# 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 describes a new approach to domain specific languages, called QDSL, that resurrects two old ideas: quoted terms for domain specific languages, 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 guide loop fusion. We give three examples of QDSL: QFeldspar (a variant of Feldsar), P-LINQ for F#, and some uses of Scala LMS; and we provide a comparison between QDSL and EDSL (embedded DSL).

## 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

lambda calculus; domain-specific language; DSL; embedded languages; EDSL; quotation; normalisation

## 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 macros in Lisp. Today, a more fashionable technique is Embdedded 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 that depends crucially on normalising the quoted term, which we dub Quoted DSLs (QDSLs).

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.

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 up that claim, we describe the EDSL approach to Feldspar for purposes of comparison. The QDSL description occupies TODO:M pages, while the EDSL decription 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, and we suggest that such DSLs may benefit from the subformula property. 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.

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

Our approach exploits the fact that normalised terms satisfy the subformula property of Gentzen (1935). The subformula property provides users of the DSL with useful guarantees, such as the following:

- write higher-order terms while guaranteeing to generate first-order code;
- write a sequence of loops over arrays while guaranteeing to generate code that fuses those loops;
- write nested intermediate terms 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 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.

> 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 this is often only partially the case. For instance, an EDSL such as Feldspar or Nicola, when embedded in Haskell, can use 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 of the host, hence necessarily identical.

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 5 we give two 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 the second, the EDSL must perform some 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 suggest the general value of an approach to building DSLs based on quotation, normalisation of quoted terms, and the subformula property, and to name this approach QDSL. (Section 1.)
- To present the design of a QDSL implementation of Feldspar, and show its implementation length and performance is comparable to an EDSL implementation of Feldspar. (Section 2.)
- 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 3.)
- 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 4.)
- To argue that, from the user's point of view, the QDSL implementation of Feldspar is conceptually easier to understand than the EDSL implementation of Feldspar, by a detailed comparison of the user interface of the two implementations (Section 5.)

Section 6 describes related work, and Section 7 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 5.

### 2.1 An introductory example

We are particularly interested in DSLs that perform *staged* computation, where at code-generation time we use host code to generate target code that is to be executed at runtime.

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:

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

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

While Feldspar programs often use higher-order functions, the generated C code should only use first-order data. Hence the argument type $a$ and result type $b$ of the main function must be first-order, which is indicated by the type-class restrictions $FO\ a$ and $FO\ b$. First order types include integers, floats, pairs where the components are both first-order, and arrays where the components are first-order.

$$
\begin{array}{l}
\textbf{instance}\ FO\ Int \\
\textbf{instance}\ FO\ Float \\
\textbf{instance}\ (FO\ a, FO\ b) \Rightarrow FO\ (a, b) \\
\textbf{instance}\ (FO\ a) \Rightarrow FO\ (Arr\ a)
\end{array}
$$

It is easy to add triples and larger tuples. Here type $Arr\ a$ is the type of arrays with indexed by integers with components of type $a$, with indexes beginning at zero.

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:

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

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 $\$\$(\cdots)$.

Evaluating $power\ (-6)$ yields the following:

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

Normalising using the technique of Section 3, with variables renamed for readability, yields the following:

$$
\begin{array}{l}
[||\lambda u \to \textbf{if}\ u == 0\ \textbf{then}\ 0\ \textbf{else} \\
\quad\quad\quad\quad \textbf{let}\ v = u \times 1\ \textbf{in} \\
\quad\quad\quad\quad \textbf{let}\ w = u \times (v \times v)\ \textbf{in} \\
\quad\quad\quad\quad 1\ /\ (w \times w)||]
\end{array}
$$

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 the 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 5.

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.

## 2.2 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 either one of the hypotheses or 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 $a \to b$ 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 a proper 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 $\mathit{float} \to \mathit{float}$ will only have proper subterms of type *float*, which is indeed true for the normalised term.

