Combinators for Impure yet Hygienic Code Generation

Yukiyoshi Kameyama

Tsukuba, Japan

Oleg Kiselyov

Sendai, Japan

Chung-chieh Shan

Bloomington, Indiana, U.S.A

Abstract

Code generation is the leading approach to making high-performance software reusable. Effects are indispensable in code generators, whether to report failures or to insert **let**-statements and **if**-guards. Extensive painful experience shows that unrestricted effects interact with generated binders in undesirable ways to produce unexpectedly unbound variables, or worse, unexpectedly bound ones. These subtleties hinder domain experts in using and extending the generator. A pressing problem is thus to express the desired effects while regulating them so that the generated code is correct, or at least correctly scoped, by construction.

We present a code-combinator framework that lets us express arbitrary monadic effects, including mutable references and delimited control, that move open code across generated binders. The static types of our generator expressions not only ensure that a well-typed generator produces well-typed and well-scoped code. They also express the lexical scopes of generated binders and prevent mixing up variables with different scopes. For the first time ever we demonstrate statically safe and well-scoped loop interchange and constant factoring from arbitrarily nested loops.

Our framework is implemented as a Haskell library that embeds an extensible typed higher-order domain-specific language. It may be regarded as 'staged Haskell.' To become practical, the library relies on higher-order abstract syntax and polymorphism over generated type environments, and is written in a mature language.

Keywords: multi-stage programming, mutable state and control effects, binders, CPS, higher-order abstract syntax, lexical scope

Email addresses: `kameyama@acm.org` (Yukiyoshi Kameyama), `oleg@okmij.org` (Oleg Kiselyov), `ccshan@indiana.edu` (Chung-chieh Shan)

URL: `http://okmij.org/ftp/` (Oleg Kiselyov)

1. Introduction

High-performance computing applications (scientific simulation [1], digital signal processing [2], network routing [3], and many others) require domain-specific optimizations that the typical domain expert performs by hand over and over again to write each specialized program. Examples include selective loop unrolling and loop tiling (described in §2.6); array padding, matrix tiling and other data representation changes; writing specialized Gaussian elimination kernels [4]. The manual optimizations are not only excruciating: their correctness is hard to see, they distort the code beyond recognition making it unmaintainable, and they require significant and rare expertise, which cannot be reused for similar cases. It is widely recognized in the high-performance computing community (see [5] and references therein) that we cannot count on optimizing compilers to perform these domain-specific optimizations. Code generation has emerged as one of the most promising approaches to producing optimal code with high confidence and relatively low effort [5].

In this approach, the optimal code is the result of a code generator – or is selected from the results of a family of generators through empirical search. A generator is built from reusable pieces written by different groups of experts; the pieces encapsulate parts of the algorithm (such as dot-product or pivoting), parameterized by optimizations (like partial loop unrolling and tiling, scalar promotion) or specializations (such as the preferred data layout, or statically known inputs). §2.6 shows off a few such pieces. The pieces may be regarded as a domain-specific language (DSL). Ideally, they should compose: a user could replace one algorithm with a putatively faster one or apply a different optimization, without changing the rest of the generator. Furthermore, the user should reason in terms of the generator pieces rather than the generated code: the generator DSL should abstract over the code. The generated code should at the very least be well-formed and well-typed, and hence compile without errors. The end user hence should be spared from looking at it, let alone debugging compilation problems. (The end user might not even know the target language of the generator). We also require another property, which we dub *well-scopedness*. It relates the generator and the generated code and is stronger than the absence of unbound variables in the generator's result. The property ensures the absence of unexpectedly bound variables (we illustrate the surprising bindings in §4.1). Somewhat informally, well-scopedness means that the scope of variables in the generated code can be determined by statically examining the generator, before running it.

The goals of expressivity, composability and static assurances are in conflict, which so far has remained unresolved. (See §5 for the discussion of trade-offs.) In this paper we report the first approach that *simultaneously* achieves the goals. We express optimizations such as loop tiling, we change optimizations without rewriting the rest of the generator, and we assure the well-formedness, well-typedness as well as well-scopedness of the generated code.

1.1. Code-generating combinators, effects, and scope extrusion

A generator is a program in one language (called a host language, or metalanguage) that produces programs in another, possibly different language (called target, or object language). In the present paper, the metalanguage is Haskell. One style of writing generators is filling in templates containing the target code in the concrete syntax. Antiquotations in Lisp is the familiar example of such code templates. The other style relies on code-generating combinators [6–8]. The former are quite more convenient for the user but require assistance from the compiler and the typechecker. Code-generating combinators are more suitable for prototyping and investigating the design space (which is our activity). In the rest of the paper, we focus on combinators.

To give a taste of building optimal code with code-generating combinators and illustrate the subtle variable scoping problems that arise along the way, we use the simple example of multiplying an $n \times m$ matrix a by a vector v obtaining v'. The textbook matrix-vector multiplication algorithm is usually given as the following C code, assuming the output vector, written as v1 in C, is initially zeroed out.

```
for (j = 0; j≤m−1; j++ )
  for (i = 0; i≤n−1; i++ )
    v1[i ] += a[i][j ] ∗ v[j ];
```

Written with code-generating combinators of our library, the example takes the form

```
loop (int 0) (int (m−1)) (int 1) (lam $ \j →
 loop (int 0) (int (n−1)) (int 1) (lam $ \i →
  vec_addto (weakens v') (vr i ) =≪ :
    (mat_get (weakens a) (vr i ) (vr j ) `mulM` vec_get (weakens v) (vr j ))))
```

This example, discussed in detail in §2.6, is shown here just to give a rough idea of how a generator of the matrix-vector multiplication may look like. On the first line we see the combinator loop to build the loop in which the index variable, represented by i in the loop body, ranges from 0 through m−1 with the unit step; int generates the code for a given integer. The combinators vec_get and mat_get generate the code to access an element of a vector or a matrix; therefore, vec_get (weakens v) (vr j) on the last line is to generate the code corresponding to v[j] in the textbook algorithm. The function vr marks the reference to a bound variable (the loop counter here). The near ubiquitous weakens lets us refer to the representation of the target code from under binders. The combinator vec_addto produces the code to add a value to a given vector element. Each combination and the whole expression, when evaluated, produces some representation for the target code, a 'code value'. Typical representations are the abstract syntax tree (AST) or text strings. In the rest of the paper, for clarity and concreteness, the combinators produce Haskell code (although other choices are possible, including OCaml, JavaScript, etc.)

It should be clear by now that the generated code is imperative, as was the C algorithm. What mat_get and vec_get produce is the effectful code to read from a mutable array. If the target language is Haskell, then mat_get would generate an IO Double expression. The combinator mulM takes two pieces of the generated effectful code and emits the code to multiply them. The imperative nature of the generated code comes through in (=≪ :), which arranges so that at the run time of the produced code, first the effects of getting a and v elements are performed and their product is obtained; then the vector v' is updated inplace with the resulting value. Again, if the target code is Haskell, (=≪ :) is the combinator that generates the monadic bind expression.

There may be several implementations of the combinator loop. In fact, there could be libraries of loop combinators written by high-performance computing experts. Besides the straightforward generation of the simple loop forM_ [0,1..m−1][1] the combinator may strip a loop into blocks, effectively converting a loop into two nested loops (the outer iterating over blocks and the inner iterating within a block). Such a combinator implements the classical 'strip mining' optimization. For example, with the blocking factor 4, the strip-mining loop will generate

```
forM_ [0,4.. m−1] $ \jj →
 forM_ [jj ,jj +1..min (jj +3) (m−1)] $ \j →
  forM_ [0,4.. n−1] $ \ii →
   forM_ [ii ,ii +1..min (ii +3) (n−1)] $ \i →
   ...
```

Strip-mining is a common optimization, used, for example, as the first phase of vectorization. The blocking factor may be requested from the user or *learned*. That is, the generator non-deterministically produces several candidates with different degrees of blocking, to benchmark and choose the fastest. SPIRAL [2] is based on this idea. Generators, therefore, need effects such as IO, exceptions, and non-determinism. We stress that now we are talking about effects when *generating* code – rather than IO, mutation and other effects that the generated code itself may perform.

We are particularly interested in control effects that are necessary for the follow-up optimization, changing the order of the loops, to produce so-called 'tiled loops'

```
forM_ [0,4.. m−1] $ \jj →
forM_ [0,4.. n−1] $ \ii →
   forM_ [jj ,jj +1..min (jj +3) (m−1)] $ \j →
   forM_ [ii ,ii +1..min (ii +3) (n−1)] $ \i →
   ...
```

which is one of the very profitable optimizations. Our aim is to write a generator once, following the textbook algorithm, and then to choose an appropriate implementation of loop for strip-mining, tiling and other optimizations.

To achieve loop tiling, specifically, to interchange striped loops, the two loop combinators in the sample code must interact – they have to be effectful. The danger of producing ill-scoped code is also clear. For example, if the wrong loops are interchanged, we may move the code that uses jj past its binder:

```
forM_ [jj ,jj +1..min (jj +3) (m−1)] $ \j →
forM_ [0,4.. m−1] $ \jj →
 ...
```

which is called 'scope extrusion'. This error becomes devious if the above code

[1] forM_ [lb,lb+st..ub] (\i → body) is how for i=lb to ub step st do body is written in Haskell.

is part of a program with another binder for jj :: Int. The generated code will successfully compile and even run, but with subtle errors. A similar optimization, also requiring control effects, is loop-invariant code motion, moving the code that does not depend on the loop index out of the loop. Moving out the code that does depend on the loop index again creates scope extrusion. We wish to prevent scope extrusion statically and right away, even for separate program fragments. The code should be well-typed and well-scoped *as it is being constructed.*

Performing optimization like loop interchange while statically preventing scope extrusion was identified as an open problem by Cohen et al. [5]. We summarize in §5.2 what makes the problem so difficult. It is only now that the problem has been solved, in a mainstream metalanguage such as Haskell, in a code-generation framework that can be used in practice.

1.2. Contributions

Our goal was to generate code with modular combinators that statically assure that the results (even intermediate, open results) are well-formed and well-typed. This goal is achieved. Specifically, our contributions are as follows.

First, we present a method for writing code combinators that may do arbitrary monadic effects – including effects that move open code across generated binders – and yet preserve lexical scope (hygiene). The key ingredients are:

- tagless-final style of encoding target language in the metalanguage [9], to ensure the generation of well-typed code: §3;
- applicative functors, or *applicatives*, [10] to represent the type environment of open target code: §3.1;
- first-class polymorphism, to ensure free variables in the target code are manipulated strictly within the dynamic extent of the generator that later binds them: §4.1;
- new applicative CPS hierarchy, for let-insertion, loop interchange, or, generally moving code that contains binding forms across another binding form: §3.2.

We use types to express and enforce a notion of lexical scope on generated code. Our type discipline ensures that generated variables are always bound intentionally, never captured accidentally. We argue (see §4.1 for details) that lexical scope in a code generator means that different generated variables cannot be substituted for each other (because they have different types in our system), even if they have the same named label or the same De Bruijn index. In the present paper, our argumentation is informal. The formalization is quite involved and is the subject of another paper. We stress that our method requires neither intersection nor dependent types; all the needed features such as higher-rank types are commonly available in mainstream functional languages.

Second, we present a Haskell library of code-generating combinators, which we built to validate our approach. This paper explains our approach to effects and scope by describing this library. Our concrete examples show how to express effects on open code across generated binders, as well as how rank-2 types enforce lexical scope.

