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
philip.wadler@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

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

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 Design

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 *Cd* that represents code in C. The top-level function of QFeldspar has the type:

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

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.

instance $FO\ Int$
instance $FO\ Float$
instance $(FO\ a, FO\ b) \Rightarrow FO\ (a, b)$
instance $(FO\ a) \Rightarrow FO\ (Arr\ a)$

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:

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

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:

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

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

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

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 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 -> 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 `float -> float` will only have proper subterms of type `float`, which is indeed true for the normalised term.

[TODO: The above analysis is not correct. We need some-

thing stronger to guarantee that `while`, which has subtypes that are functions, does not actually generate values of higher-type. This is the problem that Shayan raised earlier.]

2.3 Maybe

[TODO: Move example from Section 2 of ESOP submission.]

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

2.4 While

$$\mathit{while} :: (\mathit{FO}\ s) \Rightarrow \mathit{Qt}\ ((s \to \mathit{Bool}) \to (s \to s) \to (s \to s))$$

[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.

Example.

$$\mathit{for} :: (\mathit{FO}\ s) \Rightarrow \mathit{Qt}\ (\mathit{Int} \to s \to (\mathit{Int} \to s \to s) \to s)$$
$$\mathit{for} = [||\lambda n\ x\ b \to \mathit{snd}\ (\mathit{while}\ (\lambda(i,x) \to i < n)\ (\lambda(i,x) \to (i+1, b\ i\ x))\ (0,x))||]$$

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

$$\mathit{fib} :: \mathit{Qt}\ (\mathit{Int} \to \mathit{Int})$$
$$\mathit{fib} = [||\lambda n \to \$\$\mathit{for}\ n\ (\lambda(a,b) \to (b, a+b))\ (0,1)\ |]]$$

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.

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

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

We assume the following primitive operations.

$$\begin{aligned}
&\mathit{arr} &&:: \mathit{FO}\ a \Rightarrow \mathit{Int} \to (\mathit{Int} \to a) \to \mathit{Arr}\ a\\
&\mathit{arrLen} &&:: \mathit{FO}\ a \Rightarrow \mathit{Arr}\ a \to \mathit{Int}\\
&\mathit{arrIx} &&:: \mathit{FO}\ a \Rightarrow \mathit{Arr}\ a \to \mathit{Int} \to a
\end{aligned}$$

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

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

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 (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. 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 Γ 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 Γ. 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 β and commuting reductions, and is the only phase that is crucial for obtaining normal forms that satisfy the subformula property. Phase 3 "garbage collects" unused terms, as in the call-by-need lambda calculus (Maraist et al., 1998). Phase 3 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.

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

Figure 1: Types, Terms, and Values

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

Figure 2: Normal Forms

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 $A \to B$ are $A \to B$ 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 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.*

[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

[TODO: Sections 2 and 3 of ESOP submission]

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.

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.

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.

Phase 1 (let-insertion)

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

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

Phase 2 (symbolic evaluation)

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

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

Phase 3 (garbage collection)

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

Figure 3: Normalisation rules