Practical Normalization by Evaluation for EDSLs

Nachiappan Valliappan
Chalmers University of Technology
Gothenberg, Sweden

Alejandro Russo
Chalmers University of Technology
Gothenberg, Sweden

Sam Lindley
The University of Edinburgh
Edinburgh, United Kingdom

Abstract

Embedded domain-specific languages (eDSLs) are typically implemented in a rich host language, such as Haskell, using a combination of deep and shallow embedding techniques. While such a combination enables programmers to exploit the execution mechanism of Haskell to build and specialize eDSL programs, it blurs the distinction between the host language and the eDSL. As a consequence, extension with features such as sums and effects requires a significant amount of ingenuity from the eDSL designer. In this paper, we demonstrate that Normalization by Evaluation (NbE) provides a principled framework for building, extending, and customizing eDSLs. We present a comprehensive treatment of NbE for deeply embedded eDSLs in Haskell that involves a rich set of features such as sums, arrays, exceptions and state, while addressing practical concerns about normalization such as code expansion and the addition of domain-specific features.

1 Introduction

An embedded domain-specific language (eDSL) [24, 26] is a seamless implementation of a domain-specific language (DSL) as a library in a host language. Haskell is particularly well suited as a host for eDSLs as witnessed by the variety of practical Haskell eDSLs covering domains as diverse as circuit design [12], database querying [25], digital signal processing [6], graphics acceleration [14], and security [37]. Haskell eDSL developers have at their disposal all of Haskell's features such as higher-order functions, extensible syntax, and a rich type-system. It is common to represent programs in an eDSL using a data type that denotes them explicitly, together with compilers and interpreters that manipulate values of this type. Let us consider such a data type $Exp :: * \to *$ parameterized by the type of the expression it denotes. Whereas a value of type *Int* in Haskell denotes an integer *value*, a value of type *Exp Int* denotes an integer *expression*. (We use the words "program" and "expression" interchangeably in the rest of the paper.)

Often, eDSL designers face a choice between either adding complex features to an eDSL or keeping the core eDSL simple and exploiting the host language to construct programs. Should the eDSL support pairs in expressions ($Exp\ (a, b)$), or should it use pairs of expressions ($(Exp\ a, Exp\ b)$)? Should the eDSL support functions ($Exp\ (Int \to Int)$) directly or should it instead rely on Haskell functions ($Exp\ Int \to Exp\ Int$) to build programs? As the complexity increases, it can become difficult to draw a line between the end of the host language and the beginning of the eDSL.

In an eDSL program we may think of a value of type *Int* as a *static* integer that is known at compile-time, and a value of type *Exp Int* as a *dynamic* integer that is known only at runtime. The *stage separation* of values as static and dynamic corresponds to a manual form of *binding-time analysis* in partial evaluation [27], and presents an opportunity to exploit Haskell's execution mechanism to evaluate static computations in an eDSL program. In other words, *static values belong to the host language*, whereas *dynamic values belong to the eDSL*. For example, consider the following implementation of the exponentiation function that receives two integer arguments n and x and returns x^n

```
power₁ :: Int → Exp Int → Exp Int
power₁ n x = if (n ⩽ 1) then x else (x ∗ (power₁ (n − 1))
```

where $(*) :: Exp\ Int \to Exp\ Int \to Exp\ Int$. The type of $power_1$ ensures the first argument is static, and using this function, we can evaluate the expression $power_1\ 3\ x$ for some $x{::}Exp\ Int$ to generate the specialized expression $x{*}x{*}x$. Even though the definition of $power_1$ uses a conditional (**if** ... **then** ... **else**), comparison ($n \leqslant 1$) and function recursion ($power_1\ (n-1)$), these have all been evaluated (by Haskell) and removed in the specialized expression.

Though separation of stages enables the programmer to manually specify those parts of an eDSL program that must be evaluated by Haskell, it also burdens them to maintain multiple variants of the same program. In addition to $power_1$, we may also demand the following variants of the exponentiation function, each corresponding to a different separation of stages for its arguments and result.

```
power₀ :: Int → Int → Int
power₁ :: Int → Exp Int → Exp Int
power₂ :: Exp Int → Int → Exp Int
power₃ :: Int → Int → Exp Int
power₄ :: Exp Int → Exp Int → Exp Int
power₅ :: Int → Exp (Int → Int)
power₆ :: Exp Int → Exp (Int → Int)
power₇ :: Exp (Int → Int → Int)
```

The need for multiple variants can be mitigated to some extent by using an overloading mechanism that automatically lifts *Int* to *Exp Int*, and converts back and forth between some static and dynamic representations, such as $Exp\ (a \to b)$ and $Exp\ a \to Exp\ b$. This is done, for example, in Feldspar [6]. However, conversion between representations does not work for types with multiple introduction forms such as sum types: we cannot convert an expression of type $Exp\ (Either\ a\ b)$ to

Either (*Exp a*) (*Exp b*) as the precise injection may not be known until runtime.

Normalization by Evaluation (NbE) [9], is a program specialization technique that offers a solution to this problem by making specialization automatic, *without the need for manual stage separation.* Using the NbE approach, all variants of the exponentiation function can be recovered from the implementation of $power_7$ :: *Exp* (*Int* → *Int* → *Int*) depending on the availability of the arguments at the site of invocation, i.e., depending on how $power_7$ is used.

Unlike traditional normalization algorithms based on rewriting, NbE bypasses rewriting entirely and instead normalizes an expression by evaluating it using a special interpreter. While NbE techniques for well-typed languages, also known as *typed NbE*, have found a number of theoretical applications such as deciding equivalence of lambda-calculus with sums [3], proving completeness [16], and coherence theorems [11], the practical relevance of typed NbE remains relatively less well-understood. This paper argues that typed NbE is particularly well-suited for specializing eDSL programs in Haskell given the natural reliance on a host language. Indeed, existing techniques for embedding DSLs in Haskell (e.g. the work of Svenningsson and Axelsson [39] on combining deep and shallow embeddings), which may at first seem somewhat ad hoc, can be viewed as instances of NbE.

The contributions of this paper are as follows.

- The first comprehensive practical treatment of NbE for eDSLs.
- A coherent combination of NbE techniques to deal with a rich set of features such as sums, arrays, exceptions, and state—and in particular—an account of their interaction in a modular and extensible manner.
- Practical extensions of standard NbE techniques to implement a richer set of domain-specific equations, and variations that control unnecessary code expansion.
- Examples showing that NbE provides a principled alternative to ad hoc techniques that combine deep and shallow embedding to implement fusion for functions, loops and arrays in an eDSL.

2 Normalizing EDSL programs

This section showcases our implementation with examples of normalizing eDSL programs using NbE. We begin with standard examples of normalizing the exponentiation function and array operations, and then show examples that illustrate normalization of programs that contain side-effects, branching, and an intricate interaction between them.

Normalization is performed by a function *norm* :: *Rf a* ⇒ *Exp a* → *Exp a*, and the result is observed by printing the resulting expression. The type class constraint *Rf* limits the type of an expression to the types recognized by the eDSL, and is defined along with the data type *Exp* in the next section. The name *Rf* is short for reifiable. For convenience, we do not program with the constructors of *Exp* directly, and instead use derived combinators and "smart constructors" that provide a programming interface to the eDSL. This means that the observed result of normalizing an eDSL program is that of its internal representation, and may not directly resemble the surface program.

We make use of a form of higher-order abstract syntax (HOAS) [33] in order to repurpose the binding features of Haskell in the eDSL. Thus the *Lam* constructor takes a Haskell function on expressions as its argument (Figure 2). Our focus here is on practical implementation so we do not concern ourselves with subtleties such as ruling out so called exotic terms or exotic types [5]. Nevertheless, it is a routine exercise to adapt our approach to use standard techniques to preclude such infelicities, for instance by using an abstract type to hide the concrete type constructors [35] or moving to a tagless representation [5, 13] whereby the smart constructors are first-class.

Normalizing exponentiation. Consider again the exponentiation function from the previous section, and suppose that it is implemented as follows.

```
power :: Exp (Int → Int → Int)
power = lam $ λn → lam $ λx → rec n (f x) 1
  where f x = lam $ λ_ → lam $ λacc → (x ∗ acc)
```

This implementation corresponds to the $power_7$ variant, and is implemented using expression combinators: *lam* :: (*Exp a* → *Exp b*) → *Exp* (*a* → *b*) is a lambda expression combinator and *rec* :: *Exp Int* → *Exp* (*Int* → *a* → *a*) → *Exp a* → *Exp a* is a primitive recursion combinator such that *rec n g a* is equivalent to *g* 1 (*g* 2 (...(*g n a*))).

An expression of a function type can be applied using the combinator *app* :: *Exp* (*a* → *b*) → *Exp a* → *Exp b* as *app* (*power* 3), where the argument is a numeral expression 3 :: *Exp Int*. We can normalize this expression in the Haskell interpreter GHCi using the function *norm* as follows.

```
∗NbE.OpenNbE> norm (app power 3)
λx.(x ∗ (x ∗ x))
```

The result is a pretty-printed representation of the expression syntax of the *Exp* data type—here a function that returns the cube of its argument. Observe that the result is slightly more optimal than that of a textbook partial evaluator that returns $\lambda x.(x * (x * (x * 1)))$ by unrolling the recursion. This optimization is a simple instance of NbE's ability to aggressively reduce arithmetic expressions even in the presence of unknown values—we return to this in Section 6.