Although we use Haskell extensively in the paper for the sake of concreteness and presentation, we stress that Haskell is not essential for our approach of safely combining code generation and effects. Granted, describing the Haskell library and presenting its examples necessarily involves a great deal of Haskell code. We explain the parts that may likely cause confusion for non-Haskell programmers. It might appear from the presentation that our approach is too dependent on Haskell and difficult if not impossible to transfer to other functional languages. To allay these concerns, we first point out that the first presentation [9] of the tagless-final style, a key ingredient of our approach, was done in OCaml and with ordinary, second-class OCaml modules. The tagless-final style has been used in Scala, where it inspired the highly-successful lightweight modular staging LMS [11]. The second key ingredient, first-class polymorphism and higher-rank types, are also commonly at hand (in OCaml and Scala). Monad and applicative abstractions require higher-order types, that is, abstracting over type constructors. Such types are not available in OCaml – but can be emulated [12]. On the other hand, our use of applicatives is itself an emulation, of an effect system. Languages with native effect systems such as AST, Idris, and Rust can realize our approach more directly. Finally, a bare-bone implementation of our library replaces the type classes with the explicit dictionary passing (for an example of doing this for the tagless-final approach, see [13, §2.4]). The plumbing, the dictionary passing, can be hidden in the code-generating combinators. Thus Haskell type classes, although adding quite a bit of convenience, are not at all essential to our approach. We could have used the added convenience of type classes to a larger extent (by overloading operations such as the ordinary (∗) as well as (∗:) and `mulM` that we shall soon encounter). We deliberately refrain from too much of the type-class sugar, making the library more spartan and hence more portable and accessible.

Our library is not yet ready for real-life applications like those supported by MetaOCaml [14]. First, the syntax is rather heavy. We write `int 1 +:  int 2` to generate the code for `1 + 2`. (We avoid overloading Haskell's type class `Num`, for clarity and to emphasize that our approach is not restricted to Haskell but works in any functional language with higher-rank polymorphism.) Moreover, weakening coercions have to be applied explicitly to generated code, and there is no syntactic sugar for pattern matching. (Again, type-class overloading can help.) Although we have implemented many interesting examples with our library, more experience is needed to recommend its wide practical use. Still, our library is imminently practical in that

1. it has been built in the mature language Haskell, not an experimental language with a dearth of documentation and tools;
2. it uses higher-order abstract syntax (HOAS) [15–17] rather than De Bruijn indices, so bindings in the generator are human-readable;
3. it allows polymorphism over generated environments, so the same gener-

ator module can be reused in many environments;

4. in many cases, the types are inferred. The type signature is only necessary for generator functions that, informally, "take the target code and put it under a generated lambda"[2]
5. the language of generated code can easily be extended with more features and constants (this paper shows many examples) or changed to any other language – typed or untyped, first- or higher-order.

Our library can serve as a prototype for a more expressive and yet statically safer Template Haskell, permitting arbitrary effects including IO while still ensuring the generation of the well-typed and well-scoped code (§5 explains why the current Template Haskell falls short.)

Finally, we propose a new applicative continuation-passing-style (CPS) hierarchy that allows loop interchange and let-insertion across several generated bindings. These tasks cannot be accomplished in the traditional CPS hierarchy [18].

The structure of the paper. [3] The next section introduces the interface of our library and explains it on simple examples of code generation with effects. We then turn to the examples that were not possible to write before with static well-scopedness assurances: in §2.3 we store open code in mutable variables across binders, and in §2.5 we do the invariant code motion, which changes the order of binding forms. Attempts to move open code beyond the binders of its variables are flagged as type errors. §2.6 describes a case study of using our library for common loop optimizations: loop interchange and loop tiling. We briefly outline the implementation in §3, and describe and informally justify in §4 the static assurances of our generators. §5 discusses related work, specifically the comparison with Template Haskell (TH).

For brevity and clarity, the code shown in the paper is not always self-contained. Furthermore, we omit type class constraints on some, specifically named type variables, following the conventions introduced as needed. Should confusion arise the reader may refer to Figure 1 and 3, which show the complete signatures of the public functions in our library. Furthermore, the complete code is available online at `http://okmij.org/ftp/tagless-final/TaglessStaged/`.

2. Library

This section presents our library of impure and yet hygienic code-generating combinators and illustrates it on a progression of examples. Figure 1 shows the

[2] For generators that introduce multiple target code variables, especially the statically unknown number of them, we sometimes have to prove that the composition of applicatives is associative, by explicitly writing the corresponding conversion functions.

[3] The present paper is an extended version of the paper published in the proceedings of PEPM 2014. We have extended explanations all throughout the paper, especially the explanation of performing monadic effects during code generation in §2.3. Sections 2.4, 3.1, 3.2 and 5.2 are new.

library interface.

The target code is treated as an EDSL, a domain-specific language embedded in Haskell. The target language is simply-typed[4]: the target code of type t is represented as a Haskell value of the type repr t. It is a coincidence that the same Int denotes the integer type both in Haskell and in the target language. The Haskell function (combinator) intS represents an integer literal of the target language, and addS stands for the target's addition function. The combinator appS, which combines two code values, denotes application in the target language. Whereas (1 +2):: Int, which is the same as (+) 1 2, expresses Haskell's addition, exS1 of the characteristic type below

```
exS1 ::  SSym repr ⇒ repr  Int
exS1 = (addS `appS` intS 1) `appS` intS  2
```

represents an integer expression in the target language for adding the two integers. We should have said, however pedantic for now, that exS1 is the Haskell expression that, when evaluated, produces a value representing "1+2" in the target language. The types, inferred by the Haskell compiler, make it clear which Haskell expressions denote target code, that is, are code generators. The latter have a type like that of exS1; one may even read repr Int as "the type of code for an integer expression". By convention, we will often elide the type class constraint SSym repr (or similar constraints SymLet, LamPure, etc.) for the type variable named repr.

The method of embedding a language by defining its primitives as methods of a type class such as SSym repr – with the parameter repr :: $* \to *$ standing for a concrete realization indexed by the expression's type – is called the "tagless final" approach [9]. The approach encourages several concrete realizations of the target language. Our library provides two. The first is a so-called meta-circular interpreter: the type R, which is the instance of SSym and similar extension classes, takes the target code to be a subset of Haskell and realizes the code as a Haskell expression: R is essentially the identity functor. Since Haskell is non-strict, it is more precise to say that the R-realization represents target code as a 'thunk.' The C-realization also takes the target code to be Haskell, but represents it as a Haskell AST. The function runCS pretty-prints the AST, letting us see the generated code. For example, runCS exS1, instantiating repr to C, produces "(GHC.Num.+)␣1␣2". (When showing the code we shall assume hereafter a prettier-printer that elides the module qualification GHC.Num and uses the infix notation.) The generated code can be 'spliced-in' using Template Haskell (TH), or saved into a file and compiled separately.

[4] Producing well-typed code in C or Fortran, whose type systems differ from the type system of the generator, is described in [19]. It is possible to embed in Haskell target languages with polymorphism, and also the languages whose type system is not at all like Haskell, for example, linear lambda-calculus or non-associative Lambek calculus. For the present paper, we limit ourselves to a simply-typed target language.

Pure base combinators

```
class SSym repr where
  intS :: Int → repr Int
  addS :: repr (Int → Int → Int)
  mulS :: repr (Int → Int → Int)
  appS :: repr (a→ b) → (repr a → repr b)
```

Two representations of the code

```
newtype R a = R{unR :: a}

type C a                  −− Haskell AST
runCS :: C a → String     −− pretty−printed AST
```

Generating code with effects, in an applicative m

```
infixl 2 $$
($$ ) :: (SSym repr, Applicative m) ⇒ m (repr (a→ b)) → m (repr a) → m (repr b)
int   :: (SSym repr, Applicative m) ⇒ Int → m (repr Int )

infixl 7 ∗:      infixl 6 +:
(+: ), (∗: ) :: (SSym repr, Applicative m) ⇒
        m (repr Int ) → m (repr Int ) → m (repr Int )

−− Composition of two type constructors (kind ∗ → ∗ )
newtype (i ∘ j ) a = J{unJ:: i (j a)}
liftJ :: (Applicative m, Applicative i ) ⇒ m a → (m ∘ i ) a

runR :: Applicative m ⇒ (m ∘ Identity ) (R a) → m a
runC :: Applicative m ⇒ (m ∘ Identity ) (C a) → m String
```

Higher-order fragment

```
class LamPure repr where
  lamS :: (repr a → repr b) → repr (a→ b)

lam :: (Applicative m, AppLiftable i , SSym repr, LamPure repr) ⇒
       (∀ j . AppLiftable j ⇒ (i ∘ j ) (repr a) → (m ∘ (i ∘ j )) (repr b))
       → (m ∘ i ) (repr (a→ b))

var    :: Applicative m ⇒ i (repr a) → (m ∘ i ) (repr a)
weaken :: (Applicative m, Applicative i , Applicative j ) ⇒
          (m ∘ i ) (repr a) → (m ∘ (i ∘ j )) (repr a)

class (Applicative m, Applicative n) ⇒ Extends m n where
  weakens :: m a → n a
vr :: (Applicative m, Extends (m ∘ i ) (m ∘ j )) ⇒ i (repr a) → (m ∘ j ) (repr a)
vr = weakens ∘ var
```

Extension: let-expressions

```
class SymLet repr where
  let_S :: repr a → (repr a → repr b) → repr b

let_ :: (SSym repr, SymLet repr, Applicative m, Applicative i ) ⇒
  (m ∘ i ) (repr a)
  → (∀ j . AppLiftable j ⇒ (i ∘ j ) (repr a) → (m ∘ (i ∘ j )) (repr b))
  → (m ∘ i ) (repr b)
```

Figure 1: The interface of hygienic code-generating combinators

Interface of functors

```
class  Functor i  where
  fmap :: (a → b) → i a → i b

class Functor i ⇒ Applicative i where
  pure :: a → i a
  (<*>) :: i (a → b) → i a → i b

class Applicative j ⇒ AppLiftable j where
  app_pull :: Applicative i ⇒ i (j a) → j (i a)
```

Sample instances for the Reader applicative $\mathsf{e} \to \cdot$ (or, ((→) e) in Haskell notation): functions of the fixed argument type e

```
instance Functor ((→ ) e) where
  fmap f i = f ∘ i

instance Applicative ((→ ) e) where
  pure x    = \e → x
  (<*>) f g = \e → f e (g e)

instance AppLiftable ((→ ) e) where
   app_pull ija = \e → fmap ($ e) ija
```

Closure under composition

```
instance (Applicative i, Applicative j) ⇒ Applicative (i ∘ j)

instance (AppLiftable j, AppLiftable k) ⇒ AppLiftable (j ∘ k) where
  app_pull = J ∘ fmap app_pull ∘ app_pull ∘ fmap unJ
```

Figure 2: Applicatives: applicative functors

2.1. Faulty power

As the first example to illustrate our library we take the canonical power – specializing x^n to the given value of n [20]. Already this trivial example calls for effects in code generation – although this aspect of power is frequently glossed over. Since the effect here is simple – throwing a string as an exception – all code-generation frameworks (for example, MetaOCaml, Mint [21] or $\lambda^{\oslash}$ [22]) can assure the generation of well-scoped code. We rely on the simplicity and the familiarity of power to introduce our library, pointing out how it ensures well-scopedness by design. The later sections show off this design in full, demonstrating the unique expressive power of our library.

