the final program.

Defining an analogue of *memorise* is straightforward.

$$
\begin{aligned}
&\mathit{memorise} &&:: \mathit{Qt}\ (\mathit{Vec}\ a \to \mathit{Vec}\ a)\\
&\mathit{memorise} &&= [||\mathit{fromArr} \circ \mathit{toArr}||]
\end{aligned}
$$

The last performs the same purpose as *memorise* for CDSL, enabling the user to decide when a vector is to be materialised in memory.

5 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 1. We let A, B, C range over types, including base types (ι), functions ($A \to B$), products ($A \times B$), and sums ($A + B$). We let L, M, N range over terms, and x, y, z range over variables, and c range over primitive constants, which must be fully applied. We follow the usual convention that terms are equivalent up to renaming of bound variables. We write $FV(N)$ for the set of free variables of N, and $N[x := M]$ for substitution of M for x in N. We let V, W range over values, and P range over non-values (that is, any term that is not a value).

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

The grammar of normal forms is given in Figure 2. We re-use L, M, N to ranges over terms in normal form, V, W to range over values in normal form, and Q to range over neutral forms.

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

Fig. 1. Types, Terms, and Values

$$
\begin{array}{lll}
\text{Neutral Forms} & Q & ::= x\ W \mid c\ W \mid Q\ W \mid \mathbf{fst}\ x \mid \mathbf{snd}\ x \\
\text{Normal Values} & V, W & ::= x \mid \lambda x.N \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ W \\
\text{Normal Forms} & N, M & ::= Q \mid V \mid \mathbf{case}\ z\ \mathbf{of}\ \{\mathbf{inl}\ x.N; \mathbf{inr}\ y.M\} \mid \mathbf{let}\ x = Q\ \mathbf{in}\ N
\end{array}
$$

Fig. 2. Normal Forms

Reduction rules for normalisation are presented in Figure 3, broken into three phases. We write $M \mapsto_i N$ to indicate that M reduces to N in phase i. We let F and G range over two different forms of evaluation frame used in Phases 2 and 3 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. Phase 2 performs standard β and commuting reductions, and is the only phase that is crucial 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 may 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 1 (Strong normalisation). *Each of the reduction relations $\mapsto_i$ is 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 (e.g. Lindley (2007)). If the target language includes general recursion, normalisation should treat fixpoint as an uninterpreted constant (or free variable).

The grammar of Figure 2 characterises normal forms precisely.

Proposition 2 (Normal Form Syntax). *An expression N matches the syntax of normal forms in Figure 2 if and only if it is in normal form with regard to the reduction rules of Figure 3.*

The *subformulas* of a type are the type itself and its components; for instance, the subformulas of $A \to B$ are $A \to B$ itself and the subformulas of A and B.

Phase 1 (let-insertion)

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

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

Phase 2 (symbolic evaluation)

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

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

Phase 3 (garbage collection)

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

Fig. 3. Normalisation rules

The *proper subformulas* of a type are all its subformulas other than the type itself. Terms in normal form satisfy the subformula property.

Proposition 3 (Subformula property). *If $\Gamma \vdash M : A$ and the normal form of M is N by the reduction rules of Figure 3, then $\Gamma \vdash N : A$ and every subterm of N has a type that is either a subformula of A or a subformula of a type in Γ. Further, every subterm other than N itself and free variables of N has a type that is a proper subformula of A or a proper subformula of a type in Γ.*

6 Implementation and evaluation

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

- It expands identifiers connected with the types $(,)$, *Maybe* and *Vec*.
- It normalises the term to ensure the subformula property. Normalisation includes the special-purpose rules for *Maybe* and *Vec* given in Section 4 and the general-purpose rules of Section 5.
- 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: $(==)$, $(<)$, $(+)$, $(*)$, and similar, plus *while*, *arr*, *arrLen*, and *arrIx*.

	QDSL				CDSL without CSE				CDSL 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

Table 1. Comparison of CDSL and QDSL MiniFeldspar (all times in seconds)

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 Typed Haskell, $TH.Q$ is the quotation monad of Typed Haskell (used to look up identifiers and generate fresh names), and $TH.TExp\ a$ is the parse tree for an expression that returns a value of type a. In the latter, a is a ghost variable; type $TH.TExp\ a$ is a synonym for $TH.Exp$, the (untyped) parse tree of an expression in Template Haskell. 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 also why we translate the *Maybe* monad as a special case, rather than supporting overloading for monad operations.

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

Table 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 class *Syn* 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).

7 Related work

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

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

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

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

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

8 Conclusion

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

The subformula property may have applications in DSLs other that QDSLs. For instance, after Section 3.5 of this paper was drafted, it occurred to us that a different approach to options in CDSL would be to extend type *Dp* with constructs for type *Maybe*. So long as type *Maybe* does not appear in the input or output of the program, a normaliser that ensures the subformula property could guarantee that C code for such constructs need never be generated.

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

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

– The user may supply a type environment indicating which constants (or free variables) may appear in typed quasi-quotations.

Such a tool could easily subsume the special-purpose translator from *Qt* to *Dp* described at the beginning of Section 6, and lift most of its restrictions. For instance, the current prototype is restricted to the *Maybe* monad, while the envisioned tool will work with any monad.

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

Acknowledgement This work was funded by EPSRC Grant EP/K034413/1 and a Google European Doctoral Fellowship.

9 References

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

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

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

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

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

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

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

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

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

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

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

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

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

J. Hughes. Restricted data types in Haskell. In *Haskell*, 1999.
S. Lindley. Extensional rewriting with sums. In *TLCA*, 2007.
G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Haskell*, 2007.
G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Haskell*, 2010.
J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *JFP*, 8(03):275–317, 1998.
E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.
J. ODonnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming*. Springer, 1993.
A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *IFL*, 2011.
F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, 1988.
A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *ICSE*, 1999.
T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2010.
N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *ICFP*, 2013.
J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.
J. Svenningsson and B. Svensson. Simple and compositional reification of monadic embedded languages. In *ICFP*, 2013.
J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *IFL*. 2011.
Don Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML*, 2006.