Note here that the specialization of *power* is automatic and there was no need to manually separate the stages of arguments as static (*Int*) and dynamic (*Exp Int*). We consider the entire expression to be dynamic, and leave it to the normalizer to identify the best specialization strategy.

As another example, consider normalizing an invocation of *power* with flipped arguments using a utility function *flip′*.

```
∗NbE.OpenNbE> norm (app (flip′ power) 3)
λn.(Rec n (λy.λacc.(3 ∗ acc)) 1)
```

Observe that the (expected) definition of *flip′* has been removed in the result, producing a more optimal function.

Normalizing array operations. Normalization can be used to achieve fusion of operations over arrays such as map and fold [32]. We consider immutable *pull arrays* [40] in our eDSL, and an expression of the array type is denoted by the type *Exp* (*Arr a*), where *a* denotes the type of the elements in the array. The map and fold operations are given by derived combinators, whose types and corresponding fusion laws are given as below.

```
mapArr :: Exp (a → b) → Exp (Arr a) → Exp (Arr b)
foldArr :: Exp (b → a → b) → Exp b → Exp (Arr a) → Exp b
  -- fusion laws:
  -- 1. mapArr f (mapArr g arr) = mapArr (f . g) arr
  -- 2. foldrArr f x (mapArr g arr) = foldArr (f . g) x arr
```

These combinators are derived using simpler expression constructors, that for e.g., create an array (*NewArr*), or perform recursion (*Rec*), and the fusion laws follow from the equations that specify their behavior.

Using these combinators, we may implement a function (expression) *mapMap* that maps twice over a given argument array, first with the function (+1), and then with (+2).

```
mapMap :: Exp (Arr Int → Arr Int)
mapMap = lam $ λarr →
  mapArr (lam (+2)) (mapArr (lam (+1)) arr)
```

By the first fusion law, this expression is equivalent to one that maps once with (+3) as: *mapArr* (*lam* (+3)) *arr*. Normalizing *mapMap* returns a new array which has the same length as the argument array *arr*, and whose elements are the elements of *arr* incremented by 3.

```
∗NbE.OpenNbE> norm mapMap
λarr.(NewArr (LenArr arr) (λi.(arr ! i) + 3))
```

The result is indeed the expression constructed by applying the derived combinator *mapArr* as *mapArr* (*lam* (+3)) *arr*. Besides map fusion, NbE also eliminates the function composition from the fused function (+2) ∘ (+1) and performs constant folding to obtain (+3).

To illustrate the second fusion law, consider the following function, *mapFold*, that first maps (+2) over a given array and then computes the sum of the result using *foldArr*.

```
mapFold :: Exp (Arr Int → Int)
mapFold = lam (λarr → foldArr go 0 (mapArr (lam (+2)) arr))
  where go = lam (λacc → lam (λx → acc + x))
```

By the second fusion law, this expression is equivalent to one which simply folds the entire array as: *foldArr* (*lam* (*λacc* → *lam* (*λx* → *acc* + *x* + 2))) 0 *arr*. Normalizing *mapFold* yields the following result.

```
∗NbE.OpenNbE> norm mapFold
λarr.(Rec (LenArr arr) (λi.λacc.acc + (arr ! i) + 2) 0)
```

The normalized function receives an argument array, and performs recursion over its length to compute the sum of its elements, each of which has been incremented by 2.

Normalizing branching programs. Branching programs, or programs that perform a case analysis over a value of a sum type, complicate normalization. The difficulty arises from the fact that the outcome of a case analysis over an unknown value cannot be determined at normalization time. NbE offers a modular solution to address this difficulty and achieve normalization for branching programs, as we shall see later in Section 4.

Consider the following branching program, *prgBr*, that illustrates a scenario where map fusion on arrays is interrupted by a case analysis on an unknown value.

```
prgBr :: Exp (Either Int Int → Arr Int → Arr Int)
prgBr = lam $ λscr → lam $ λarr →
  mapArr (lam (+1)) $ case′ scr
    (lam $ λx → mapArr (lam (+x)) arr)
    (lam $ λy → arr)
```

It performs a case analysis using the combinator *case′* :: *Exp* (*Either a b*) → *Exp* (*a* → *c*) → *Exp* (*b* → *c*) on the argument *scr* (an unknown value), and if the left injection is found with an integer *x*, it returns an array that increments elements of *arr* by *x*, else *arr* is returned as found otherwise. The array returned by *case′* is further incremented by 1.

Normalizing *prgBr* yields the following result.

```
∗NbE.OpenNbE> norm prgBr
(λscr.(λarr.NewArr (LenArr arr)
  (λi.Case scr (λx.((arr ! i) + x + 1)) (λy.((arr ! i) + 1)))))
```

The normalized function returns a new array whose elements are given by performing case analysis on *scr*. Observe that the effect of *mapArr* (*lam* (+1)) in *prgBr* has been fused with the application of *mapArr* in the first branch, and left unaltered in the second branch. The normalized program delays case analysis on *scr* to the point at which it is required, thus avoiding the materialisation of an intermediate array.

Normalizing stateful programs. Programs with side-effects can be also normalized using NbE, and the following example illustrates such a program that writes to and reads from a global state in a *State* monad.

```
prgSt :: Exp (Arr Int → State (Arr Int) Int)
prgSt = lam $ λarr →
  put (mapArr (lam (+2)) arr)
  ≫st put (mapArr (lam (+1)) arr
  ≫st get ≫=st (lam $ λarr′ → returnst (ixArr arr′ 0)))
```

The program *prgSt* receives an integer array *arr*, and returns an *Int* by writing to and reading from (using combinators *get* and *put*) a global state that contains an array of type *Arr Int*. Precisely, it performs the following actions (sequenced using monadic combinators $\gg_{st}$ and $\ggg=_{st}$):

- writes the result of mapping over *arr* with (+2)
- writes the result of mapping over *arr* with (+1)
- reads the array from state, and returns its first element

The combinators *put*, *get*, $\gg_{st}$, $\ggg=_{st}$ and $return_{st}$ have their expected types lifted to expressions. For example, $put :: Exp\ s \to Exp\ (State\ s\ ())$ and $get :: Exp\ (State\ s\ s)$.

Normalizing *prgSt* yields the following result.

```
*NbE.OpenNbE> norm prgSt
λarr.(Get ≫= λs.(Put (NewArr (LenArr arr) (λi.(arr ! i) + 1))
   ≫ return ((arr ! 0) + 1)))
```

The resulting program puts a new array that contains the elements of the original array incremented by 1, and returns the head of the original array, also incremented by 1. The first *put* operation in *prgSt* is removed as it is overwritten by the subsequent *put*. Similarly, the operation *get* and the intermediate array *arr′* in *prgSt* are also removed, as the array in the state is known locally from the previous *put* operation. The *Get* in the result is redundant as the state *s* is never used. This *Get* is introduced by the normalizer as a consequence of η-expansion (see Section 5). We show later, in Section 6, how such redundancy in generated code can be eliminated by disabling η-expansion.

Normalizing branching stateful programs. The presence of side-effects and branching in the same language creates subtle interactions between the primitives that must be considered when implementing normalization. To illustrate that our NbE procedure can also be applied seamlessly to their combination, we consider the following program that combines the last two examples.

```
prgBrSt :: Exp (Either Int Int → Arr Int → State (Arr Int) Int)
prgBrSt = lam $ λscr → Lam $ λarr →
  put (mapArr (lam (+1)) arr)
    ≫=_st (case′ scr
      (lam $ λx → put (mapArr (Lam (+x)) arr))
      (lam $ λy → return unit))
    ≫_st get ≫=_st (lam (λarr′ → return_st (ixArr arr′ 0))))
```

Unlike in *prgSt*, the first *put* here cannot be eliminated as the second branch does not have a subsequent *put*. Moreover, elimination of *get* here is less straightforward as we cannot readily determine the value of the array in the state.

Normalizing *prgBrSt* yields the following result.

```
*NbE.OpenNbE> norm prgBrSt
λscr.(λarr.(Get ≫= (λs.(Case scr of
  (λx.(Put (NewArr (LenArr arr) (λi.(arr ! i) + x))
     ≫ Return ((arr ! 0) + x)))
  (λy.(Put (NewArr (LenArr arr) (λi.(arr ! i) + 1))
     ≫ Return ((arr ! 0) + 1)))))))
```

The resulting program pattern matches on *scr*, performs appropriate *put* operations and returns the expected result individually on each branch. The first *put* operation, i,e., *put* (*mapArr* (*lam* (+1)) *arr*) is discarded in the first branch but preserved in the latter!

```
  -- Expressions, neutrals and normal forms
data Exp a where ...
data Ne a where ...
data Nf a where ...
  -- Embedding functions
embNe :: Ne a → Exp a
embNf :: Nf a → Exp a
  -- NbE semantics
class Rf a where
  type Sem a :: *
  reify   :: Sem a → Nf a
  reflect :: Ne a → Sem a
  -- Evaluation function
eval   :: Rf a ⇒ Exp a → Sem a
  -- Normalization function
norm :: Rf a ⇒ Exp a → Exp a
norm = embNf ∘ reify ∘ eval
```

Figure 1. Components of NbE

3 NbE for an eDSL Core

NbE is the process of *evaluating*, or interpreting, expressions of a language in a semantic domain and then obtaining normal forms by *reifying*, or extracting, normal forms from values in the semantic domain. The key idea behind NbE is to leverage an (often non-standard) evaluator implemented in the host language to normalize expressions in the object language—hence the name normalization *by* evaluation.