There is a minor problem. One of the free variables of our quoted term is

$$(\times) :: \mathit{float} \to \mathit{float} \to \mathit{float}$$

which has $\mathit{float} \to \mathit{float}$ as a subtype. This is alieviated by a standard trick: assign an arity to each free variable, and treat a free variable applied to fewer arguments than its arity as a value. We discuss the details in Section **??**.

## 2.3 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.

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. Occurrences of **do** are expanded to applications of (≫=), as usual, and *Nothing*, *return*, (≫=), and *maybe* are treated specially by the normaliser, as described below. 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. In order for the subformula property to apply, we cannot take *return*, (≫=), and *maybe* as free variables; instead, we treat them as known definitions to be eliminated by the normaliser. 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
```

The *Maybe* type is essentially a sum type, and normalisation for these is as described in Section 3.

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

## 2.4 While

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

```
while :: (FO s) ⇒ Qt ((s → Bool) → (s → s) → s → s)
```

Rather than using side-effects, the *while* primitive takes three arguments: a predicate over the current state, of type $s \to Bool$; a function from current state to new state, of type $s \to s$; and an initial state of type $s$; and it returns a final state of type $s$.

[TODO: Why don't we need to worry about intermediate values of type $s \to Bool$ or type $s \to s$?]

As explained in Section 2.1, primitives of the language to be compiled, such as (×), are treated as free variables with regard to the subformula property.

[TODO: Observe that the *FO s* restriction in the definition of *while* is crucial. Without it, the subformula property could not guarantee to eliminate types such as *Vec a* or *Maybe a*. The reason we can eliminate these types is because they are not legal as instantiations of $s$ in the definition above.

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

```
for :: (FO s) ⇒ Qt (Int → s → (Int → s → s) → s)
for = [||λn s_0 b → snd (while (λ(i, s) → i < n)
                                (λ(i, s) → (i + 1, b i s))
                                (0, s_0))||]
```

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

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.

[TODO: Here is Fibonacci, but better to have an example involving specialisation.]

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

## 2.5 Arrays

[TODO: Note that we do not have instances for *Vec a*, which prohibits creating C code that operates on these types.]

Two types, *Arr* for manifest arrays and *Vec* for "pull arrays" guaranteed to be eliminated by fusion.

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

Recall that if *FO a* then *FO* (*Arr a*), but not *FO* (*Vec a*).

We assume the following primitive operations.

```
arr    :: FO a ⇒ Int → (Int → a) → Arr a
arrLen :: FO a ⇒ Arr a → Int
arrIx  :: FO a ⇒ Arr a → Int → a

toArr   :: Qt (Vec a → Arr a)
toArr   = [||λ(Vec n g) → arr n (λx → g x)||]
fromArr :: Qt (Arr a → Vec a)
fromArr = [||λa → Vec (arrLen a) (λi → arrIx a i)||]