The integral exponentiation x^n can be written in Haskell as:

```
type ErrMsg = String
powerF :: (ErrT m ~ ErrMsg, ErrorA m) ⇒ Int → Int → m Int
powerF 0 x = pure 1
powerF n x | n >0 = fmap (x ∗ ) (powerF (n−1) x)
powerF _ _ = throwA "negative␣exponent"
```

Unlike the common presentations of power, we have made clear its partiality: it is not defined for the negative values of the exponent. We use the Error *applicative* (the Error monad, which is also an applicative) to throw a String exception. (The reason for using applicatives will become clear soon.)

Recall that applicatives, or applicatives functors [10], represent effectful computations, like monads. In fact, applicatives are a super-set of monads and have a similar interface: Fig. 2. Applicative's pure is the same as monadic return, lifting any value into an applicative computation. Two applicative computations are combined via (<∗>), which is weaker than monadic bind (≫=) in the sense that the result of the first applicative computation cannot be used to choose the next computation to perform. Unlike monads, applicatives are closed under composition. Hereafter we shall assume that the type variables m, i and j represent applicatives; therefore, we will often omit the Applicative (or AppLiftable, see below) constraint. Should the confusion arise, please refer to Fig. 1, which shows complete signatures.

Our task is to specialize powerF to the statically known value of the exponent: to generate the code computing x^n for the given value of n. We are to write a generator, which takes an integer n and the code for "x". Since the exponentiation is partial, we have to decide what to do if that n turns out to be negative. The straightforward specialization of powerF will generate the code that throws the exception. Although the library presented so far, in Fig. 1, lets us generate only pure code, extending the library to build effectful code is straightforward. In fact, the introductory example in §1.1 (described in detail in §2.6) generated code that mutates a vector inplace. One may argue, however, against the straightforward specialization of powerF: if the code computing x^n for the given value of n cannot be generated, we should not be generating any code at all. The generator itself should throw the exception. The problematic negative exponent should be reported and appropriately handled well before running the generated program. The specialized powerF therefore should have the following type

```
spowerF :: (SSym repr, ErrT m ~ ErrMsg, ErrorA m) ⇒
           Int → m (repr Int) → m (repr Int)
```

Recall, repr Int is the type of a code value, the code of an Int expression; m (repr Int) is the type of a computation that will generate Int code and possibly have effects. We represent effects in an applicative rather than in a full monad (we will soon see applicatives that are not monads).

To write spowerF of this signature we have to 'lift' our primitive code generators such as intS and mulS to an applicative m. Figure 1 shows such lifted generators; for example, (∗:) combines two pieces of generated code to build their product, while performing the effects of their generation.

```
spowerF :: (SSym repr, ErrT m ~ ErrMsg, ErrorA m) ⇒
           Int → m (repr Int) → m (repr Int)
spowerF 0 x = int 1
spowerF n x | n > 0 = x ∗: spowerF (n−1) x
spowerF _ _ = throwA "negative␣exponent"
```

The generator spowerF is lucid and obviously correct, matching in appearance the original, 'textbook' code powerF. One may even worry that spowerF matches powerF too closely: in either case, the error is thrown only when the function is applied to both arguments. However, in powerF the second argument, x:: Int, is a value to exponentiate. In spowerF, the second argument has a different type: x :: m (repr Int); it is the code for the value to exponentiate. The argument is supplied at the code generation time rather than at the exponentiation time. We now describe how this argument is supplied and hence how we generate code for functions. After all, the specialized exponentiation should be an Int→ Int function.

Our library provides the primitive lamS for generating functions in the target code (see Figure 1), which uses higher-order abstract syntax (HOAS), relying on Haskell functions to represent the bodies of target functions. For instance, the code for the twice eta-expanded addition is generated by

```
exS2 ::  repr  (Int → Int→ Int)
exS2 = lamS(\x → lamS (\y → addS `appS` x `appS` y))
```

In HOAS, bound variables in the target code are represented by Haskell variables, of the type repr t. We get to use the familiar Haskell notation for target-code functions, with human-readable variable names and with the automatic α-conversion. For that reason, HOAS is popular in code generation; for a good early example see Xi et al. [7]. The tagless-final approach uses HOAS too [9]; that paper (and the accompanying code) describes the instances of LamPure for our two concrete realizations, R and C. The latter instance lets us see the generated code; for example, runCS exS2 gives "\x_0␣→ ␣\x_1␣→ ␣x_0␣+␣x_1" (the C interpreter makes its own variable names). The alternative to HOAS is De Bruijn indices, which were also used in [9]. One would not want to write more than a couple of lines of code with De Bruijn indices.

We stress that we pursue the so-called pure generative approach, which treats the generated code as a black box (See the related work section for other approaches and their comparison). For us, the representation of code values is opaque; we may only combine code values. Therefore our library deliberately offers no facilities to examine the generated code.

To finish our task of specializing powerF we want to write lamS (\x → spowerF n x), but it will not type. The argument of lamS is a function that should return a code value, of the type repr Int. However, spowerF is an effectful generator, of the type m (repr Int). We have to 'lift' lamS to the applicative in which the body of the function was generated. The effect of generating the body should be allowed to 'propagate through the binder': if the generation of the body of the function terminates with an exception, the same exception should terminate the generation of the whole function. Such a lifting to an applicative is the major innovation of the paper; it is here where our code generating library differs sharply from the state of the art, such as MetaOCaml, Template Haskell or Scala LMS. The main problem to overcome is that the 'effect propagation' threatens well-scopedness. Although no ill-scoped code can come from a simple string exception, an exception that carries open code may well lead to code with unbound variables. This problem of permitting arbitrary effects and statically ensuring well-scopedness is a difficult one. The solution has to be involved.

The combinator lam in Figure 1 produces the code of an a→ b function in an applicative m ∘ i, which is a composition of two applicatives m and i. (The applicative m may be, and often is, a monad. The composition m ∘ i however is generally not a monad.) One may think of m as representing an effect of code generation such as throwing exceptions and of i as representing the (type) environment describing the free variables that may occur in the generated code. To put it differently, i denotes the effect of generating target-code variables. The argument of lam is to build the body of the function, in the applicative m ∘ i' where i' is itself the composition i ∘ j. The applicative j represents the environment for the fresh free variable that will be bound by the lam. The composition i ∘ j is in effect the concatenation of the corresponding type environments.[5] The applicatives i and j must be AppLiftable, see Fig. 2, with an additional law app_pull that makes their composition (partially) commute, to support the familiar exchange rule for the components of type environments. (AppLiftable applicatives are also closed under composition. The Identity and the Reader applicatives are AppLiftable.) The generator for the body of the function receives as its argument the value (i ∘ j) (repr a) representing the bound variable. The combinator var lifts the variable to the type (m ∘ (i ∘ j)) (repr a) so that it can be combined with other generators. The reasons for such an intricate type of lam will become clear much later, in §4. For now, we point out the similarity with runST of Haskell's ST monad. The higher-rank type of lam prevents 'leaking' of bound variables, just like the type of runST, namely (∀s. ST s a) → a, prevents leaking of references created within its region.

We now have all the ingredients to complete the task: the following is the generator of the exponentiation function specialized to the given exponent. Its inferred type makes it clear that the result is either the code for the Int→ Int function, or an exception.

```
spowerFn :: (LamPure repr, SSym repr, AppLiftable i , ErrorA m, ErrT m ~ ErrMsg) ⇒
      Int → (m ∘ i ) (repr (Int → Int ))
spowerFn n = lam (\x → spowerF n (var x))
```

To run the generator and see the generated code, we instantiate m to be Either ErrMsg (a particular error applicative) and i to be the Identity. The accompanying code, file `TSPower.hs`, includes several sample specializations.

2.2. Tracing

To illustrate the remaining core part of our library, weaken, we need another example. It will also demonstrate code generation with IO, to print a trace. The tracing is not part of the library; rather, it is easily defined by the user as

```
trace :: String → (IO ∘ i ) a → (IO ∘ i ) a
trace msg m = liftJ (putStrLn msg) *> m
```

using the Haskell Prelude's action putStrLn msg :: IO () to print the msg on the standard input. The IO () action is lifted to the (IO ∘ i) () applicative with liftJ, which is also user-definable. Our library nevertheless provides it for convenience, see Figure 1. Since putStrLn msg produces the dummy result () not used in subsequent computations, IO in this action can be treated as an applicative; liftJ hence lifts an applicative action to a 'richer' applicative. §2.3 shows off code generation with a truly monadic rather than a mere applicative effect.

[5] To some extent i is similar to an environment classifier [23]: both describe a set of free variables in the generated code. However, environment classifiers are coarse-grained: new bindings introduced by lam do not affect the classifier.

The user-defined addt below logs all generated additions; the trace is emitted when the code is generated, rather than when it is executed.

```
addt :: (IO ∘ i)(repr Int) → (IO ∘ i)(repr Int) → (IO ∘ i)(repr Int)
addt x y = trace "generating␣add" (x +: y)
```

The simplest example generates the code for the addition function, and logs that fact:

```
ex2 :: (IO ∘ i) (repr (Int → Int → Int))
ex2 = lam (\x → lam (\y → weaken (var x) `addt` var y))
```

From the type of lam in Figure 1 we deduce the types

```
x :: (IO ∘ (i ∘ j1)) (repr Int)
y :: (IO ∘ ((i ∘ j1) ∘ j2)) (repr Int)
```

where j1 is the AppLiftable introduced by the outer lam and j2 comes from the inner lam. That is, x and y both denote integer variables in the generated code, but in different type environments. The environment of y is longer. Therefore, to use x and y in the same expression, we need to make the type of x the same as the type of y, that is, to *weaken* x. Weakening asserts that a variable in a type environment is a variable in any longer environment. The explicit weaken is a chore, which can be automated in many cases with weakens – the method in the type class Extends m n that checks that the applicative type n is a weakened, by 0 or more steps, version of m. So, our example can be re-written as

```
ex2 = lam (\x → lam (\y → weakens (var x) `addt` weakens (var y)))
```

Since the composition weakens ∘ var occurs frequently, it is given the name vr. The example takes the final form

```
ex2 = lam (\x → lam (\y → vr x `addt` vr y))
```

On the simplest examples of power and addition we have seen the main components of our code-generation framework. The accompanying code, file `TSEx.hs`, has many more examples, some significantly more complex. The next sections will show generators where the danger of scope extrusion is real. The coming examples were not possible to generate in the existing mainstream frameworks such as MetaOCaml or TH while statically assuring the absence of scope extrusion at all times.

2.3. Moving open code

The warm-up example in §2.1 was rather simple, and could be implemented with the existing techniques, such as Mint [21] or a trivial ad hoc extension of $\lambda^{\oslash}$ [22]. The code-generation library introduced in §2.1 permits however the manipulation of open code in *any* applicative. Furthermore, it permits not just applicative but also monadic effects. The generation applicative can truly be anything, far beyond throwing text-string exceptions. In this section we instantiate the generation applicative to that of reference cells, and demonstrate monadic effects of storing open code and retrieving it across the binders, while statically ensuring the generation of well-scoped code. We demonstrate that scope extrusion becomes a type error. That is beyond any existing higher-order code-generation approach with safe code motion.