Figure 1 summarizes the components of NbE in our implementation. The object language is defined by the expression data type *Exp*, and its normal forms are defined by *Nf* and *Ne* (a subcategory of normal forms called *neutrals*). Unlike a traditional evaluator, an NbE evaluator interprets expressions in a semantic domain that is carefully chosen such that normal forms can be reified from it. The type class *Rf* specifies the requirements of such a semantic domain.

In the class *Rf*, the type family *Sem* maps types in the object language to the Haskell types that interpret them. The definition of *Rf* requires that an interpretation of a type be chosen such that we can also implement the functions *reify* and *reflect*. The function *reify* performs reification, and the function *reflect* performs a process known as *reflection*.

```
data Exp a where
  Var  :: Rf a ⇒ String → Exp a
  Lift :: Base a ⇒ a → Exp a
  Lam  :: (Rf a, Rf b) ⇒ (Exp a → Exp b) → Exp (a → b)
  App  :: (Rf a, Rf b) ⇒ Exp (a → b) → Exp a → Exp b
  Unit :: Exp ()
  Pair :: (Rf a, Rf b) ⇒ Exp a → Exp b → Exp (a, b)
  Fst  :: (Rf a, Rf b) ⇒ Exp (a, b) → Exp a
  Snd  :: (Rf a, Rf b) ⇒ Exp (a, b) → Exp b
  Mul  :: Exp Int → Exp Int → Exp Int
  Add  :: Exp Int → Exp Int → Exp Int
  Rec  :: (Rf a) ⇒ Exp Int
         → Exp (Int → a → a) → Exp a → Exp a
```

Figure 2. Basic core expression language

Reflection inserts neutral expressions into the semantic domain, and is used to evaluate free variables whose values are unknown. Reflection is crucial to reifying functions: to convert a semantic function to a syntactic one, we apply it to a semantic value given by the reflecting the argument variable of the syntactic function. Our syntax for functions calls for a slightly different treatment, as we shall see shortly.

In this section, we discuss the implementation of NbE for an eDSL core language that is defined by the *Exp* data type. This language is based on a simply-typed lambda calculus (STLC) with product and base types, extended with primitive recursion and simple arithmetic operations. We later extend it further with array and sum types (Section 4), exception and state effects (Section 5), and other uninterpreted primitives (Section 6). These features have been chosen to illustrate the practical applicability, extensibility, and customizability of NbE to a class of functional eDSLs like Feldspar [6], Haski [41], and others found in eDSL literature [4, 39].

Figure 2 summarizes the pure fragment of the core expression syntax. It consists of expression constructors for unknowns (*Var*), constants (*Lift*), functions (*Lam*, *App*), products (*Pair*, *Fst*, *Snd*), arithmetic operations (*Mul*, *Add*), and primitive recursion (*Rec*). The constructor *Var* allows us to insert unbound free variables, and *Lift* allows us to lift constant values of primitive base types (identified by the type class *Base*) directly to expressions. For example, instances *Base Int* and *Base String* allow us to lift integers and strings to expressions of type *Exp Int* and *Exp String* respectively.

Function and product types. The NbE algorithm for function and product types applies the equations below by orienting them from left to right, and the generated normal forms correspond to β-short η-long form.

```
f :: Exp (a → b) ≈ Lam (App f)
App (Lam f) e    ≈ f e
p :: Exp (a * b)  ≈ Pair (Fst p) (Snd p)
Fst (Pair e1 e2)  ≈ e1
Snd (Pair e1 e2)  ≈ e2
```

The type directed equations, or η-laws, specify the structure of the resulting normal forms, and the reduction laws, or β-laws, specify how expressions should be reduced. We begin by defining neutral and normal forms for these types.

```
data Ne a where
  NVar :: (Rf a) ⇒ String → Ne a
  NApp :: (Rf a) ⇒ Ne (a → b) → Nf a → Ne b
  NFst :: (Rf b) ⇒ Ne (a, b) → Ne a
  NSnd :: (Rf a) ⇒ Ne (a, b) → Ne b
data Nf where
  NUnit :: Nf ()
  NLam  :: (Rf a, Rf b) ⇒ (Exp a → Nf b) → Nf (a → b)
  NPair :: (Rf a, Rf b) ⇒ Nf a → Nf b → Nf (a, b)
```

Intuitively, neutral expressions denote expressions whose reduction is stuck at unknowns, and normal forms denote value expressions. Note that the normal form constructor for functions, *NLam*, receives an argument of type *Exp a* → *Nf b* instead of the more restrictive type *Nf a* → *Nf b*. This is to allow the *syntactic embedding*—i.e, without invoking functions that involve semantics, such as *eval* or *reify*—of normal forms to expressions by mapping *NLam* to *Lam*, which would not be possible with the latter option.

The semantic domain for product and function types are readily available in Haskell, so we simply interpret them by their Haskell counterparts by defining instances of *Rf* as follows.

```
instance Rf () where
  type Sem () = ()
  reify   () = NUnit
  reflect _  = ()
instance (Rf a, Rf b) ⇒ Rf (a, b) where
  type Sem (a, b) = (Sem a, Sem b)
  reify   p = NPair (reify (fst p)) (reify (snd p))
  reflect n = (reflect (NFst n), reflect (NSnd n))
instance (Rf a, Rf b) ⇒ Rf (a → b) where
  type Sem (a → b) = Sem a → Sem b
  reify f   = NLam (reify ∘ f ∘ eval)
  reflect n = λy → reflect (NApp n (reify y))
```

The implementation of functions *reify* and *reflect* is achieved by converting from and to Haskell values. To reify a pair *p* :: (*Sem a*, *Sem b*), we construct a normal form using the constructor *NPair*, whose arguments are obtained by recursively reifying the projections of *p*. To reflect a neutral *n* :: *Ne* (*a*, *b*), we construct a pair whose components are obtained by recursively reflecting the projections of *n* using neutral constructors *NFst* and *NSnd*. On the other hand, to reify a function *f* :: *Sem a* → *Sem b*, we evaluate the expression argument[1]

[1]Traditionally, reflection is sufficient since the argument in *Lam* is a variable, but our formulation demands evaluation since it can be any expression.

provided by the constructor *NLam* and recursively reify its application to f, and to reflect a neutral $n :: Ne\ (a \to b)$, we recursively reflect the application of n using the constructor *NApp* with the reification of the semantic argument y.

Evaluation resembles a standard evaluator, with the exception of the *Var* and *Lam* cases, as witnessed below.

```
eval (Var x)     = reflect (NVar x)
eval Unit        = ()
eval (Lam f)     = λy → eval (f (embNf (reify y)))
eval (App f e)   = (eval f) (eval e)
eval (Pair e e′) = (eval e, eval e′)
eval (Fst e)     = fst (eval e)
eval (Snd e)     = snd (eval e)
```

For the *Var* case, we use reflection to insert the neutral *NVar x* into the semantics, and for the *Lam* case, we recursively evaluate the application of f to an expression obtained by reifying and embedding the semantic argument y. Reflection converts the unknown *Var x* to a semantic value, which when reified, has the effect of η-expansion [9]. For example, evaluating an unknown *Var* "x" :: *Exp* $(() \to ())$ returns its reflection $\lambda y \to ()$, which when reified yields the normal form *NLam* $(\lambda y \to NUnit)$, where η-expansion has been applied for both the function and unit types.

Base types. The expression syntax can be freely extended with values of base type by defining new instances of the type class *Base*. To extend NbE with base types, we define normal forms of these types as either neutrals or values of the type, and use a semantic domain that resembles this definition. We illustrate this for the types *Int* and *String* below.

```
data Nf where ...
  NUp   :: Base a ⇒ Ne a → Nf a
  NLift :: Base a ⇒ a → Nf a
instance Rf Int where
  type Sem Int = Either (Ne Int) Int
  reify x      = either NUp NLift x
  reflect n    = Left n
instance Rf String where
  type Sem Int = Either (Ne String) String
    -- similar to above
```

We extend the definition of normal forms with constructors for embedding neutrals (*NUp*) and values (*NLift*) of base types. For defining an instance *Rf Int*, we use the type *Either* (*Ne Int*) *Int* as the semantic domain—and similarly for *String*. Reification replaces *Left* by *NUp* and *Right* by *NLift*, while reflection embeds a neutral into the semantic domain using *Left*.