zipWithVec :: Qt ((a → b → c) → Vec a → Vec b → Vec c)
zipWithVec = [||λf (Vec m g) (Vec n h) →
```

$$
\begin{array}{lll}
 & & Vec\ (\$\$minim\ m\ n)\ (\lambda i \rightarrow f\ (g\ i)\ (h\ i))||] \\
sumVec & :: & (FO\ a, Num\ a) \Rightarrow Qt\ (Vec\ a \rightarrow a) \\
sumVec & = & [||\lambda(Vec\ n\ g) \rightarrow \$\$for\ n\ 0\ (\lambda i\ x \rightarrow x + g\ i)||] \\
scalarProd & :: & (FO\ a, Num\ a) \Rightarrow Qt\ (Vec\ a \rightarrow Vec\ a \rightarrow a) \\
scalarProd & = & [||\lambda u\ v \rightarrow \$\$sumVec\ (\$\$zipWithVec\ (\times)\ u\ v)||] \\
norm & :: & Qt\ (Arr\ Float \rightarrow Float) \\
norm & = & [||\lambda v \rightarrow \mathbf{let}\ w = fromArr\ v\ \mathbf{in}\ \$\$\ scalarProd\ w\ w||]
\end{array}
$$

Invoking *qdsl norm* produces the following C code.

*// [TODO: give translation to C.]*

## 2.6 Implementation

[TODO: Section 6 of ESOP submission]

# 3. 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 ($\iota$), functions ($A \rightarrow B$), products ($A \times B$), and sums ($A + B$). We let $L$, $M$, $N$ range over terms, and $x$, $y$, $z$ range over variables. We let $c$ range over primitive constants, which are fully applied (applied to a sequence of terms of length equal to the constant's arity). 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 capture-avoiding 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 $\Gamma$ range over type 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 type environment $\Gamma$. The typing rules are standard.

The grammar of normal forms is given in Figure 2. We reuse $L, M, N$ to range over terms in normal form and $V, W$ to range over values in normal form, and we let $Q$ range over neutral 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 $\beta$ 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 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 3.1 (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 (see, for example, Lindley (2007)). If the target language includes general recursion, normalisation should treat the fixpoint operator as an uninterpreted constant.

The grammar of Figure 2 characterises normal forms precisely.

Proposition 3.2 (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 $A \rightarrow B$ are $A \rightarrow B$ itself and the subformulas of $A$ and $B$. The *proper subformulas* of a type are all its subformulas other than the type itself. Terms in normal form satisfy the subformula property.

Proposition 3.3 (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 $\Gamma$. 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 $\Gamma$.*

[TODO: explain how the FO restriction on *while* works in conjunction with the subformula property.]

[TODO: explain how the subformula property interacts with *fix* as a constant in the language.]

[TODO: explain how including an uninterpreted constant *id* allows us to disable reduction whenever this is desirable.]

# 4. OTHER EXAMPLES OF QDSLS

## 4.1 F# P-LINQ

## 4.2 Scala LMS

# 5. A COMPARISON OF QDSL AND EDSL

## 5.1 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.

$$
\begin{array}{l}
power :: Int \rightarrow Float \rightarrow Float \\
power\ n\ x = \\
\quad \mathbf{if}\ n < 0\ \mathbf{then} \\
\quad\quad \mathbf{if}\ x == 0\ \mathbf{then}\ 0\ \mathbf{else}\ 1\ /\ power\ (-n)\ x \\
\quad \mathbf{else\ if}\ n == 0\ \mathbf{then} \\
\quad\quad 1 \\
\quad \mathbf{else\ if}\ even\ n\ \mathbf{then} \\
\quad\quad sqr\ (power\ (n\ \mathrm{div}\ 2)\ x) \\
\quad \mathbf{else} \\
\quad\quad x \times power\ (n-1)\ x
\end{array}
$$

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

**Figure 1: Types, Terms, and Values**

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

**Figure 2: Normal Forms**

$$
\begin{array}{l}
\mathit{sqr} \quad :: \mathit{Float} \to \mathit{Float} \\
\mathit{sqr}\ x = x \times x
\end{array}
$$

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);
  }
}
```

### *5.1.1 CDSL*

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

$$\mathit{cdsl} :: (\mathit{Type}\ a, \mathit{Type}\ b) \Rightarrow (\mathit{Dp}\ a \to \mathit{Dp}\ b) \to C$$

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.

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

Invoking $\mathit{cdsl}\ (\mathit{power}\ (-6))$ generates the C code above.

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

In 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 $\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N$ for the former but $M\ .{=}{=}.\ N$ and $L\ ?\ (M, N)$ for the latter.