Before we get to our main example, we have to clear a hurdle. We intend to use IORef reference cells, whose creation requires monadic, not just applicative effects. We describe this problem and its simple solution in detail, to illustrate that our applicative code generation framework indeed permits arbitrary monadic effects on open code. We will also gain the intuition about the central role of applicatives. The first example stores integer code in a reference cell and generates the code by reading it from the cell. First we need to create a reference cell holding, for example, a literal 1: newIORef (intS 1) :: IO (IORef (repr Int)). This IO action can be lifted to the generating applicative (IO ∘ i):

```
ref1 :: (IO ∘ i) (IORef (repr Int))
ref1 = liftJ $ newIORef (intS 1)
```

The only way we can read from our reference cell using the applicative interface is

```
readBad :: (IO ∘ i) (IO (repr Int))
readBad = fmap readIORef ref1
```

which however has a wrong type for the Int code generator. To transform readBad to the desired type (IO ∘ i) (repr Int), we either need an operation IO (repr Int) → repr Int (which the IO monad deliberately does not provide), or attempt

```
join ∘ tr $ readBad where
 tr :: (IO ∘ i) (IO (repr Int)) → (IO ∘ i) ((IO ∘ i) (repr Int))
 tr = fmap liftJ
 join :: (IO ∘ i) ((IO ∘ i) (repr Int)) → ((IO ∘ i) (repr Int))
 join = ???
```

Alas, the needed join does not exist since (IO ∘ i) is generally not a monad. Manipulating reference cells hence requires monadic, not just applicative effects.

Recall that IO ∘ i is a composition of IO and an applicative i; that is, modulo the annoying but inevitable J and unJ newtype wrapping, (IO ∘ i) (repr Int) is just IO (i (repr Int)) – which is the type that lets us do arbitrary IO operations. Our example of creating a reference cell and generating code by reading from it is implemented as follows, taking the full advantage of monadic bind and the do-notation.

```
withRef :: a → (IORef a → (IO ∘ i) (repr w)) → (IO ∘ i) (repr w)
withRef v k = J $ do
  r ← newIORef v
  unJ $ k r

readGood :: (IO ∘ i) (repr Int)
readGood = withRef (intS 1) $ \r → J $ do
  v ← readIORef r
  return $ pure v
```

To illustrate how monadic operations play together with applicative generators, e.g., lam, we extend the example to read from the reference cell as we generate the body of a function:

```
readBetter :: (IO ∘ i) (repr (Int → Int → Int))
readBetter =
  lam (\x →
      withRef x (\r →
       lam (\y → var y +: (weaken ∘ J $ readIORef r))))
```

(The generated code is \x_0 → \x_1 → x_1 +x_0). It is instructive to see what happens if we attempt to 'leak out' a bound variable, that is, write y into the reference cell r accessible outside the inner lam:

```
readFail :: (IO ∘ i) (repr (Int → Int → Int))
readFail =
  lam (\x →
      withRef x (\r →
        lam (\y → var y +: (weaken ∘ J $ do
                                         v ← readIORef r
                                         writeIORef r y
                                         return v))))
```

Type checking of this code fails, with the error message[6]:

```
Occurs check: cannot construct the infinite  type: i ˜ i ∘ j
Expected type: (∘ ) i j (repr Int)
  Actual type: (∘ ) (i ∘ j) j1 (repr Int)
Relevant bindings include
  v :: (∘ ) i j (repr Int)
  y :: (∘ ) (i ∘ j) j1 (repr Int)
  r :: IORef ((∘ ) i j (repr Int))
  x :: (∘ ) i j (repr Int)
  readFail :: (∘ ) IO i (repr (Int → Int → Int))
In the second argument of writeIORef, namely y
In a stmt of a do block: writeIORef r y
```

The message explicitly names the culprit: “the second argument of writeIORef, namely y”, and the problem: the attempt to put y into a reference cell that stores code in a “shorter” applicative. That is, y will escape its scope. The error message gives a hint at the role of the type (m ∘ i) (repr a) in our framework: this type, isomorphic to m (i (repr a)) lets us do arbitrary monadic effects in the monad m on a target code expression, but that expression must always be wrapped in an applicative i. In other words, we can manipulate open code but only the code that is *explicitly* marked with the typing environment for free variables. It is this explicit marking that prevents leaking out of the variable y in readFail: the bound variable y has the type ((i ∘ j) ∘ j1) (repr Int), where j1 represents the environment for the fresh variable of the inner lam. The variable y cannot be stored in a reference cell whose type does not provide for that j1. We get the first glimpse at how types tell and enforce the variable scope in our framework. We return to this point in §3.

We now turn to our main example of *assertion-insertion*, a special case of *if-insertion*. It has been described in detail in [22], which argued that in practice an assertion has to be inserted beyond the closest binder. Such an insertion was left to future work – which becomes the present work in this section. For the sake of exposition, we first demonstrate open code motion under a binder; a simple modification will move beyond the binder.

We start by extending our DSL with assertPos and the integral division (/:), see Figure 3. The tagless-final approach makes extending an EDSL trivial, by defining a new type class and its instances for the existing interpreters, R and C in our case. The expression assertPos test m will generate an assertion statement

[6] It is well-known that (type) error messages are the most troublesome aspect of embedded DSLs: they report a problem with the DSL code in terms of the (often significantly more complex) meta-language. For the discussion of the problem and some of its solutions, please see [8, §7] and especially [24–26].

```
class AssertPos repr where
  assertPosS :: repr Int → repr a → repr a
assertPos :: m (repr Int) → m (repr a) → m (repr a)

class SymDIV repr where divS :: repr (Int → Int → Int)
infixl 7 /:
(/:) :: m (repr Int) → m (repr Int) → m (repr Int)
```

Figure 3: Library extension: assertion statement and integer division

in the target code, to check that the code generated by test produces a positive integer. If the assertion checks, the code m is run; otherwise the program is abnormally terminated.

Our goal is to write a guarded division, making sure the divisor is positive. The first version is

```
guarded_div1 :: m (repr Int) → m (repr Int) → m (repr Int)
guarded_div1 x y = assertPos y (x /: y)
```

to be used as

```
lam (\y → complex_exp +: guarded_div1 (int 10) (var y))
```

The first version is unsatisfactory: we check for the divisor right before doing the division. If the divisor is zero, we crash the program wasting all the (potentially long) computations done before. The error should be reported as soon as possible, when we learn the value of the divisor. We have to move the assertion code.

We can accomplish the movement with reference cells. We allocate a reference cell holding a code-to-code transformer, originally identity. As the code is generated, the cell accumulates post-hoc transformations. At the point where the assertions are to be inserted, the assertion locus, the resulting transformer is retrieved and applied to the generated code. The code below implements the idea, using IO and its reference cells IORef. The code follows the pattern of the simple examples at the beginning of the section.

```
assert_locus :: (IORef ((IO ∘ i) (repr a) → (IO ∘ i) (repr a)) → (IO ∘ i) (repr a))
                → (IO ∘ i) (repr a)
assert_locus f = J $ do
  assert_code_ref ← newIORef id
  mv ← f assert_code_ref
  transformer ← readIORef assert_code_ref
  transformer (pure mv)
```

We re-define guarded division to insert the positive divisor assertion at the given locus, that is, to modify the contents of the cell locus, composing the current transformer with assertPos test.

```
add_assert :: IORef (m a → m a) → (m a → m a) → (IO ∘ i) ()
add_assert locus transformer =
  liftJ $ modifyIORef locus (∘ transformer)

guarded_div2 locus x y =
  add_assert locus (assertPos y) *> x /: y
```

Here is the example:

```
exdiv2 = lam (\y → assert_locus $ \locus →
  complex_exp +: guarded_div2 locus (int 10) (vr y))
```

The generated code

```
\x_0 → assert (x_0 >0) (complex_code +(div 10 x_0))
```

demonstrates that assert is inserted before the complex_code, right under the binder, as desired. We stress that the code transformer, assertPos y, includes the open code. We do store functions that contain open code. However, the reference cell that accumulates the transformer has been manipulated completely inside a binder. There is no risk of scope extrusion then. The above example is hence implementable in $\lambda^{\oslash}$[22].

In the second, main part of our example, we change guarded_div2 slightly so that it may insert the assertion even beyond the binder. Such a movement of the open code has not been possible before, while ensuring well-scopedness. The modification to guarded_div2 is simple: adding the generic weakens. The inferred signature tells the difference

```
guarded_div3 :: (Extends m (IO ∘ i)) ⇒
      IORef (m (repr a) → m (repr a))
      → (IO ∘ i) (repr Int) → m (repr Int) → (IO ∘ i) (repr Int)
guarded_div3 locus x y =
     add_assert locus (assertPos y) *> x /: weakens y
```

The divisor and the dividend expressions do not have to be in the same environment; the environment of the dividend, (IO ∘ i), may be weaker, by an arbitrary amount. The generalized guarded_div3 can be used in place of guarded_div2 in exdiv2. A more general example becomes possible:

```
exdiv3 = lam (\y → assert_locus $ \locus →
            lam (\x → complex_exp +:
                          guarded_div3 locus (vr x) (vr y)))
```

The generated code

```
\x_0 → assert (x_0 >0) (\x_1 → complex_code +(div x_1 x_0))
```

shows the variable positivity assertion is inserted right after the binding for that variable, at the earliest possible moment – exactly as desired. Thus assertPos (var y) has really been moved across the binder. If we make a mistake and switch x and y

```
lam (\x → assert_locus $ \locus →
  lam (\y → complex_exp +:
              guarded_div3 locus (vr x) (vr y)))
```

attempting to move assertPos (var y) beyond the binder for y, the type checker reports a problem

```
    Could not deduce (Extends (IO ∘ ((i ∘ j) ∘ j1)) (IO ∘ (i ∘ j)))
      arising from a use of 'vr'
    In the third argument of 'guarded_div3', namely '(vr y)'
```

telling us that we have attempted to move y to a smaller binding environment. In other words, the y binding leaks. Scope extrusion indeed becomes a type error. (The generated code is shown in full as regression tests of the generators, in the code accompanying the article. The file `Anaphora.hs` includes other examples of code motion with mutable cells.)

The example is simplistic but extensible, all the way to code generation with constraints (supercompilation). The locus, describing where the assertion is to be inserted, could be bundled with the bound variable in a new data structure, so that it does not have to be passed around separately. Alternatively, one could use a form of dynamic binding, which could be implemented via the continuation monad as the generating applicative. Code generation with continuations is described in §2.5.

2.4. Generating `let`

When we first introduced the library in §2, we used the type repr Int for an expression that will produce an integer-valued target code. We have just seen that to ensure the absence of scope extrusion in the presence of effects such as mutation, we have to use the more complex type i (repr Int) for target code, with the applicative i standing for the type environment describing free variables in the target code. Such an explicit marking of target code types with the type environment can cause problems when generating code with arbitrarily many new variables, whose number is determined only at run-time. What should be the type environment annotation if even the number of free variables in it is not statically known? This is indeed the show-stopper for some stage calculi, e.g., [27]. This section demonstrates that our library has sufficient type-environment polymorphism to express generators that introduce arbitrarily many new variables. The polymorphism does not impose high cost: writing such generators is no more complex than writing polymorphically recursive functions, which are well-supported in Haskell (and, recently, in OCaml). This section also introduces let-statements to express sharing in the generated code; combining sharing with code motion is the subject of the next section.

Our running example, introduced in [28], is determining the n-th element of a Fibonacci-like sequence with the user-specified first two elements. To be precise, the goal is to build the efficient code that takes the first two elements and returns the n-th element, for the statically known n. The generator is to do IO (this time, to print the trace of generating addition, to demonstrate that the number of additions is linear in n). Getting a bit ahead we note that despite the apparent triviality, the example could not be implemented before with the same static guarantees: some frameworks permit effectful generators but not the statically unknown number of free variables; some others have environment polymorphism but no effects.