In the absence of primitives that return a value of base type, such as *String*, we need not perform any further modifications. For base types with primitives, such as *Int*, however, we must also extend evaluation and the definition of neutrals to accommodate them.

```
data Exp a where ...
  NewArr :: Rf a ⇒ Exp Int → Exp (Int → a) → Exp (Arr a)
  LenArr :: Rf a ⇒ Exp (Arr a) → Exp Int
  IxArr  :: (Rf a) ⇒ Exp (Arr a) → Exp Int → Exp a
  Inl    :: (Rf a) ⇒ Exp a → Exp (Either a b)
  Inr    :: (Rf b) ⇒ Exp b → Exp (Either a b)
  Case   :: (Rf a, Rf b, Rf c) ⇒ Exp (Either a b)
      → Exp (a → c) → Exp (b → c) → Exp c
```

Figure 3. Extension with arrays and sums

For a simple treatment of integer expressions, let us suppose that we would like to normalize them using the following equations.

```
Add (Lift x) (Lift y) ≈ Lift (x + y)
Mul (Lift x) (Lift y) ≈ Lift (x ∗ y)
```

These equations specify that addition and multiplication must be performed when both the operands are available as lifted integer values. In the absence of one of the operands, such as in *Add* (*Lift* 2) (*Var* "x"), the expression cannot be reduced further, and must be considered to be in normal form.

To implement these equations, we extend the definition of neutrals for "stuck" applications *Add* and *Mul* as follows:

```
data Ne a where ...
  NAdd1 :: Ne Int → Int    → Ne Int
  NAdd2 :: Int    → Ne Int → Ne Int
  NAdd  :: Ne Int → Ne Int → Ne Int
    -- similarly NMul1, NMul2 and NMul
```

Following this, evaluation can be implemented as follows using semantic functions *add′* and *mul′*.

```
eval (Add e e′) = add′ (eval e) (eval e′)
eval (Mul e e′) = mul′ (eval e) (eval e′)
```

The functions *add′*, *mul′* :: *Sem Int* → *Sem Int* → *Sem Int* are implemented by performing the corresponding operation when both the right injections are available, and constructing neutrals otherwise.

4 NbE for Arrays and Sums

Figure 3 summarizes the extension of the core language with array and sum types. The type *Exp* (*Arr a*) denotes an *a* array expression indexed by integers, and the type *Exp* (*Either a b*) denotes a sum expression of type *Either a b*. The array operation *NewArr* constructs a new array, *LenArr* computes the length of an array, and *IxArr* indexes into an array. Sum types are formulated in the usual way with injections (*Inl* and *Inr*) and case analysis (*Case*).

4.1 Arrays

Arrays satisfy the following equations.

```
arr :: Exp (Arr a) ≈ NewArr (LenArr arr) (Lam (IxArr arr))
LenArr (NewArr n f) ≈ n
IxArr (NewArr n f) k ≈ f k
```

The first is η-expansion for arrays, and the second and third are reductions for *LenArr* and *IxArr*.

Neutral and normal forms are defined by placing stuck applications of *LenArr* and *IxArr* in neutrals, and an array construction using *NewArr* in normal forms.

```
data Ne a where ...
  NLenArr :: (Rf a) ⇒ Ne (Arr a) → Ne Int
  NIxArr :: (Rf a) ⇒ Ne (Arr a) → Nf Int → Ne a
data Nf a where ...
  NNewArr :: (Rf a) ⇒ Nf Int
            → (Exp Int → Nf a) → Nf (Arr a)
```

The semantic domain for arrays, defined by *SArr* below, is given by a refinement of a shallow embedding of arrays in Haskell (called *vectors* in Feldspar [6]).

```
data SArr a where
  SNewArr :: Sem Int → (Exp Int → a) → SArr a
instance (Rf a) ⇒ Rf (Arr a) where
  type Sem (Arr a) = SArr (Sem a)
  reify (SNewArr n f) = NNewArr (reify n) f
  reflect n              = SNewArr
    (reflect (NLenArr n))
    (reflect ∘ NIxArr n ∘ reify ∘ eval)
```

The constructor *SNewArr* constructs a semantic array from the length of an array, given by a semantic integer *Sem Int*, and a function *Exp Int* $\rightarrow$ *a* that returns elements of the array for a given index expression. Reification converts a semantic array constructed using *SNewArr* to a syntactic one in normal form constructed using *NNewArr*. Reflection, on the other hand, inserts a neutral *n* :: *Exp* (*Arr a*) into semantics by constructing a semantic array with the same length and same elements as *n*.

Evaluation is extended to arrays by interpreting *NewArr* as *SNewArr*, and the array operations *IxArr* and *LenArr* by extracting the appropriate components of *SNewArr*.

```
eval (NewArr n f) = SNewArr (eval n) f
eval (IxArr arr i)  = let (SNewArr _ f) = eval arr in f i
eval (LenArr arr)  = let (SNewArr n _) = eval arr in n
```

4.2 Sum types

Equations and normal forms. Expressions of sum types are given the following standard equations.

```
e :: Exp (a + b)   ≈ Case e Inl Inr
Case (Inl e) f g ≈ f e
Case (Inr e) f g ≈ g e
F (Case e g h)   ≈ Case e (F ∘ g) (F ∘ h)
```

The first equation specifies a restricted η-expansion for sums. The second and third equations are the standard β-rules for sums. The last equation is a *commuting conversion*, where the function *F* denotes an elimination context, which arises from a more general η-rule [29] and enables more opportunities to apply the β-rules [34]. This equation is further explained in Appendix A.1. Normal forms for sums comprise injections *and case analysis*.

```
data Nf a where ...
  NInl   :: (Rf a)
      ⇒ Nf a → Nf (Either a b)
  NInr   :: (Rf b)
      ⇒ Nf b → Nf (Either a b)
  NCase :: (Rf a, Rf b, Rf c) ⇒ Ne (Either a b)
      → (Exp a → Nf c) → (Exp b → Nf c) → Nf c
```

Unlike stuck applications of eliminators, such as *NFst* and *NSnd*, that we class as neutral, we classify a stuck application of *NCase* as a normal form. This has to do with the implementation of the commuting conversions for sums.

Classifying *NCase* as neutral does not force commuting reductions, and may cause case analysis to prevent reductions by harboring introduction forms. For example, defining *NCase* under neutrals would deem the following expression to be neutral, and thus normal (via *NUp*).

```
NApp (NCase (NVar "x") (NLam id) (NLam id)) (NLift 1)
```

Placing *NCase* in normal forms, on the other hand, forces this expression to be reduced further as below since a normal form of function type cannot be applied.

```
NCase (Var "x") (NLam $ λ_ → Lift 1) (NLam $ λ_ → Lift 1)
```

Semantic domain for sums. It is tempting to interpret sum types by their Haskell counterpart, i.e., *Sem* (*Either a b*) = *Either* (*Sem a*) (*Sem b*). But this interpretation is insufficient for NbE, and does not support reflection. For example, what should be the reflection of the unknown *Var* "x" :: *Exp* (*Either* () ())? We cannot make a choice over the *Left* or *Right* injection! To solve this dilemma, we define a semantic domain that captures branching over neutrals (which subsume unknowns), and use that to interpret sums.

```
data MDec a where
  Leaf    :: a → MDec a
  Branch :: (Rf a, Rf b) ⇒ Ne (Either a b)
      → (Exp a → MDec c) → (Exp b → MDec c) → MDec c
instance Monad MDec where ...
instance (Rf a, Rf b) ⇒ Rf (Either a b) where
  type Sem (Either a b) = MDec (Either (Sem a) (Sem b))
  reify (Leaf (Left x))   = NInl (reify x)
  reify (Leaf (Right x)) = NInr (reify x)
  reify (Branch n f g)   = NCase n (reify ∘ f) (reify ∘ g)
  reflect n = Branch n
    (Leaf ∘ Left ∘ eval)
    (Leaf ∘ Right ∘ eval)
```

Intuitively, the data type *MDec* defines a decision tree (monad) that prevents us from having to make a choice during reflection. Unlike a value of type *Either* (*Sem a*) (*Sem b*), a value of type *MDec* (*Either* (*Sem a*) (*Sem b*)) can be constructed using the *Branch* constructor without making a choice. The *Branch* constructor requires us to handle both possible injections, and is the semantic equivalent of the normal form *NCase*—as witnessed by the implementation of *reify*.

Evaluating case analysis. The introduction of sum types causes a subtle problem for evaluation: consider the following expression of type *Exp Int*.

```
Case (Var "x") (Lam $ λ_ → Lift 1) (Lam $ λ_ → Lift 2)
```

While irreducible (and representable as a normal form), the semantic domain for integers, i.e., *Either* (*Ne Int*) *Int*, has no room for its interpretation! How should this *Case* expression of type *Expr Int* be evaluated as a value of type *Either* (*Ne Int*) *Int*? We proceed to adapt our interpretation of *Int* (and similarly with *String*) as follows.

```
instance Rf Int where
  type Sem Int = MDec (Either (Ne Int) Int) ...
instance Rf String where
  type Sem Int = MDec (Either (Ne String) String) ...
```

In short, we place the decision tree monad *MDec* on top of the original interpretation of *Int* allowing room for constructing case trees in the semantics. The problematic integer expression from above can now be evaluated to:

```
Branch (NVar "x") (λ_ → Right 1) (λ_ → Right 2)
```

Reification and reflection can be implemented easily by adapting our previous implementation to deal with *MDec* along the lines of the *Rf* (*Either a b*) instance.