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

$$\lambda u \to (u\ .{=}{=}.\ 0)\ ?\ (0, 1\ /\ ((u \times ((u \times 1) \times (u \times 1))) \times (u \times ((u \times$$

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

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

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

### *5.1.2 QDSL*

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

$$\mathit{qdsl} :: (\mathit{FO}\ a, \mathit{Type}\ a, \mathit{Type}\ b) \Rightarrow \mathit{Qt}\ (a \to b) \to C$$

[TODO: Should be as follows]

$$\mathit{qdsl} :: (\mathit{FO}\ a, \mathit{FO}\ b) \Rightarrow \mathit{Qt}\ (a \to b) \to C$$

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

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

Invoking $\mathit{qdsl}\ (\mathit{power}\ (-6))$ generates the C code above.

Phase 1 (let-insertion)

$$
\begin{array}{lll}
F & ::= & c\ (\overline{M}, [\ ], \overline{N}) \mid M\ [\ ] \mid ([\ ], N) \mid (M, [\ ]) \mid \mathbf{fst}\ [\ ] \mid \mathbf{snd}\ [\ ] \\
 & \mid & \mathbf{inl}\ [\ ] \mid \mathbf{inr}\ [\ ] \mid \mathbf{case}\ [\ ]\ \mathbf{of}\ \{\mathbf{inl}\ x.M; \mathbf{inr}\ y.N\}
\end{array}
$$

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

Phase 2 (symbolic evaluation)

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

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

Phase 3 (garbage collection)

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

**Figure 3: Normalisation rules**

Type $\mathit{Float} \to \mathit{Float}$ in the original becomes $\mathit{Qt}\ (\mathit{Float} \to \mathit{Float})$ in the QDSL solution, meaning that $\mathit{power}\ n$ 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 $\mathit{power}\ (-6)$ yields the following:

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

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

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

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

### *5.1.3 Comparison*

Here are some points of comparison between the two approaches.

- A function $a \to b$ is embedded in CDSL as $\mathit{Dp}\ a \to \mathit{Dp}\ b$, a function between representations, and in QDSL as $\mathit{Qt}\ (a \to b)$, 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 **??**). 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 5.2.) 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.

## 5.2 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.

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

Here *sqr* is as before. The above uses

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \\
\mathit{return} :: a \to \mathit{Maybe}\ a \\
(\gg\!=)\ :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b \\
\mathit{maybe} :: b \to (a \to b) \to \mathit{Maybe}\ a \to b
\end{array}
$$

from the Haskell prelude. Type *Maybe* is declared as a monad, enabling the **do** notation, which translates into $(\gg\!=)$ and *return*. 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.

### 5.2.1 CDSL

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

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

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

$$
\begin{array}{l}
\mathit{none}\ \ :: \mathit{Opt}\ a \\
\mathit{return} :: a \to \mathit{Opt}\ a \\
(\gg\!=)\ :: \mathit{Opt}\ a \to (a \to \mathit{Opt}\ b) \to \mathit{Opt}\ b \\
\mathit{option} :: (\mathit{Syn}\ a, \mathit{Syn}\ b) \Rightarrow b \to (a \to b) \to \mathit{Opt}\ a \to b
\end{array}
$$

from the CDSL library. Details of the type *Opt* and the type class *Syn* are explained in Section 5.7. Type *Opt* is declared 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.

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:

$$
\begin{array}{l}
(((\mathit{fst}\ ((x == (0.0))\ ?\ ((((\mathit{False})\ ?\ ((\mathit{True}), (\mathit{False}))), ((\mathit{False})\ ?\ (u \\
\mathit{undef}))), ((\mathit{True}), ((1.0)\ /\ ((x \times ((x \times (1.0)) \times (x \times (1.0)))) \times (x \\
(1.0)) \times (x \times (1.0))))))))))\ ?\ ((\mathit{True}), (\mathit{False})))\ ?\ (((\mathit{fst}\ ((x == (0. \\
((((\mathit{False})\ ?\ ((\mathit{True}), (\mathit{False}))), ((\mathit{False})\ ?\ (\mathit{undef}, \mathit{undef}))), ((\mathit{True} \\
((x \times ((x \times (1.0)) \times (x \times (1.0)))) \times (x \times ((x \times (1.0)) \times (x \times (1.0 \\
((\mathit{snd}\ ((x == (0.0))\ ?\ ((((\mathit{False})\ ?\ ((\mathit{True}), (\mathit{False}))), ((\mathit{False})\ ?\ (u \\
\mathit{undef}))), ((\mathit{True}), ((1.0)\ /\ ((x \times ((x \times (1.0)) \times (x \times (1.0)))) \times (x \\
(1.0)) \times (x \times (1.0)))))))))), \mathit{undef})), (0.0)))
\end{array}
$$

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.