The generator that computes the n-th element of the Fibonacci-like sequence can be written simply and purely:

```
gib :: repr Int → repr Int → Int → repr Int
gib x y 0 = x
gib x y 1 = y
gib x y n = gib y ((addS `appS` x) `appS` y) (n−1)
```

To print the trace of additions we insert the tracing version of the addition generator addt from §2.2. The inferred signature reflects the effectful code generation (in the applicative IO ∘ i):

```
gib :: (IO ∘ i) (repr Int) → (IO ∘ i) (repr Int) → Int → (IO ∘ i) (repr Int)
gib x y 0 = x
gib x y 1 = y
gib x y n = gib y (addt x y) (n−1)
```

```
gibn n = lam (\x → lam (\y → gib (vr x) (vr y) n))
−− \x → \y → (y +(x +y)) +((x +y) +(y +(x +y)))
```

Evaluating gibn 5 prints "generating add" seven times rather than the expected four. The generated code shown in the comments has lots of duplication: x+y appears three times. When we first invoke gib (vr x) (vr y) 5, the second argument is the code for a simple variable. On the recursive invocation, that argument is the code for adding x and y. This complex expression is incorporated into further generated code, leading to duplication. The resulting code becomes exponentially larger and slower with n. To overcome this serious performance problem, we should avoid repeated copying (i.e., inlining) the expression x+y; rather, we should arrange for evaluating x+y only once, and then share its result.

Our library in Figure 1 provides the primitive generator of let-expressions in the target code, and the effectful generator let_, whose two arguments generate the expression to share, and the let-body. Whereas the generator using Haskell's let

```
let x = int 1 +: int 2 in x ∗: x
−− (1 +2) ∗ (1 +2)
```

yields the code (shown underneath in the comments) with the obvious code duplication, the generator producing the target-code **let**

```
let_ (int 1 +: int 2) $ \x → var x ∗: var x
−− let z_0 = 1 +2 in z_0 ∗ z_0
```

shares the result of the addition without re-computing it. The code generation for the addition also happens once in the latter case and twice in the former case (which is noticeable if the addition generator is effectful, e.g., printing a trace message).

If we add let_ in the last clause of gib to share the result of adding x and y

```
gib :: (IO ○ i) (repr Int) → (IO ○ i) (repr Int) → Int → (IO ○ i) (repr Int)
gib x y 0 = x
gib x y 1 = y
gib x y n = let_ (addt x y) $ \z → gib (weaken y) (var z) (n−1)
```

the trace of gibn 5 tells that only four additions are generated, as expected. The resulting code, in the A-normal form,

```
 \x_0 → \x_1 →
let z_2 = x_0 +x_1 in
let z_3 = x_1 +z_2 in
let z_4 = z_2 +z_3 in
let z_5 = z_3 +z_4 in z_5
```

exhibits no duplication. The simplicity of just adding let_ seems magical. The example is indeed less trivial than it appears to be. Let's examine the evaluation of gibn 5 step-by-step. First, the expression reduces to gib (vr x) (vr y) 5 :: (IO ○ i) (repr Int) where the applicative i stands for the type environment containing the variables x and y. That expression in turn reduces to

```
let_ (addt (vr x) (vr y)) $ \z → gib (weaken (vr y)) (var z) (n−1)
```

The let_ generator introduces a new free target variable z and generates the let-expression binding z to x+y. That free variable is passed to the generator of the let-expression body, which in our case recursively invokes gib. Therefore, the type of gib (weaken (vr y)) (var z) (n−1) is now (IO ∘ (i ∘ j)) (repr Int) where j is the new applicative representing the type environment containing the introduced variable z. To type check gib therefore we first need environment polymorphism: a way to describe the type environment with arbitrarily many free variables. Second, we need polymorphic recursion, a way to recursively invoke a polymorphic function at a different type. Our library has both necessary features. Recall that (i ∘ j) in the above gib type stands for the original type environment of gib extended with the binding for z; the latter is described by the applicative j. Also recall that if i and j are applicatives their composition i ∘ j is itself an applicative and so the type (IO ∘ (i ∘ j)) (repr Int) of the recursive invocation of gib is an instance of the polymorphic type ∀i. Applicative i ⇒ (IO ∘ i) (repr Int). Thus the applicative i reflects the progressively richer type environment. Polymorphic recursion has been available in Haskell since Haskell98. Because of the polymorphic recursion, the type signature of gib is no longer optional.

We have just seen the generation of the statically unknown number of let-expressions and hence of the statically unknown number of new variables, while using effects. Such code generators statically assuring well-scoped and well-typed result code have not been demonstrated before. The effect of code generators has been simple – printing a trace message. We now turn to a more serious effect to combine generating let-expressions with code motion.

2.5. Inserting `let` across binders

We have seen the examples of what looked like 'patching-up' the already generated code, to insert assertions (whose contents are determined after the generation is complete). One may view such patching-up as code motion. The ultimate, and the most difficult task is inserting not just assert statements but binding forms such as **let** and loop statements – moving not just open code but binders. We describe let-insertion now and loop-insertion in the next section.

The significance of let-insertion, as we have just seen, is generating code with the explicit sharing of the result of a sub-expression, eliminating code duplication. If the target code is imperative, controlling code duplication is not only desirable but necessary. For that reason, let-insertion is used extensively in partial evaluation, staging [28] and other meta-programming. The paper [22] argued for the need to let-insert across binders and posed several such cases as open problems. We now demonstrate the solution, with well-scopedness safety guarantees.

Just like the assert statements, we often wish to insert **let** in an earlier part of the code, to accomplish what is called an invariant code motion (moving code out of the loop or a function). Inserting **let** is significantly more complex than inserting assert. Recall from §2.3 that assert (x>0) body appearing in the generated code was produced not in the straightforward way by first generating the condition x>0 and then generating body. Rather, we started by generating body and in the process we discovered the need to assert that x is positive. Hence the code for x>0 was produced as we were generating the body. Let-insertion, generating **let** z= exp **in** body, follows the similar pattern: we start by generating body, discover an exp to share, insert the let-expression and continue generating

```
type CPSA w m a -- abstract
instance Applicative (CPSA w m)

runCPSA :: CPSA a m a → m a
runJCPSA :: (CPSA (i a) m ◦ i) a → (m ◦ i) a
runJCPSA = J ◦ runCPSA ◦ unJ

resetJ  :: (CPSA (i a) m ◦ i) a → (CPSA w m ◦ i) a

genlet  :: (CPSA (i0 (repr a)) m ◦ i0) (repr a) →
           (CPSA (i0 (repr w)) m ◦ i) (repr a)

-- growing the hierarchy
liftCA  :: m a → CPSA w m a
liftJA  :: (m ◦ j) a → (CPSA w m ◦ j) a
```

Figure 4: The interface for let-insertion

of the body. However, we seemingly cannot start generating body before exp is known: without exp the binder cannot be generated, thus z that may appear in body does not yet exist. The answer to the quandary was found in the partial evaluation community a ong time ago: writing the code or the generator in the continuation-passing style (CPS) [29] (for the detailed explanation, see [4, Section 3.1]) – or else using control operators [30]. Alas, the ordinary CPS [18] cannot be used for let-insertion beyond binders (that is, cannot be used for the invariant code motion) [22]. Our library provides a new CPS hierarchy, called CPSA, which is applicative rather than monadic. It lets us implement, in the file `TSCPST.hs`, the let-insertion interface shown in Figure 4. The implementation is outside the core of our library, relying *only* on the interface of Figure 1 but not on any details of its implementation.

The interface defines an applicative CPSA w m where w is the answer type and m is a (base) applicative. The latter can be Identity, IO, or another CPSA w' m'. Thus CPSA may be iterated, giving us a hierarchy and the possibility of let-insertion beyond many bindings. The combinator for let-insertion itself is called genlet. It receives an expression to let-bind and evaluates to the let-bound variable. The place to insert the **let** form is marked by resetJ. An example should make it clear:

```
resetJ $ lam (\x → var x +: genlet (int 2 +: int 3))
-- let z_0 = 2 +3 in
-- \x_1 → x_1 +z_0
```

with the generated code shown in comments. The let-expression indeed occurs outside the generated function: we have moved the expression 2+3, which does not depend on the function's argument, outside the function's body. The let-insertion point may be arbitrarily many binders away from the genlet expression:

```
resetJ $ lam (\x → lam (\y →
  var y +: weaken(var x) +: genlet (int 2 +: int 3)))
-- let z_0 = 2 +3 in
-- \x_1 → \x_2 → (x_2 +x_1) +z_0
```

The right-hand-side of the binder may contain variables; that is, we may let-bind open code. Here the type-checker watches that we do not move such open expressions too far. For example, the following code attempts to let-bind var x +: int 3 at the place marked by resetJ, which is outside the x's binder.

```
resetJ $ lam (\x →
  (lam (\y → var y +: weaken (var x) +:
              genlet (var x +: int 3))))

    Expected type: i0 (repr0 Int)
      Actual type: (∘ ) i0 j (repr0 Int)
    In the first argument of ‘var’, namely ‘x’
    In the first argument of ‘genlet’, namely ‘(var x +: int 3)’
```

The type checker reports the error, pointing out the binder whose variable escapes (attempted to be smuggled to a shorter environment, without j). We must move the insertion point within that binder, moving the resetJ:

```
lam (\x →
  resetJ (lam (\y → var y +: weaken (var x) +:
                     genlet (var x +: int 3))))
-- \x_0 → let z_1 = x_0 + 3 in
-- \x_2 → (x_2 + x_0) + z_1
```

One may use several genlet expressions and even nest them:

```
lam (\x → resetJ (lam (\y →
   int 1 +: genlet (var x +: genlet (int 3 +: int 4))
   +: genlet (int 5 +: int 6))))
-- \x_0 → let z_1 = let z_1 = 3 + 4 in
--                   x_0 + z_1 in
--         let z_2 = 5 + 6 in
--         \x_3 → (1 + z_1) + z_2
```

The result is not quite satisfactory: since one of the let-bound expressions contains the variable x, we must insert resetJ under the binder for x, marking the let-insertion point for all genlet. Whereas genlet (var x +: ...) truly cannot be inserted any further without scope extrusion, (int 5 +: int 6) is closed and can be let-bound outside of the outer lam (\x→ ...).

To permit multiple let-insertion at multiple points, we have to use the CPSA hierarchy, with the applicative CPSA w (CPSA (i0 (repr w1)) m). The nested CPSA lets different genlets move to different places. We only need to indicate which genlet goes to which place using liftJA, which may be used repeatedly (the more liftJA combinators, the wider the scope of the corresponding genlet):

```
lam (\x → resetJ (lam (\y →
  int 1 +: genlet (var x +:
             (liftJA $ genlet (int 3 +: int 4)))
       +: (liftJA $ genlet (int 5 +: int 6)))))
-- let z_0 = 3 + 4 in
-- let z_1 = 5 + 6 in
-- \x_2 → let z_3 = x_2 + z_0 in
--        \x_4 → (1 + z_3) + z_1
```

We have demonstrated generating code in which different let-bound expressions are moved to different places, as far as possible, crossing an arbitrary number of target code binders, including the binders introduced by the earlier genlet.

2.6. Loop tiling

This section presents the case study of using our library for common high-performance computing loop optimizations: strip mining, loop interchange and loop tiling. Performing loop interchange with static assurances of well-typedness and well-scopedness of the generated code was posed as an open problem by Cohen in [5]. This section presents the first solution.