Following this change to the interpretation, we proceed with evaluation as below using a utility function *run*::*Rf a* ⇒ *MDec* (*Sem a*) → *Sem a* that can be implemented by induction on the type parameter *a*—which rephrases the branches of the decision tree as semantic ones.

```
eval (Inl e)      = Leaf (Left (eval e))
eval (Inr e)      = Leaf (Right (eval e))
eval (Case s f g) = let s′ = eval s; f′ = eval f; g′ = eval g
  in run (fmap (either f′ g′) s′)
```

Not all type constructors suffer from this problem. In particular, all type constructors with a single introduction form and a corresponding η-rule (such as functions, products, and arrays) avoid this problem as we may perform η-expansion of the *Case* expression, followed by commuting conversions, to represent the value in the semantic domain. For example, the following expression of type *Exp* (*Int*, *Int*)

```
Case (Var "x") (Lam $ λ_ → Var "y") (Lam $ λ_ → Var "z")
```

can be η-expanded and then two commuting conversions applied to give

```
Pair
  (Case (Var "x")
     (Lam $ λ_ → Fst (Var "y")) (Lam $ λ_ → Fst (Var "z")),
   Case (Var "x")
     (Lam $ λ_ → Snd (Var "y")) (Lam $ λ_ → Snd (Var "z")))
```

which is interpreted in the semantic demain as a pair of integer expressions:

```
(Branch (NVar "x")
   (λ_ → NFst (NVar "y")) (λ_ → NFst (NVar "z")),
 Branch (NVar "x")
   (λ_ → NSnd (NVar "y")) (λ_ → NSnd (NVar "z")))
```

This means that we need only to refine our interpretation of *Int* and *String*, where we lack a combination of a single introduction form accompanied by a corresponding η-rule.

The above treatment of sums is sound and often suffices in practice, but it does not capture all natural equations for sums. In Section 6 we outline how to augment our implementation to eliminate repeated and redundant case splits.

```
data Exp a where ...
  Throw     :: Rf a ⇒ Exp String → Exp (Err a)
  Catch     :: Rf a ⇒ Exp (Err a)
     → Exp (String → Err a) → Exp (Err a)
  Return_err :: Rf a ⇒ Exp a → Exp (Err a)
  Bind_err   :: (Rf a, Rf b) ⇒ Exp (Err a)
     → Exp (a → Err b) → Exp (Err b)
```

(a) Exceptions

```
data Exp a where ...
  Get       :: Rf s ⇒ Exp (State s s)
  Put       :: Rf s ⇒ Exp s → Exp (State s ())
  Return_st :: Rf a ⇒ Exp a → Exp (State s a)
  Bind_st   :: (Rf a, Rf s) ⇒ Exp (State s a)
     → Exp (a → State s b) → Exp (State s b)
```

(b) State

Figure 4. Extension with exception and state effects

5 NbE for Monadic Effects

Figure 4 summarizes the extension of the expression syntax respectively with exceptions and state formulated as monadic types. Exceptions consist of a throw operation (*Throw*) to throw string exceptions, a catch operation (*Catch*) to handle exceptions, along with the return (*Return*$_{err}$), and bind (*Bind*$_{err}$) of the monadic type *Err*. Stateful computations are formulated similar to the State monad in Haskell, and consists of a get operation (*Get*) to retrieve the state, a put operation (*Put*) to overwrite the state, along with the return (*Return*$_{st}$), and bind (*Bind*$_{st}$) of the monadic type *State s*.

Both monadic types (denoted *M*) are subject to the following equations, typically called the *monad laws*.

$$\begin{aligned}
&m :: Exp\ (M\ a) && \approx Bind\ m\ Return\\
&Bind\ (Return\ x)\ f && \approx f\ x\\
&Bind\ (Bind\ e_1\ f)\ g && \approx Bind\ e_1\ (Lam\ (\lambda x \to Bind\ (f\ x)\ g))
\end{aligned}$$

The first equation is η-expansion for monads, the second β-reduction, and the third a commuting conversion that arranges *Bind* operations in a right-associative chain.

5.1 Exceptions

As well as the monad laws, exception computations also obey the following equations.

$$\begin{aligned}
&m :: Exp\ (Err\ a) && \approx Catch\ m\ Throw\\
&Catch\ (Throw\ e)\ f && \approx f\ e\\
&Catch\ (Return\ x)\ f && \approx Return\ x
\end{aligned}$$

The first equation is η-expansion and the second and third equations are β-reductions for exceptions.

Notice here that there is a contention between two η laws: one for the *Err* monad and one specific to exceptions. What should be the η-expanded form of an expression $e :: Exp\ (Err\ a)$? We must make a choice here, and we choose $Catch\ (Bind_{err}\ e\ Return)\ Throw$, where we first apply the η-rule for monads, and then apply the one for exceptions. Our normal forms reflect this choice using a normal form constructor *NTryUnless* that denotes a fusion of $Bind_{err}$ and *Catch* in normal form [8].

$$\begin{aligned}
&\textbf{data}\ Nf\ a\ \textbf{where} \ldots\\
&\quad NRetErr && :: Rf\ a \Rightarrow Nf\ a \to Nf\ (Err\ a)\\
&\quad NThrow && :: Nf\ String \to Nf\ (Err\ a)\\
&\quad NTryUnless && :: (Rf\ a, Rf\ b) \Rightarrow Ne\ (Err\ a)\\
&\qquad && \to (Exp\ a \to Nf\ (Err\ b))\\
&\qquad && \to (Exp\ String \to Nf\ (Err\ b)) \to Nf\ (Err\ b)
\end{aligned}$$

The constructors *NRetErr* and *NThrow* are the normal form counterparts of $Return_{err}$ and *Throw*.

The semantic domain is defined by a data type *MErr* that closely parallels the structure of normal forms.

$$\begin{aligned}
&\textbf{data}\ MErr\ a\ \textbf{where}\\
&\quad SRetErr && :: Rf\ a \Rightarrow a \to MErr\ a\\
&\quad SThrow && :: Nf\ String \to MErr\ a\\
&\quad STryUnless && :: (Rf\ a, Rf\ b) \Rightarrow Ne\ (Err\ a)\\
&\qquad && \to (Exp\ a \to MErr\ b)\\
&\qquad && \to (Exp\ String \to MErr\ b) \to MErr\ b\\
&\textbf{instance}\ (Rf\ a) \Rightarrow Rf\ (Err\ a)\ \textbf{where}\\
&\quad \textbf{type}\ Sem\ (Err\ a) && = MErr\ (Sem\ a)\\
&\quad reify\ (SRetErr\ x) && = NRetErr\ (reify\ x)\\
&\quad reify\ (SThrow\ n) && = NThrow\ n\\
&\quad reify\ (STryUnless\ n\ f\ g) && = NTryUnless\ n\ (reify \circ f)\ (reify \circ g)\\
&\quad reflect\ n = STryUnless\ n\ (SRetErr \circ eval)\ (SThrow \circ eval)
\end{aligned}$$

The data type definition of *MErr* gives rise to a semantic monad that can be used to evaluate the monadic expression constructors $Return_{err}$ and $Bind_{err}$ We evaluate *Throw* using semantic constructor *SThrow*, and *Catch* using a semantic function *catch′* that is implemented by pattern matching on its first argument.

$$\begin{aligned}
&eval\ (Return_{err}\ e) && = return\ (eval\ e)\\
&eval\ (Bind_{err}\ e\ f) && = eval\ e \gg\!= eval\ f\\
&eval\ (Throw\ e) && = SThrow\ (eval\ e)\\
&eval\ (Catch\ e\ f) && = catch'\ (eval\ e)\ (eval\ f)\\
&\textbf{instance}\ Monad\ MErr\ \textbf{where} \ldots\\
&catch' :: MErr\ sa \to (Sem\ String \to MErr\ sa) \to MErr\ sa
\end{aligned}$$

The rest of the definitions can be found in Appendix A.3.

5.2 Stateful computations

Similar to exceptions, stateful computations are also given equations specific to the operations *Put* and *Get*, in addition to the monad laws.

$$\begin{aligned}
&m :: Exp\ (State\ s\ a) \approx Bind_{st}\ Get\ (Lam\ (\lambda s \to seq\ (Put\ s)\ m))\\
&seq\ (Put\ x)\ (seq\ (Put\ y)\ m) \approx seq\ (Put\ y)\ m\\
&seq\ (Put\ x)\ (Bind_{st}\ Get\ f) \approx seq\ (Put\ x)\ (f\ x)
\end{aligned}$$

We have an η law as usual, and two reduction laws that reduce sequencing of *Put* and *Get* operations. Note that the function *seq* is a shorthand defined as $seq\ m\ m' = Bind_{st}\ m\ (Lam\ (\lambda\_ \to m'))$.