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

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.

### 5.2.2 QDSL

In QDSL, type *Maybe* is represented by itself. Here is the refactored code.

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

$$
\begin{array}{l}
\quad\quad [||\lambda x \rightarrow \textbf{do}\ y \leftarrow \$\$(\mathit{power}'\ (n\ \mathrm{div}\ 2))\ x; \mathit{return}\ (\$\$\mathit{sqr}\ y)||] \\
\quad \textbf{else} \\
\quad\quad [||\lambda x \rightarrow \textbf{do}\ y \leftarrow \$\$(\mathit{power}'\ (n-1))\ x; \mathit{return}\ (x \times y)||] \\
\mathit{power}'' :: \mathit{Int} \rightarrow \mathit{Qt}\ (\mathit{Float} \rightarrow \mathit{Float}) \\
\mathit{power}''\ n = [||\lambda x \rightarrow \mathit{maybe}\ 0\ (\lambda y \rightarrow y)\ (\$\$(\mathit{power}'\ n)\ x)||]
\end{array}
$$

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 $\mathit{power}''\ (-6)$ yields a term of similar complexity to the term yielded by the CDSL. Normalisation by the rules discussed in Section 3 reduces the term to the same form as before, which in turn generates the same C as before.

### 5.2.3 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 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.

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).

## 5.3 The deep embedding

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{data}\ \mathit{Dp}\ a\ \textbf{where}} \\
\quad \mathit{LitB} & :: \mathit{Bool} \rightarrow \mathit{Dp}\ \mathit{Bool} \\
\quad \mathit{LitI} & :: \mathit{Int} \rightarrow \mathit{Dp}\ \mathit{Int} \\
\quad \mathit{LitF} & :: \mathit{Float} \rightarrow \mathit{Dp}\ \mathit{Float} \\
\quad \mathit{If} & :: \mathit{Dp}\ \mathit{Bool} \rightarrow \mathit{Dp}\ a \rightarrow \mathit{Dp}\ a \rightarrow \mathit{Dp}\ a \\
\quad \mathit{While} & :: (\mathit{Dp}\ a \rightarrow \mathit{Dp}\ \mathit{Bool}) \rightarrow (\mathit{Dp}\ a \rightarrow \mathit{Dp}\ a) \rightarrow \mathit{Dp}\ a \rightarrow \mathit{Dp}\ a \\
\quad \mathit{Pair} & :: \mathit{Dp}\ a \rightarrow \mathit{Dp}\ b \rightarrow \mathit{Dp}\ (a, b) \\
\quad \mathit{Fst} & :: \mathit{Dp}\ (a, b) \rightarrow \mathit{Dp}\ a \\
\quad \mathit{Snd} & :: \mathit{Dp}\ (a, b) \rightarrow \mathit{Dp}\ b \\
\quad \mathit{Prim1} & :: \mathit{String} \rightarrow (a \rightarrow b) \rightarrow \mathit{Dp}\ a \rightarrow \mathit{Dp}\ b \\
\quad \mathit{Prim2} & :: \mathit{String} \rightarrow (a \rightarrow b \rightarrow c) \rightarrow \mathit{Dp}\ a \rightarrow \mathit{Dp}\ b \rightarrow \mathit{Dp}\ c \\
\quad \mathit{Arr} & :: \mathit{Dp}\ \mathit{Int} \rightarrow (\mathit{Dp}\ \mathit{Int} \rightarrow \mathit{Dp}\ a) \rightarrow \mathit{Dp}\ (\mathit{Array}\ \mathit{Int}\ a) \\
\quad \mathit{ArrLen} & :: \mathit{Dp}\ (\mathit{Array}\ \mathit{Int}\ a) \rightarrow \mathit{Dp}\ \mathit{Int} \\
\quad \mathit{ArrIx} & :: \mathit{Dp}\ (\mathit{Array}\ \mathit{Int}\ a) \rightarrow \mathit{Dp}\ \mathit{Int} \rightarrow \mathit{Dp}\ a \\
\quad \mathit{Variable} & :: \mathit{String} \rightarrow \mathit{Dp}\ a \\
\quad \mathit{Value} & :: a \rightarrow \mathit{Dp}\ a
\end{array}
$$