Our running example is matrix-vector multiplication[7]: multiplying the matrix a with n rows and m columns by the vector v with the result in the vector v'.

$$v'_i = \sum_j a_{ij} v_j$$

The standard Haskell implementation of the textbook code is as follows (see the complete code in the file `TSLoop.hs`).

```
mvmul_textbook n m a v v' = vec_clear_ n v' ≫
 forM_ [0,1.. m−1] (\j →
  forM_ [0,1.. n−1] (\i →
   vec_addto_ v' i =≪
     mat_get_ a i j ∗ vec_get_ v j ))
```

The operations vec_get_ and mat_get_ retrieve a vector/matrix element by its index. We assume that n is much greater than m. Once a_{00} is accessed, memory loads the whole cache line, that is, elements a_{00} through a_{07} (with the cache line 8*8 bytes). Alas, by the time the algorithm needs a_{01}, at the next major iteration, it will be already evicted. So mvmul_textbook performs poorly since it fails to take advantage of the memory bandwidth's bringing in several array elements at a time. A tiled program handles the array one chunk (of size b) at a time.

```
mvmul_tiled b n m a v v' = vec_clear_ n v' ≫
 forM_ [0, b.. m−1] (\jj →
 forM_ [0, b.. n−1] (\ii →
  forM_ [jj ,jj +1..min (jj +b−1) (m−1)] (\j →
   forM_ [ii ,ii +1..min (ii +b−1) (n−1)] (\i →
    vec_addto_ v' i =≪
      mat_get_ a i j ∗ vec_get_ v j ))))
```

Since a tile is small enough, the element a_{01}, brought along with the requested a_{00}, will not be evicted when it is needed at the $j = 1$ iteration. Tiling improves spatial locality, and is one of the basic optimizations in high-performance computing. It is not a general-purpose optimization since it heavily relied on the fact that the evaluations of loop bodies are uncorrelated. Tiling is converting each i and j loop into a nested pair of loops, followed by loop interchange, pulling the ii loop right after the jj loop. The body of the loops remains exactly the same as before; it is executed the same number of times – but in a different pattern.

The tiled code looks more complex; it is easy to make a mistake when tiling by hand. We need automation. We need automation even more when we will

[7] Matrix-matrix multiplication benefits more from loop tiling, but it is less suitable for exposition.

be combining loop tiling with scalar promotion, partial unrolling and other optimizations. Our task thus is to generate ordinary and tiled loop nests in a modular way.

The starting point is converting mvmul_textbook to a generator:

```
mvmul0 n m a v v' = vec_clear (int n) v' ≫:
 loop (int 0) (int (m−1)) (int 1) (lam $ \j →
  loop (int 0) (int (n−1)) (int 1) (lam $ \i →
   vec_addto (weakens v') (vr i ) =≪:
     (mat_get (weakens a) (vr i ) (vr j ) `mulM`
      vec_get (weakens v) (vr j ))))
```

The code is the straightforward staging of mvmul_textbook assuming for a bit of simplicity that the dimensions n and m are known statically. It clearly corresponds to the textbook code and seems 'obviously' correct. Here mat_get and vec_get are the generators of matrix/vector indexing operations. There may be several implementations for loop, the generator of a loop with the given lower and upper bounds and the step. The straightforward one generates forM_ [lb,lb+step..ub], which gives back mvmul_textbook. The second implementation does the so-called 'strip mining', stripping a loop into blocks and hence converting a single loop into two, iterating over blocks (by the statically known factor b) and then within a block:

```
loop_nested :: Int → Int → Int→ (m ∘ i) (repr (Int → IO ()))
      → (m ∘ i ) (repr (IO ()))
loop_nested b lb ub body =
     loop (int lb ) (int ub) (int b) (lam $ \ii →
      loop (var ii ) (min_ (var ii +: int (b−1)) (int ub)) (int 1)
       (weakens body))
```

This generator is written once and for all, in terms of the primitive loop, by a domain expert, and put in a library. If we just replace loop with loop_nested b in mvmul0, keeping everything else the same, we obtain potentially faster, strip-mined code.

Yet another implementation of the loop generator is to split a loop in two, as in strip mining, and hoist the first loop. The only change to loop_nested is insloop, which, like genlet from §2.5, inserts the loop at some position, to be indicated by resetJ.

```
loop_nested_exch b lb ub body =
     let_ (insloop (int lb ) (int ub) (int b)) (\ii →
      loop (var ii ) (min_ (var ii +: int (b−1)) (int ub)) (int 1)
       (weakens body))
```

Using loop_nested_exch instead of loop in mvmul0 with resetJ at the top – but keeping exactly the same loop body – results in the generation of the tiled loop code just like mvmul_tiled. (See the accompanying code for the full details.) Truly, tiling is strip mining with the loop interchange.

We have demonstrated the step-wise development of the optimized iterative code. We write the loop body once, and apply various transformations (strip-mining, tiling, etc) many times. In particular, we interchange loop bodies, moving open code with binders across other binders.

3. Implementation

This section outlines the implementation of the interface in Figure 1. The full implementation is in the file `TSCore.hs` in the accompanying code.

The two representations of the target code, the data types R and C are as follows:

```
newtype R a = R{unR :: a}
newtype C a = C{unC :: VarCounter → Language.Haskell.TH.Exp}
```

R is just the identity functor; C represents the target code as a Haskell AST, as reflected in the TH.Exp data type (VarCounter is used internally for generating fresh names). Making R and C instances of SSym, LamPure, SymLet, etc. is simple; see the exposition of the tagless-final approach [9] for a detailed discussion.

The C-representation, unlike R, denotes the truly future-stage code, to be compiled later. The code is represented as an untyped Template Haskell TH.Exp. Nevertheless, the C-representation itself is typed (extrinsically). A generator polymorphic over repr, such as exS1 in §2, abstracts the differences between R and C, ensuring at the same time that the generated code is well-typed [9] since R can generate only well-typed 'code'.

We now describe the main ideas behind the implementation of two particularly challenging parts of our library: lam (and the similar let_) generator of binding forms that permit arbitrary effects while ensuring a well-scoped result; and the CPSA hierarchy for let-insertion across binders.

3.1. Challenges of generating binding forms with effects

Before showing the implementation of lam, we explain why it has such a complex and strange type. Recall that the primitive generator of abstractions lamS from §2.1

```
lamS :: (repr a → repr b) → repr (a→ b)
```

permitted no effects during code generation. To accommodate effects, the type should, at first blush, be

```
lamM :: (repr a → m (repr b)) → m (repr (a→ b))
```

Such a generator has two problems. First, it cannot be written in terms of lamS. Every attempt

```
lamM body = ... lamS (\x → body x)
```

runs into the stumbling block: body must be applied to a value of the type repr a whose only source is x passed by lamS to its argument. However, lamS expects that argument to return repr b rather than m (repr b). The second problem with lamM is allowing us to write the code with scope extrusion

```
badM = do
  r ← newIORef (intS 0)
  lamM $ \x → do
                writeIORef r x
                return (addS (intS 1) x)
  readIORef r
```

resulting in an unbound variable. The first problem of lamM may seem minor. It is however symptomatic: lamS is the generator of abstractions in the target language, assuring by construction the well-scoped and closed resulting code. Not being able to use lamS is ominous.

Let us again look at lamS and the example exS2 of its use from §2.1:

```
lamS :: (repr a → repr b) → repr (a→ b)
exS2 = lamS(\x → lamS (\y → addS `appS` x `appS` y))
```

Haskell variables x and y represent free variables in the target code, of the type repr Int. The result of exS2 is closed code, of the type repr (Int→ Int→ Int). Clearly, the code type does not tell whether the target code is closed. Nevertheless, effect-free generators like exS2 guarantee the well-scoped and closed result, because of the strict region discipline of lamS-bound variables. A free variable introduced by lamS is manipulated only within the dynamic extent of that lamS. Haskell's (metalanguage) *dynamic* environment (of lamS invocations) stands for the target *type* environment.

Effects break the correspondence between the dynamic environment of lamS and the target type environment by letting bound variables escape the dynamic scope of their binders, as badM clearly shows. We have to differentiate open and closed code in their types, to give the type checker enough information to detect scope extrusion and to re-enforce the region discipline. We have to annotate the type of a code value with free variables that may appear in it. We have to make the target type environment explicit in the code type.

The environment (Reader) monad immediately springs to mind: the type of a potentially open target code would be $\gamma \to$ repr a, where the tuple γ describes the types of free variables in the code. Closed code is typed as ()→ repr a. The generator of abstractions with the explicit environment types can then be written as follows:

```
lamH :: (((γ,repr a) → repr a) → ((γ,repr a) → repr b))
          → (γ → repr (a→ b))
lamH body = \γ → lamS (\x → (body var)(γ,x))
  where var = \ (γ,x) → x
```

The variables are hence identified by their offsets in the tuple: De Bruijn indices essentially. The user of our library is spared, however, from programming with the indices because lamH provides and names a function to project a variable from the environment. The generator lamH still operates without effects. However, it is expressed in terms of lamS, and it generalizes to effects.

Introducing generation-time effects becomes straightforward: a generator yielding potentially open code and performing side-effects in the monad m gets the type m (γ→ repr a). In particular, the generator of target abstractions can be written as

```
lamE :: Functor m ⇒ (((γ,repr a) → repr a) → m ((γ,repr a) → repr b))
      → m (γ → repr (a→ b))
lamE body = fmap (\body' γ→ lamS (\x → body' (γ,x))) (body var)
  where var = \ (γ,x) → x
```

It is written in terms of lamS as desired. The lamE code and its type signature desperately call for abstraction. Exposing the representation of the environment as the nested tuple makes it ripe for abuse, e.g., letting the programmer examine the environment and modify it at will and hence break any well-scopedness guarantees (see §4.1 for examples).

The hint at a better abstraction comes from the type of lamE, which needs m to be a mere Functor. The second hint comes from examining the type of effectful generators, m ($\gamma\to$ repr a), which looks like the composition of m and the Reader monad ($\gamma\to$) applied to repr a. Alas, the composition of two monads is not a monad in general. Although the Reader monad ($\gamma\to$) is a 'benign' effect, the composition m ∘ ($\gamma\to$) with an arbitrary monad m is *not* a monad. The reader is encouraged to try to write (≫=) for m ∘ ($\gamma\to$) to see why this is not possible.

However, any monad is an applicative, and applicatives are closed under composition. Hence m ∘ ($\gamma\to$), and generally m ∘ (t1 →) ∘ (t2 →) ... ∘ (tn →), are all applicatives. This fact leads to the final representation for the code type: i (repr a) is the type of the generator that produces potentially open code and has some effects. Effects and the typing environment are both hidden, abstracted away by the applicative i. The generator for the target abstraction is written as follows[8]:

```
lam :: Applicative i ⇒
         (∀ j . Applicative j ⇒ j (repr a) → (i ∘ j) (repr b)) →
         i (repr (a→ b))
lam body = fmap' lamS (body var)
 where var = \x → x        −− j is the Reader applicative

fmap' :: (j a → b) → (i ∘ j) a → i b
```

In the implementation of lam the Applicative j is taken to be the Reader applicative (repr a→). However, the generator body for the body of the abstraction is not allowed, by the quantification over j, to know what j really is. (The reason for the quantification will become clear in §4.) All the body knows is that it receives the value of the type j (repr a) that somehow represents the bound variable and it should produce the abstraction's body of the type (i ∘ j) (repr b), in the applicative (i ∘ j), possibly performing effects denoted by i and using free variables that are somehow bound in i. The detailed structure of the type environment is hidden.

Thanks to the abstraction of the target code binder lamS and of the environment and effects, the code of lam is exceptionally simple. The implementation of let_ is similar. The implementation of fmap' above and of var and weaken in Figure 1 follows from the Applicative laws.

3.2. Challenges of let-insertion

We now describe how the representation of the target code with the explicit type environment makes let-insertion very difficult.