As with exceptions, there is a contention between two η-rules, and we choose the η-expanded form of an expression $e :: Exp\ (State\ s\ a)$ to be

$$\begin{aligned}
&Bind_{st}\ Get\ (Lam\ \$\ \lambda s \to (Bind_{st}\ e\ (Lam\ \$\ \lambda x \to\\
&\quad (Bind_{st}\ Get\ (Lam\ \$\ \lambda s' \to seq\ (Put\ s')\ (Return_{st}\ x))
\end{aligned}$$

Our normal forms reflect this choice, while also ensuring that the expression they denote cannot be further reduced by the β-rules.

$$\begin{aligned}
&\textbf{data}\ Nf\ a\ \textbf{where} \ldots\\
&\quad NGet :: (Rf\ a, Rf\ s) \Rightarrow (Exp\ s \to NfSt_{res}\ s\ a) \to Nf\ (State\ s\ a)\\
&\textbf{data}\ NfSt_{res}\ s\ a\ \textbf{where}\\
&\quad NPutRet_{st} :: (Rf\ a, Rf\ s) \Rightarrow Nf\ s \to Nf\ a \to NfSt_{res}\ s\ (State\ s\ a)\\
&\quad NBind_{st} :: (Rf\ a, Rf\ b, Rf\ s) \Rightarrow Ne\ (State\ s\ a)\\
&\qquad \to (Exp\ a \to Nf\ (State\ s\ b)) \to NfSt_{res}\ s\ (State\ s\ b)
\end{aligned}$$

Intuitively, a normal form of a state computation is a function that maps the global state to a chain of neutrals bound using $NBind_{st}$ that retrieve the latest state using *NGet*, and end with $NPutRet_{st}$.

$$\begin{aligned}
&NBind_{st}\ n_1\ (Lam\ \$\ \lambda e_1 \to Bind_{st}\ Get\ (Lam\ \$\ \lambda s_1 \to\\
&\quad NBind_{st}\ n_2\ (Lam\ \$\ \lambda e_2 \to Bind_{st}\ Get\ (Lam\ \$\ \lambda s_2 \to\\
&\qquad \ldots\\
&\quad NPutRet_{st}\ s_i\ a))))
\end{aligned}$$

The normal form constructor *NGet* captures the desired structure in its argument using the data type $NfSt_{res}$, that defines a separate syntactic category of normal forms. The constructor $NBind_{st}$ denotes a stuck binding, and $NPutRet_{st}$ denotes a fusion of *Put* followed by a $Return_{st}$.

The semantic domain is given by data types *MSt* and MSt_{res} that once again parallel the structure of normal forms.

```
newtype MSt s a = MSt {runMState :: Sem s → MSt_res s a}
data MSt_res s a where
  SPutRet_st :: (Rf a, Rf s) ⇒ Sem s → a → MSt_res s a
  SBind_st :: (Rf a, Rf b, Rf s) ⇒ Ne (State s a)
    → (Exp a → MSt s b) → MSt_res s b
instance (Rf s, Rf a) ⇒ Rf (State s a) where
  type Sem (State s a) = MSt s (Sem a)
  reify m                = NGet (reify_res ∘ runMState m ∘ eval)
    where
      reify_res :: MSt_res s (Sem a) → NfSt_res s a
      reify_res (SPutRet_st s x) = NPutRet_st (reify s) (reify x)
      reify_res (SBind_st n f)   = NBind_st n (reify ∘ f)
  reflect n = MSt $ λs → SBind_st n
    ((λx → MSt $ λs′ → SPutRet_st s′ x) ∘ eval)
```

The interpretation of *State s a* as *MSt s* (*Sem a*), along with the definition of *MSt* and MSt_{res} lends itself naturally to both reification and reflection.

Evaluation makes use of a monad instance for *MSt s* (defined in Appendix A.3) for $Return_{st}$ and $Bind_{st}$, and the *Get* and *Put* constructs are evaluated using a combination of the semantic constructors *MSt* and $SPutRet_{st}$.

```
eval (Return_st e) = return (eval e)
eval (Bind_st e e′) = eval e ≫= eval e′
eval (Get e)       = MSt $ λs → SPutRet_st s s
eval (Put e)       = MSt $ λ_ → SPutRet_st s ()
instance Monad (MSt s) where ...
```

5.3 Interaction with Sum types

As in the pure case, the semantic domains with effects also require refinement to account for sums. Unlike in the pure case, it is insufficient to merely place the monad *MDec* on top of the existing interpretation. This is because case analysis can also be performed in between operations and monadic binds. Our solution is to extend the data types that interpret effects with a constructor similar to *Branch* that allows case distinction within the monad.

```
data MErr a where ...
  SCaseErr :: (Rf a, Rf b) ⇒ Ne (Either a b)
    → (Exp a → MErr c)
    → (Exp b → MErr c) → MErr c
data MSt_res s a where ...
  SCaseSt :: (Rf a, Rf b) ⇒ Ne (Either a b)
    → (Exp a → MSt_res s c)
    → (Exp b → MSt_res s c) → MSt_res s c
```

Both *SCaseErr* and *SCaseSt* are reified using the normal form constructor *NCase*, and their addition does not preclude the implementation of the semantic functions such as *return*, (≫=), *catch′*, etc., which can be adapted easily to accommodate the new constructors (see Appendix A.3).

6 Practical NbE Extensions and Variations

The normalization procedures described in previous sections are adaptations of NbE for simply typed lambda calculus, that strive to identify the normal form of an expression as a canonical element of its equivalence class of semantically identical expressions. This traditional approach to NbE suffers from the following problems for practical eDSL applications:

- Our implementation β-reduces expressions as much as possible and η-expands expressions, yielding normal forms that are in β-short η-long form. Such aggressive normalization can lead to unnecessary code explosion, which may be harmful for code-generating eDSLs.
- The treatment of base types in Section 3 is insufficient for many practical applications. For example, the expression *Add* (*Var* `"x"`) (*Lift* 0) is not reduced, while we would typically like it to be reduced to (*Var* `"x"`).
- We have not yet explained how to incorporate *uninterpreted primitives*, that is, primitives without equations that dictate their behaviour.

In this section, we show these three problems can be addressed by refining the semantic domain used to implement NbE. Specifically, we present techniques to tame code expansion in NbE, a variation of NbE for integers that performs more advanced arithmetic reductions, and a recipe for adding uninterpreted primitives.

6.1 Taming Code Expansion

Disabling η-expansion using glueing. While η-expansion can be useful for some applications such as deciding program equivalence, it may be unsuitable for other applications such as code generation. For example, observe how the normalizer η expands the unknown *Var* `"f"` :: *Exp* (*Int* → *Arr Int*).

```
*NbE.OpenNbE> norm (Var "f" :: Exp (Int → Arr Int))
λarr.(NewArr (LenArr (f arr)) (λi.(f arr ! i)))
```

Our implementation of NbE applies η-expansion by default, but we show here how η-expansion can be selectively disabled using the *glueing* technique [17], yielding (potentially) smaller normal forms.

We begin by modifying our definition of normal forms to allow neutrals to be embedded directly.

```
data Nf a where ...
  NUp :: Ne a → Nf a
```

We remove the type constraint *Base a* on the constructor *NUp*, which relaxes the definition of normal forms to include, for example, the unknown *Var* `"f"` above as *NUp* (*NVar* `"f"`).

Let us suppose that we would like to disable η expansion for function types. We refine the semantic domain of function types to include a syntactic component by "glueing" (i.e. pairing) it with normal forms of the function type as follows.

```
instance (Rf a, Rf b) ⇒ Rf (a → b) where
  type Sem (a → b) = (Sem a → Sem b, Nf (a → b))
  reify       = snd
  reflect n = (..., NUp n)
```

Here we write ellipsis (...) for the original implementation of reflection. Reification projects the second component, a normal form, and reflection is modified to include an embedding of the neutral n to normal forms.

We proceed with evaluation as follows.

```
eval (Lam f)   = (..., NLam (reify ∘ eval ∘ f))
eval (App f e) = (fst (eval f)) (eval e)
```

For the case of *Lam*, we retain our previous implementation for the first component, and build a normal form in the second component. The evaluation of application is as before, with a minor modification that projects out the semantic function from the recursive evaluation of the expression f.

We may also disable η-expansion for the other types by modifying the interpretation similarly.

```
type Sem (a, b)    = ((Sem a, Sem b), Nf (a, b))
type Sem (Arr a) = (SArr (Sem a), Nf (Arr a))
 ...
```

Glueing provides a compositional solution to disabling η-expansion for some (or all) types without changing the implementation for other types. In contrast, another approach described by Lindley [28], where, for instance, the type $a \longrightarrow b$ is interpreted by *Either* (*Sem* $a \longrightarrow$ *Sem* b) (*Ne* ($a \longrightarrow b$)), requires a more involved reimplementation of the evaluator. Glueing can also be applied for effect types, but the definition of normal forms requires more careful consideration to avoid unnecessary expansion. Unlike in a strict language, the implementation of glueing in Haskell avoids a significant performance cost as the semantic and syntactic parts are only computed as required thanks to lazy evaluation.

Controlling duplication with explicit sharing. Much like other program specialization techniques, NbE can cause code duplication. For example, consider a function *double* that doubles its argument as *Lam* ($\lambda x \longrightarrow$ *Add* x x). Normalizing an application of *double* to an irreducible expression *large* causes it to be duplicated as *Add large large*.

Code duplication can be avoided with a *Let* construct for explicit sharing, for which NbE can be extended as follows.

```
data Exp a where ...
  Let :: (Rf a, Rf b) ⇒ Exp a → Exp (a → b) → Exp b
data Ne a where ...
  NLet :: (Rf a, Rf b) ⇒ Nf a → Nf (a → b) → Ne b
eval (Let e f) = reflect (NLet (norm e) (norm f))
```

Normalizing *Let* expressions respects sharing, and the expression *Let large double* does not reduce, avoiding duplication.