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

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

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

## 5.4 Class *Syn*

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class}\ \mathit{Syn}\ a\ \textbf{where}} \\
\multicolumn{2}{l}{\quad \textbf{type}\ \mathit{Internal}\ a} \\
\quad \mathit{toDp} & :: a \rightarrow \mathit{Dp}\ (\mathit{Internal}\ a) \\
\quad \mathit{fromDp} & :: \mathit{Dp}\ (\mathit{Internal}\ a) \rightarrow a
\end{array}
$$

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

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

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

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*.

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

Numbers are made convenient to manipulate via overloading.

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

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

Comparison also benefits from smart constructors.

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

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.

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

## 5.5 Embedding pairs

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

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

This permits us to manipulate pairs as normal, with $(a, b)$, *fst a*, and *snd a*. (Argument $p$ is duplicated in the definition of *fromDp*, which may require common subexpression elimination as discussed in Section 5.1.)

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

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

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.

## 5.6 Embedding undefined

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

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

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

For example,

$$\begin{array}{ll}(/\#) & :: \mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float} \to \mathit{Dp}\ \mathit{Float}\\ x\ /\#\ y & = (y\ .{=}{=}.\ 0)\ ?\ (\mathit{undef}, x\ /\ y)\end{array}$$

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

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

## 5.7 Embedding option

We now explain in detail the *Option* type seen in Section 5.2.

The deep-and-shallow technique cleverly represents deep embeddding $\mathit{Dp}\ (a, b)$ by shallow embedding $(\mathit{Dp}\ a, \mathit{Dp}\ b)$. Hence, it is tempting to represent $\mathit{Dp}\ (\mathit{Maybe}\ a)$ by $\mathit{Maybe}\ (\mathit{Dp}\ a)$, 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 $\mathit{Dp}\ (\mathit{Maybe}\ a)$, 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 $\mathit{Opt}'\ a$, 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 (≫=) 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 5.2.

## 5.8 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 (A
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 -
zipWithVec f (Vec m g) (Vec n h) = Vec (minim m n) (λi → f
sumVec              :: (Syn a, Num a) ⇒ Vec a -
sumVec (Vec n g)    = for n 0 (λi x → x + g i)
```

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

## 5.9 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

$$
\begin{aligned}
&\mathit{norm} \quad :: \mathit{Vec}\ (\mathit{Dp}\ \mathit{Float}) \rightarrow \mathit{Dp}\ \mathit{Float} \\
&\mathit{norm}\ v = \mathit{scalarProd}\ v\ v
\end{aligned}
$$

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.

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

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

For example, if

$$\mathit{blur} :: \mathit{Syn}\ a \Rightarrow \mathit{Vec}\ a \rightarrow \mathit{Vec}\ a$$

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

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

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.

## 6. RELATED WORK

[TODO: Section 7 of ESOP submission]

[TODO: Citations for quotation, macros, early DSLs in Lisp]

## 7. CONCLUSION

[TODO: Section 8 of ESOP submission.]

## 8. 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.

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

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.

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.