Recall that let-insertion, §2.5, is introducing let-binding forms in the generated code; specifically, making sure that marked sub-expressions are let-bound

[8] The real type of lam, shown in Figure 1, is more elaborate, for the sake of let-insertion, §3.2.

in the resulting code. The following is a typical example:

```
li   e = lam (\x → var x +: genlet e)
tli    = lam (\y → resetJ $ li (var y +: int 1))
-- \y → let z = y+1 in (\x → x+z)
```

with the generated code shown in the comments. The combinator genlet (see Figure 4) 'marks' an expression for let-bindings, and resetJ marks the place where to insert that let-binding.

Just accomplishing the let-insertion regardless of typing is a challenge in itself, as we have seen in §2.5. That section mentioned that the solution, well-known in the partial evaluation community, involves continuation-passing style or delimited control operators. We can program that solution as follows, using the standard CPS monad and the combinator let_S for generating let-expressions in the target code.

```
type CPS w a = (a → w) → w
runCPS :: CPS a a → a
runCPS m = m id

reset  ::  CPS a a → CPS w a
reset  m = \k → k (runCPS m)

genlet_simple  ::  CPS (repr a) (repr a) → CPS (repr w) (repr a)
genlet_simple  e = \k → let_S (runCPS e) (\z → k z)
```

The sample generator below yields the code shown in the comments

```
reset $ int 1 +: genlet_simple (int 2 +: int 3)
-- let z_0 = 2+3 in 1 +z_0
```

The combinator genlet_simple is too simple. For one, it cannot insert let across the binders. For example, we cannot use it in place of genlet in the expression li above. In order to be used in li, the let-insertion combinator should at least have the type

```
(CPS (i (repr w)) ∘ i ) (repr a) → (CPS (i (repr w)) ∘ (i ∘ j )) (repr a)
```

reflecting the fact that the expression e in genlet e is a potentially open expression, whose type environment is described by the applicative i (in the example tli, the function li does receive the open expression (var y +: int 1) as the argument). It is a challenge to make our genlet_simple to conform to the above type. Instead of let_S we need a let-generator of a type like i (repr a) → ((repr a) → i (repr w)) → i (repr w). The problem now looks quite like the one with lamS in §3.1, which we solved by introducing lam, the effectful generator of abstractions. The same approach produces let_ for let-insertion in the presence of effects. If we could use let_ instead of let_S in the definition of genlet_simple, we solve the current problem and also the second one: genlet_simple does not work for tracing, using mutable state or other effects beside the control effect for let-insertion.

Alas, it is not that simple. It is exasperating to generalize genlet_simple from repr a to the more detailed type of code values i (repr a). The straightforward attempt

```
genlet0 e = \k → runCPS $ let_ e (\z → k (var z))
```

fails to type check, because z ostensibly leaks out from the scope of let_. Indeed, if e :: i (repr a), the type of z must be (i ∘ j') (repr a) where j' represents the new binding entered by let_. To prevent the scope extrusion (see §4), j' is universally quantified within let_. Since z is passed to the continuation k, the type of z is part of the type of genlet0, and hence the quantified variable j' escapes.

The real reason for the type-checking failure of genlet0 may be understood as follows. Consider again the typical example:

```
li  e = lam (\x → var x +: genlet e)
tli   = lam (\y → resetJ $ li (var y +: int 1))
```

The function li is invoked with the open expression of the type i (repr a) where i represents the type environment with the binding for y. The expression genlet e then has the type (i ◦ j) (repr a) in the extended type environment that also includes the binding for x. The continuation k, see genlet0 code, should therefore receive the argument of that type. On the other hand, if e:: i (repr a) in let_ e (\z → k (var z)) then the continuation k receives the argument of the type (i ◦ j') (repr a) where j' stands for the type environment of the variable z. Since z and x are not related, j and j' are distinct and cannot be reconciled. What makes let-insertion so complex is that the type environment is extended in such a non-linear way.

The solution is quite involved, see the file `TSCPST.hs`. It introduces a new CPS hierarchy, called CPSA, with the applicative CPS transformer of a rank-3 type. The type is very complex; a good way to understand it is as a specialization of the following

```
newtype CPSA2 w m a =
    CPSA2 (∀ m1. Extends m m1 ⇒
              (∀ m2. Extends m1 m2 ⇒ m2 a → m2 w) → m1 w)
```

If we set m1 and m2 to be the same as m, we obtain the standard CPS applicative

```
newtype CPSA0 w m a = CPSA0 ((m a → m w) → m w)
```

which is only useful for let-insertion outside any lambda-expressions. To place the inserted let-statement within the body of a generated function, we have to account for the fact that the binding environment at that point is richer than the base m (e.g., the Identity) applicative. Therefore, we should define

```
newtype CPSA1 w m a =
    CPSA1(∀ m1. Extends m m1 ⇒ (m1 a → m1 w) → m1 w)
```

This applicative assumes that the captured continuation m1 a → m1 w has the same binding environment, represented by m1, as that of the inserted **let**. To insert let across the binders, we inevitably come to the rank-3 CPSA2.

To make CPSA2 an instance of Applicative we need the transitivity of Extends, which is very difficult to express in Haskell. Therefore, the real CPSA type is the specialization of CPSA2:

```
newtype CPSA w m a =
    CPSA{unCPSA ::
            ∀ hw. AppLiftable hw ⇒
             (∀ h. AppLiftable h ⇒
              ((m ◦ hw) ◦ h) a → ((m ◦ hw) ◦ h) w)
            → (m ◦ hw) w}
```

Recall, from Fig. 2 that AppLiftable is an Applicative with the additional law that lets us change the order in the composition of applicatives, hence supporting the familiar exchange rule for the components of type environments. AppLiftable applicatives are also closed under composition. One may view AppLiftable as both an Applicative and a so-called Distributive functor[9].

Since the parameter m can be instantiated to be CPSA w' m' again, CPSA generates the hierarchy. Unlike that of Danvy-Filinski [18], ours is applicative but not monadic. Before looking at the code, the reader is encouraged to write an applicative instance for `CPSA w m` as an exercise. Only by trying this exercise one can truly understand this complex type.

4. Safety properties

This section details static assurances of our generators and gives informal justification. Formal justification is quite involved: the related [31] gives a taste. It is the subject of another paper.

Our library relies on the tagless-final [9] representation of the target language (which is a simply-typed subset of Haskell, presently). Since the encoding is tight, we have

Proposition 1. *Every value of the type*

$$\forall \textit{repr. (SSym repr, LamPure repr, ...)} \Rightarrow \textit{repr a}$$

in an environment x_i : repr $a_i, \ldots$ *denotes a well-typed target term of the type* a *in a target-language type environment* $x_i : a_i, \ldots$.

It immediately follows, by the type soundness of Haskell, that every code value produced by a well-typed Haskell program denotes a well-typed target term. Our library statically ensures well-typedness even for parts of the generated code, not only for the entire generated program.

Our effectful generators explicitly carry the target-language type environment: by design, an effectful generator of the type (m ∘ i) (repr a) (omitting the constraints on the type variables per our convention), if it successfully terminates, produces potentially open target code, whose free variables are in the type environment represented by the applicative i. The 'run' functions such as runC in Figure 1 set i to be the Identity, corresponding to the null environment. It follows that

Proposition 2. *The code value produced by the functions* runR *and* runC *represents closed target code.*

Thus our Haskell generators produce well-typed code without unbound variables. The property is relatively weak: if e is a faulty generator that attempts to produce code with unbound variables, the type error will be emitted only upon type-checking the runC e expression. Our library has a stronger property, maintaining well-scopedness at all times and making such e ill-typed. More importantly, our library statically prevents generation of the code with accidentally bound variables. The notion of well-scopedness is subtle; the next section explains.

[9] See the package `distributive` on Hackage.

4.1. Examples of breaking lexical scope

Guaranteeing the generation of well-typed and closed code is not enough however. The generated code may be closed, but its bindings could be 'mixed-up' or 'unexpected'. It is a quite subtle problem to define what it means exactly to generate code with expected bindings; the literature, which we review in this section, relies on negative examples, of intuitively wrong binding or violations of lexical scope.

As the first example of intuitively wrong behavior we use the one from [32, Section 3.3]. The example, unfortunately admitted in the system of [32], exhibits the problem that bindings "vanish or occur 'unexpectedly' ". The example can be translated to our library, but only if we break it:

```
exCX f = unsafeLam(\y → unsafeLam (\x → f (var x)))
```

here unsafeLam is the *unsafe* version of the lam generator for building target code functions – without the higher-rank type (without the ∀j). We introduce unsafeLam for the sake of this problematic example, because otherwise, happily, it will not type check. We may apply exCX to different functions, obtaining the code shown in the comments beneath the generator:

```
exCX_c1 = exCX id
−− \x_0 → \x_1 → x_1

permute_env :: (m ∘ ((i ∘ j1) ∘ j2)) a → (m ∘ ((i ∘ j2) ∘ j1)) a
exCX_c2 = exCX permute_env
−− \x_0 → \x_1 → x_0
```

The binding structure of the generated code depends on the argument passed to exCX. Thus scope is not lexical in the sense that the mapping between binding and reference occurrences of variables cannot be determined just by looking at the code for exCX or its type. Speaking of the type, here is the inferred type of exCX (omitting the constraints per our convention and abbreviating (→) (repr a) to j1 and (→) (repr b) to j2):

```
exCX :: ((m ∘ ((i ∘ j1) ∘ j2)) (repr b) → (m ∘ ((i ∘ j1) ∘ j2)) (repr c))
      → (m ∘ i) (repr (a → b → c))
```

The type says that the argument of exCX maps the target code valid in the environment with at least two slots, j1 and j2, into the target code in the same environment – or in the environment of the same structure. If the type variables a and b happen to be instantiated to the same type (Int in exCX_c1 and exCX_c2), j1 and j2 become the same and so exchanging these slots in the type environment preserves its structure. That is why exCX_c2 above was accepted. If the type environment is just a sequence and variables are identified by the offsets in the sequence, swapping two elements in the environment preserves the property that each free variable in a term corresponds to a slot in the environment. Alas, swapping changes the mapping between the variable references and the slots. If the type system of the staged language enforces merely the well-formedness property that each free variable in the target code should correspond to *some* slot in the (explicit) target environment, we lose lexical scoping for the generated code. We cannot statically tell the correspondence between binding and reference occurrences of target variables. We thus give further, clearer evidence for the argument of Pouillard and Pottier [33] that well-scoped De Bruijn indices do not per se ensure that the variable names are handled "in a sound way." (The system of Chen and Xi [32] used raw De Bruijn indices for variables; therefore, they could demonstrate the problem by choosing f to be either the identity or the De Bruijn shifting function. In our system, a variable reference is a projection from the environment rather than an abstract numeral, which makes the example a bit more complicated.)