Disabling normalization on subexpressions. An alternative to explicit sharing is to disable normalization entirely on a subexpression using a construct *Save*, such that normalizing *Save* (*App double large*) returns the original expression unaffected[2]. We achieve this with a *Save* construct as follows.

```
data Exp a where ...
  Save :: Exp a → Exp a
data Ne a where ...
  NSave :: Exp a → Ne a
eval (Save e) = reflect (NSave e)
```

Optimizing case expressions. The implementation of commuting conversions for sums in Section 4 can produce normal forms with redundant or repeated case analysis.

```
Case scr (Lam $ λ_ → e) (Lam $ λ_ → e)
Case scr (Lam $ λx → Case scr...) (Lam $ λy → Case scr...)
```

The use of *Case* in these expressions is wasteful, and can be optimized further to reduce the size of the generated normal forms. Specifically, we are interested in the following two equations (identified by Lindley [29] as constituents of the general η-rule for sums).

```
Case scr (Lam $ λ_ → e) (Lam $ λ_ → e) ≈ e
Case scr (Lam $ λx → Case scr f1 f2)
         (Lam $ λy → Case scr g1 g2)
          ≈ Case scr (Lam $ λx → f1) (Lam $ λy → g2)
```

The first equation removes a redundant case analysis on *scr*, while the second removes a repeated analysis on *scr*.

One way to implement these equations is to refine the definition of *MDec* to preclude problematic decision trees by construction. However, given Haskell's limited support for dependent types and the pervasive nature of NbE for sums, this is a somewhat non-trivial modification. An easier (albeit ad hoc) solution is to implement a post-processing function *optimize* :: *Rf a* ⇒ *MDec* (*Nf a*) → *MDec* (*Nf a*) that is invoked before reifying decision trees. This is made possible since these transformations are merely syntactic manipulations of case trees that introduce no further reductions.

6.2 Applying Arithmetic Equations

To implement richer arithmetic equations (specified in Appendix A.1), our selection of normal forms must force the normalizer to perform reductions specified by these equations, for example, by reducing *Add* (*Lift* 1) (*Lift* 2) to *Lift* 3, *Add* (*Lift* 0) (*Var* "x") to *Var* "x", and so on.

We consider normal forms of integers to be in a sum-of-products form $(a_k * n_k) + (a_{k-1} * n_{k-1}) + ... + a_0$, where a_i denotes a constant, and n_i denotes a neutral expression, for each i. Correspondingly, we extend the definition of neutrals and normal forms as follows.

```
data Ne a where ...
  NMul :: Ne Int → Ne Int → Ne Int
data Nf a where ...
```

[2]in an implementation that disables η expansion entirely

```
  NInt  :: Int → Nf Int
  NAdd :: (Int, Ne Int) → Nf Int → Nf Int
```

The *NMul* constructor in neutrals denotes a stuck multiplication, and *NAdd* denotes the addition of an integer $(a_i * n_i)$ to the left end of an integer in sum-of-products form.

We define the semantic domain for integers using a data type *SOPInt* that is identical to the shape of normal forms, and use it in our definition of an instance of *Rf*.

```
data SOPInt where
  SInt :: Int → SOPInt
  SAdd :: (Int, Ne Int) → SOPInt → SOPInt
instance Rf Int where
  type Sem Int             = SOPInt
  reify (SInt a0)          = NInt a0
  reify (SAdd (ai, n_i) k) = NAdd (ai, n_i) (reify k)
  reflect n                = SAdd (1, n) (SInt 0)
```

The implementation of *reify* simply converts from the semantic domain to normal forms, while *reflect* expands a neutral n :: *Exp Int* to the form $(1 * n) + 0$.

Evaluation can be implemented by interpreting *Add* and *Mul* by their semantic counterparts *add′* and *mul′*, which can be defined by induction on values of *SOPInt*.

```
add′ :: SOPInt → SOPInt → SOPInt
mul′ :: SOPInt → SOPInt → SOPInt
eval (Add e_1 e_2) = add′ (eval e_1) (eval e_2)
eval (Mul e_1 e_2) = mul′ (eval e_1) (eval e_2)
```

The function *add′* adds two integers $(a_k * n_k) + ... + a_0$ and $(b_j * m_j) + ... + b_0$ in sum-of-products form by joining them as $(a_k * n_k) + ... + (b_j * m_j) + ... + (a_0 + b_0)$, and function *mul′* multiplies them as $(a_k * b_j) * (n_k * m_j) + (a_k * b_{j-1}) * (n_k * m_{j-1}) + ... + (a_0 * b_0)$.

6.3 Adding Uninterpreted Primitives

The core eDSL can be freely extended with uninterpreted primitives using the unknown constructor *Var*. For example, to extend our eDSL with a fixed-point construct without the corresponding equation, we define a combinator *fix* as:

```
fix :: Rf a ⇒ Exp ((a → a) → a)
fix = Var "Fix"
```

Normalizing an application *fix f* returns the equivalent of the expression of *fix* (*embNf* (*norm f*)), normalizing the function *f*, but leaving *fix* uninterpreted.

7 Related Work

The NbE technique goes back at least as far as Martin-Löf [31] who used it for proving normalization in his work on intuitionistic type theory. The core NbE algorithm for STLC was pioneered by Berger and Schwichtenberg [10]. The name is due to Berger et al. [9] who used it to speed up the Minlog theorem prover. A closely related technique is type-directed partial evaluation (TDPE) [18, 20, 21]. TDPE amounts to an instance of NbE used for partial evaluation in which the NbE semantics is exactly that of the host language. In contrast, for embedding DSLs we make essential use of non-standard semantics, e.g. using glueing for suppressing η-expansion.

Normalization for pure call-by-name STLC with sums is notoriously subtle [23] because general η-rule for sums includes additional equations such as those described in Section 6. Altenkirch et al. [3] give an NbE algorithm for sums based on a *Grothendieck topology* which implicitly captures the kind of decision tree that we use, but at every type. Balat et al. [7], in contrast, make use of multiprompt delimited control to allow retrospective exploration of different branches during reification. Both algorithms build in a degree of syntactic manipulation in order to manage redundant and repeated case splits similarly to what we describe in Section 6.

NbE for sums becomes considerably easier in an effectful call-by-value setting, as fewer equations hold. Danvy [18] uses (single prompt) delimited control operators for handling sums in TDPE. Filinski [22] adapts Danvy's approach to computational lambda calculus extended with sums. Lindley [30] adapts Filinksi's work to replace delimited control with an *accumulation* monad which we here call a *decision tree* monad and Abel and Sattler [1] characterise as a *cover* monad. The Danvy/Filinski approach based on delimited control is at the heart of the treatment of sums in existing eDSLs [39].

Ahman and Staton [2] give an NbE algorithm for general algebraic effects. We speculate that our bespoke treatment of specific monadic effects can be related to their generic approach, but we do not know to what extent their approach maps conveniently onto the Haskell eDSL setting.

Implementations of NbE in Haskell are not new. For instance, Danvy et al. [19] give an implementation not dissimilar to ours for plain STLC. Prior work on combining deep and shallow embeddings [39] implicitly uses a restricted form of NbE. Their *Syntactic* type class plays a similar role to our *Rf* type class. However, they do not make a connection with NbE and they do not use an instance for functions.

We have presented NbE as a unifying framework for eDSLs based on solid theoretical foundations. A related framework is offered by quoted domain-specific languages (QDSLs) [32]. QDSLs exploit a similar normalization procedure as part of the embedding process. A key difference is that QDSLs are based on staging and a separate normalization algorithm.

8 Final Remarks

We have presented, to the best of our knowledge, the first comprehensive practical implementation of NbE for Haskell eDSLs. NbE provides a systematic and modular approach to specialize eDSL programs in Haskell, and provides a principled account of ad hoc techniques previously developed using a combination of deep and shallow embedding. We have shown how problems that arise from a traditional approach to NbE can be addressed to suit practical concerns such as code expansion, normalization with domain-specific equations, and extension with uninterpreted primitives. We believe that NbE has a broader applicability beyond the examples of fusion shown here. For example, NbE could be used in the security domain, to automatically remove superfluous security checks performed at runtime by programs written in a security eDSL (e.g., [38]). Similarly, in databases (e.g., [36]), NbE could be used normalize queries written in a higher-order eDSL to achieve elimination of higher-order functions and other intermediate data-structures [15, 25].

References

[1] Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 1–12.

[2] Danel Ahman and Sam Staton. 2013. Normalization by Evaluation and Algebraic Effects. In *MFPS (Electronic Notes in Theoretical Computer Science, Vol. 298)*. Elsevier, 51–69.

[3] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. 2001. Normalization by Evaluation for Typed Lambda Calculus with Coproducts. In *LICS*. IEEE Computer Society, 303–310.

[4] Markus Aronsson, Emil Axelsson, and Mary Sheeran. 2014. Stream processing for embedded domain specific languages. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. 1–12.

[5] Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding domain-specific languages. In *Haskell*. ACM, 37–48.

[6] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. 2010. The Design and Implementation of Feldspar - An Embedded Language for Digital Signal Processing. In *IFL (Lecture Notes in Computer Science, Vol. 6647)*. Springer, 121–136.

[7] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*. ACM, 64–76.

[8] Nick Benton and Andrew Kennedy. 2001. Exceptional syntax. *Journal of Functional Programming* 11, 4 (2001), 395–410.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalisation by Evaluation. In *Prospects for Hardware Foundations (Lecture Notes in Computer Science, Vol. 1546)*. Springer, 117–137.

[10] Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *LICS*. IEEE Computer Society, 203–211.

[11] Ilya Beylin and Peter Dybjer. 1995. Extracting a Proof of Coherence for Monoidal Categories from a Proof of Normalization for Monoids. In *TYPES (Lecture Notes in Computer Science, Vol. 1158)*. Springer, 47–61.

[12] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *ICFP*. ACM, 174–184.

[13] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543.

[14] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP*. ACM, 3–14.

[15] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ICFP*. ACM, 403–416.

[16] Catarina Coquand. 1993. From Semantics to Rules: A Machine Assisted Analysis. In *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers (Lecture Notes in Computer Science, Vol. 832)*, Egon Börger, Yuri Gurevich, and Karl Meinke (Eds.). Springer, 91–105. https://doi.org/10.1007/BFb0049326

[17] Thierry Coquand and Peter Dybjer. 1997. Intuitionistic Model Constructions and Normalization Proofs. *Math. Struct. Comput. Sci.* 7, 1 (1997), 75–94.

[18] Olivier Danvy. 1998. Type-Directed Partial Evaluation. In *Partial Evaluation (Lecture Notes in Computer Science, Vol. 1706)*. Springer, 367–411.

[19] Olivier Danvy, Morten Rhiger, and Kristoffer Høgsbro Rose. 2001. Normalization by evaluation with typed abstract syntax. *J. Funct. Program.* 11, 6 (2001), 673–680.

[20] Peter Dybjer and Andrzej Filinski. 2000. Normalization and Partial Evaluation. In *APPSEM (Lecture Notes in Computer Science, Vol. 2395)*. Springer, 137–192.

[21] Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *PPDP (Lecture Notes in Computer Science, Vol. 1702)*. Springer, 378–395.

[22] Andrzej Filinski. 2001. Normalization by Evaluation for the Computational Lambda-Calculus. In *TLCA (Lecture Notes in Computer Science, Vol. 2044)*. Springer, 151–165.

[23] Neil Ghani. 1995. $\beta\eta$-Equality for Coproducts. In *TLCA (Lecture Notes in Computer Science, Vol. 902)*. Springer, 171–185.

[24] Andy Gill. 2014. Domain-specific languages and code synthesis using Haskell. *Commun. ACM* 57, 6 (2014), 42–49.

[25] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. 2010. Haskell Boards the Ferry - Database-Supported Program Execution for Haskell. In *IFL (Lecture Notes in Computer Science, Vol. 6647)*. Springer, 1–18.

[26] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (1996), 196.

[27] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA.

[28] Sam Lindley. 2005. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. Ph.D. Dissertation. University of Edinburgh.

[29] Sam Lindley. 2007. Extensional Rewriting with Sums. In *TLCA (Lecture Notes in Computer Science, Vol. 4583)*. Springer, 255–271.

[30] Sam Lindley. 2009. Accumulating bindings. In *NBE 2009*. 49–56.

[31] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*. Vol. 80. Elsevier, 73–118.

[32] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *PEPM*. ACM, 25–36.

[33] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. ACM, 199–208.

[34] Dag Prawitz. 1971. Ideas and results in proof theory. In *Proceedings of the 2nd Scandinavian Logic Symposium (Studies in Logics and the Foundations of Mathmatics, 63)*. North Holland, 235–307.

[35] Morten Rhiger. 2003. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.* 25, 3 (2003), 291–315.

[36] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2–9.

[37] Alejandro Russo, Koen Claessen, and John Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008*. 13–24.

[38] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell,*. 95–106.

[39] Josef Svenningsson and Emil Axelsson. 2015. Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.* 44 (2015), 143–165.

[40] Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. 2014. Design exploration through code-generating DSLs. *Commun. ACM* 57, 6 (2014), 56–63.

[41] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards secure IoT programming in Haskell. In

Haskell@ICFP. ACM, 136–150.

A Appendix

A.1 Equational Theory

Commmuting conversions.

$F\ (Case\ e\ g\ h) \approx Case\ e\ (F \circ g)\ (F \circ h)$

This equation specifies commuting conversions that enable us to push eliminators under a case expression, for example, as:

$App\,f\ (Case\ e\ g\ h) \approx Case\ e\ (App\,f \circ g)\ (App\,f \circ h)$

The symbol F in the equation is a unary function on expressions that denotes an elimination context such as $App\ f :: Exp\ a \longrightarrow Exp\ b$ (for some $f :: Exp\ (a \longrightarrow b)$), $Fst :: Exp\ (a, b) \longrightarrow Exp\ a$, or $Add\ e :: Exp\ Int \longrightarrow Exp\ Int$, etc.

Arithmetic and primitive recursion.

$$
\begin{aligned}
(Lift\ x) + (Lift\ y) &\approx Lift\ (x + y)\\
(Lift\ 0) + e &\approx e\\
(Lift\ x) + (e_1 + e_2) &\approx e_1 + (Lift\ x + e_2)\\
(e_1 + e_2) + e_3 &\approx e_1 + (e_2 + e_3)
\end{aligned}
$$

$$
\begin{aligned}
(Lift\ x) * (Lift\ y) &\approx Lift\ (x * y)\\
(Lift\ 0) * e &\approx Lift\ 0\\
(Lift\ 1) * e &\approx e\\
(Lift\ x) * (e_1 + e_2) &\approx (Lift\ x * e_1) + (Lift\ x * e_2)\\
(e_1 + e_2) * e_3 &\approx (e_1 * e_3) + (e_2 * e_3)
\end{aligned}
$$

A.2 Normalizing Primitive Recursion

The recursion construct *Rec* can be used to perform primitive recursion. As mentioned earlier for its combinator counterpart *rec*, an expression $Rec\ n\ f\ x$ is the equivalent of applying f repetitively as $f\ 1\ (f\ 2\ (\ldots(f\ n\ x)))$. This behaviour can be specified by the following equations.

```
Rec i f x           ≈ x    -- (i <= 0)
Rec (e₁ + e₂) f x ≈ Rec e₁ f (Rec e₂ f x)
```

To extend our NbE implementation with recursion, we extend the definition of neutrals with a new constructor for "stuck" recursion as follows.

```
data Ne a where ...
  NRec :: Rf a ⇒ (Int, Ne Int)
      → (Exp Int → Exp a → Nf a) → Nf a → Ne a
```

We then evaluate recursion using a semantic function *rec′*.

```
rec′ :: Rf a ⇒ SOPInt
  → (Sem Int → Sem a → Sem a) → Sem a → Sem a
rec′ (SInt i) f x
  | i ⩽ 0       = x
  | otherwise = rec′ (SInt (i − 1)) f (f (SInt i) x)
rec′ (SAdd aini k) f x
  = reflect (NRec aini f′ (reify (rec′ k f x)))
   where
     f′ i b = reify (f (eval i) (eval b))
eval (Rec n f x) = rec′ (eval n) (eval f) (eval x)
```

When the value of an integer is available, *rec′* performs the expected recursion, and otherwise simply applies the second equation of recursion.

A.3 Semantic Monads

```
instance Monad MDec where
  return x = Leaf x
  (Leaf x)          ≫= f = f x
  (Branch n g h) ≫= f = Branch n ((=≪) f ∘ g) ((=≪) f ∘ h)

instance Monad MErr where
  return x = SRetErr x
  (SRetErr x)            ≫= f = f x
  (SThrow x)             ≫= f = SThrow x
  (STryUnless n g h) ≫= f = STryUnless n ((=≪) f ∘ g) ((=≪) f ∘ h)
  (SCaseErr n g h)    ≫= f = SCaseErr n ((=≪) f ∘ g) ((=≪) f ∘ h)

catch′ :: MErr sa → (Sem String → MErr sa) → MErr sa
catch′ (SRetErr x)        f = SRetErr x
catch′ (SThrow x)         f = f x
catch′ (STryUnless n g h) f = STryUnless n
  (flip catch′ f ∘ g) (flip catch′ f ∘ h)
catch′ (SCaseErr n g h) f = SCaseErr n
  (flip catch′ f ∘ g) (flip catch′ f ∘ h)

instance Functor (MSt s) where
  fmap f m = MSt $ fmap f ∘ runMState m
instance Functor (MSt_res s) where
  fmap f (SPutRet_st s x)  = SPutRet_st s (f x)
  fmap f (SBind_st n g)     = SBind_st n (fmap f ∘ g)
  fmap f (SCaseSt n g h) = SCaseSt n (fmap f ∘ g) (fmap f ∘ h)
joinMState :: MSt s (MSt s a) → MSt s a
joinMState m = MSt $ magic ∘ runMState m
magic :: MSt_res s (MSt s a) → MSt_res s a
magic (SPutRet_st s m)  = runMState m s
magic (SBind_st n g)     = SBind_st n (joinMState ∘ g)
magic (SCaseSt n g h) = SCaseSt n (magic ∘ g) (magic ∘ h)
instance Monad (MSt s) where
  return x = MSt $ λs → SPutRet_st s x
  m ≫= f = joinMState (fmap f m)
```