We must stress that without unsafeLam, the problematic example does not type in our system! If we use our regular lam, the type checker immediately complains of the escaping quantified variable j1. We may try to give the signature that specifically permits the environment-shuffling f:

```
-- ill-typed!
exCX4 :: (Applicative  m, SSym repr, LamPure repr, AppLiftable  i ) ⇒
      (∀ j1  j2 . (Applicative  j1 ,  Applicative  j2 ) ⇒
       (m ∘  ((i  ∘  j1 ) ∘  j2)) (repr  b) →
       (m ∘  ((i  ∘  j2 ) ∘  j1)) (repr  c))
      → (m ∘  i ) (repr  (a → b → c))
exCX4 f = lam(\y → lam (\x → f (var  x)))
```

It is rejected by the type checker because of the attempt to identify the j1 associated with y and the j2 associated with x. These two are independently quantified and not unifiable. Thus, our library enforces the abstraction of the type environment and makes the mapping between bound and reference occurrences of the variables statically apparent. Since we identify future-stage variables with quantified type variables j, the scope of future-stage variables is the quantification scope of the corresponding j type variables, which is evident from the type. *Present-stage code types tell future-stage variable scopes.*

Let us take another example of an effectful code generator, from Kim et al. [34, §6.4]. Written with our library, it is as follows (see `Unsafe.hs` for the complete code for these examples.)

```
exKYC1 :: (IO ∘  i ) (repr  (Int  → Int  → Int ))
exKYC1 = do
  a ← int  1 ≫= newIORef
  f ← unsafeLam (\x → unsafeLam (\y →
        (weaken (var x)  +: var  y) ≫= writeIORef a ≫ int  2))
  g ← unsafeLam (\y → unsafeLam (\z → readIORef a))
  return  g
-- \x_0 → \x_1 → (+) x_0 x_1
```

The generator stores the open code `(weaken (var x) +:  var y)` in an outside reference cell a and inserts the code under the scope of two different abstractions, in g. Kim et al. argue that a (Lisp) programmer might have expected that only the variable y is captured by the new abstraction in g; if the programmer used the system of Chen and Xi [32], then both variables would be captured (producing the code shown on the comment line). We view this example as a blatant violation of lexical scope: leaking bound variables from under their binders, and especially capturing them by different binders, violating hygiene, is an offense. We can only write exKYC1 if we deliberately break our library; inserting even one regular, safe lam provokes the ire of the type checker.

We stress again that the two problematic examples will not type with the unbroken lam. It is the higher-rank type of lam that is responsible for the rejection of the generators attempting to produce ill-scoped code. Since the let-insertion and loop-interchange code is written in terms of lam and the similar let_, their safety follows from the safety properties of the latter.

We may liken the higher-rank type of lam to that of runST in the ST monad [35]. In fact, our ill-scoped examples are akin to those [35, §2.5.2] of mutable cells created in one state thread escaping or being used in another thread. Launchbury and Peyton Jones argue how parametricity (which comes from the universal quantification over the state of the thread – target code type environment in our case) prevents bad examples. The analogy with runST helps us make precise the notion of well-scopedness. Recall that (m ∘ i) (repr a) is a generator of a potentially open code whose free variables are described by the type environment denoted by i. The implementation of lam realizes a particular mapping, 'coding function' in the words of Launchbury and Peyton Jones [35], assigning target-code free variables particular slots in i. The generated code is well-scoped if using a different coding function will generate α-equivalent code. The paper [35] outlines an argument based on logical relations to prove the coding-function independence for their runST threads. We expect that a similar argument can apply to our case, but its formal treatment is left for another paper.

5. Related work

5.1. Template Haskell

For generating Haskell code, our library relies on Haskell AST as represented by Exp data type of Template Haskell (TH). We also rely on TH's pretty-printing.

Template Haskell also lets us build code in a much more convenient way, compared to Exp: using quasiquotation in the spirit of Lisp and MetaML. Template Haskell permits effects, including IO, within unquotes, via the so-called Q monad. Template Haskell does limited and idiosyncratic type checking of under quasiquotation – and still permits construction of ill-typed code or code with unbound variables. Like low-level Lisp macros, TH is unhygienic. Therefore, the completely generated code must be type-checked, at which time code generation errors become apparent. Alas, the error messages are quite unhelpful, referring to the generated code, which is often large and hardly comprehensible. Normally, a generator is made of many components written by separate people. Effects can be non-local. When effectful generators produced wrong code (inserted a binding to a wrong place), it could be quite difficult to figure out who did it. We aim at the generated code to be well-formed and well-typed by construction; attempts to generate bad code should be reported when the generator itself is type-checked.

We must stress that the post-validation approach employed by TH – type check the generated code before use – does *not* catch all violations of the lexical scope. Lexical scope mix-up (accidental capture) may well generate well-formed code – but with unanticipated bindings. The generated code will compile, but run in an unexpected way, which is *very* difficult to debug. Thus the TH approach is not acceptable to us.

As of GHC version 7.8, TH has introduced typed quoted expressions TExp, which are quite like MetaOCaml brackets, only restricted to two levels, with no run and no polymorphic lift (although that may be a feature). TExp are type checked as they are constructed, reporting the errors in terms of the generator. TExp thus provides the same static assurances as MetaOCaml. Alas, TExp permits no effects whatsoever during code generation. Therefore, most of the examples in the present paper – in particular, let-insertion across the binders §2.5 and loop interchange §2.6 – are not possible with TH's typed quoted expressions.

The code-generation library described in the present paper has the same static guarantees as TExp; in addition, it permits all effects of the built-in Q monad of TH, including arbitrary IO, plus all other monadic effects. Our library generates Template Haskell expressions Exp without this monad. The Q monad therefore does not have to be built-in.

5.2. Why the problem is so difficult

Let us review what makes generating assuredly well-typed and well-scoped code so difficult in the presence of effects. The source of all problems is generating abstractions in the target code, which is inherently a two-step process. First, a free variable is created and the body of the abstraction is produced. Second, the binding form such as let or lambda, to bind the variable in the generated body, is built. Scope extrusion occurs when the created fresh variable is not bound in the second step, by its intended binder. The variable is either left unbound or, insidiously, is accidentally captured by some other binder. Scope extrusion thus is an ever-present danger when generating higher-order code.

Scope extrusion may occur even in a pure-functional generator, if the meta-programming system has a first-class operation to run or print the generated code (which is typically the case). In our library, such a dangerous code can be written as

```
lam (\x → runR x)
```

In our library, this code does not type-check; the error message essentially describes the attempt to run an open code. In many other meta-programming systems such code does type-check; evaluating it causes some sort of a run-time error (see [32, §1] for examples). "This subtle problem, which seems rather technical, can lead to serious difficulties in establishing type soundness for typed meta- programming." [32]. This is the main reason (related to the general difficulty of dealing with open terms in higher-order abstract syntax) that Chen and Xi had to abandon the convenient, 'clean and elegant' HOAS.

Scope extrusion when attempting to run an open code (code that we have not yet finished constructing) is the problem that stimulated the development of environment classifiers [23]. A classifier, annotating code types, stands for an open set of free variables of the same stage. Alas, this elegant solution does not extend to generators with side-effects: environment classifiers are too coarse-grained to prevent scope extrusion while still allow effectful manipulation of open code. The following example, a simplification of assert-insertion in §2.3, demonstrates the need to *precisely* annotate code types with the target variables that may be free in that code.

```
writeGood = lam (\x →
  J $ do
    r ← newIORef (int 0)
    c ← unJ $ lam (\y → J $ do
                        writeIORef r $ var x +: int 2
                        return $ int 1)
    z ← readIORef r
    unJ $ (var c $$ z)
  )
```

While generating the body for the inner lam, we store open code var x +: int 2 in the reference cell r. The code saved in r is later applied to the generated inner lam. Such a manipulation of open code is desired, and should be expressible, and it is in our library: writeGood type checks. However, if we by mistake store var y +: int 2 in r, the generator should be rejected because the scope extrusion, of the variable y, will result otherwise. To reject that bad code and still accept writeGood, the type checker needs to know which target variables are free in var x +: int 2 and var y +: int 2. The types of the two code expressions must differ. Alas, the environment classifier frameworks assigns the two code values the same classifier.

It is inevitable that preventing scope extrusion in the presence of effects while still permitting manipulations of open code requires as precise as possible annotations of code types with free target variables. We face then three problems pointed out in [23, §1.4]: the large size of annotated code types, α-conversion and environment polymorphism. We have to brace for the size of the code type being linear in the number of free variables. Maintaining α-equivalence for (eventually) bound target variables requires avoiding concrete variable names in type annotations. The need for environment polymorphism is better illustrated on three progressively more complex examples, extended from the one in [23, §1.4].

```
eta  f = lam (\x → f (var x))

teta0 = eta (\z → int 1 +: z)
teta1 = lam (\y → eta (\z → z +: vr y))
teta2 = lam (\y → lam (\w → eta (\z → z +: vr y *: vr w)))
```

The generator eta is the two-level η-expansion often used in partial evaluation community; teta0, teta1, and teta2 are three examples of its use. In teta0, the argument of eta is a function that increments the received code value; the result has the same number of free variables. In teta1, the argument of f returns the code value with an extra free variable, y; in teta2, f extends the environment of the received code value with two free variables. The type of eta must say that f is a function that receives open code (variable x) and returns code with any number of free variables. We now have to make sure all these new free variables are distinct from x, the one created by eta itself. We review the resulting problems later below. The following two extensions of eta demonstrate further complications:

```
eta2 f = lam (\x → lam (\y → f (weaken (var x)) (var y)))

etan f 0 = lam (\x → f (var x))
etan f n = lam (\x → etan (\z → f (z +: vr x)) (n−1) $$ (var x))
```

The type of f in etan should not only specify that it produces a code value with arbitrarily many free variables. The argument of f is also a code value with an arbitrary, furthermore, statically unknown, number of free variables. All these free variables should somehow be kept distinct from the ones f itself adds.

The code type i (repr t) in our library solves the problem. The type environment of the open code is abstractly represented by an applicative i; the extended type environment, accounting for a new free variable represented by j, is the composition i ∘ j, which is itself an applicative. We hence obtain a way to concisely represent an arbitrarily many free variables. Our library eliminates the problems of scope extrusion and makes it possible again to enjoy clean and elegant higher-order abstract syntax. Below we briefly describe other approaches for dealing with the problems identified in this section.

5.3. Code generation with effects

The present paper is the last in the line of research on effectful program generation. The most notable in this line is [36, 37], who developed an off-line partial evaluator for programs with mutation. Partial evaluator can perform some of the source code mutations at specialization time, if possible. Such operations may involve code, including open code. Scope extrusion is prevented by careful programming of the partial evaluator (followed by a proof). The partial evaluator is not extensible and is not maintained; if new specializations are desired, a user has little choice but to thoroughly learn the implementation, extend it, and redo the correctness proof.

Staged languages attempt to ease the burden, giving the user code-generating facilities without requiring the user to become a compiler writer. The latter requirement implies that the generated code should be well-formed and well-typed and free from unbound variables, so the end user should not need to examine it. Since the unrestricted use of effects quickly leads to the generation of code with unbound variables, it has been a persistent problem to find the right balance between the restrictions on effects and expressiveness. So far, that balance has been tilted away from expressiveness. We can judge the expressiveness by several benchmarks: (1) Faulty power §2.1: throwing simple exceptions in code generators; (2) Gibonacci §2.4: generating an arbitrary, statically unknown number of bindings; (3) assert-insertion beyond the closest binder, §2.3; (4) let-insertion beyond the binder, §2.5. Only the present work implements all four benchmarks; even assert-insertion was not reachable before with statically assured generators.

We now review in more detail the limitations on expressivity that have been previously imposed for the sake of static assurances. The work [38] presents a type-and-effect system for meta-programming with exceptions, allowing exception propagation beyond target-code binders. Exceptions are treated as atomic constants, and cannot include open code. The system permits Faulty power but not the other benchmark in our suite. [39, 40] permitted mutations but only of the closed code; the approach cannot therefore implement the assertion-insertion benchmark.

Mint [21] is a staged imperative language, hence permitting generators with effects such as mutation and exceptions. Mint does support the Faulty power. Mint severely restricts the code values that may be stored in mutable variables or thrown in exceptions, by imposing so-called weak separability. Even closed code values cannot be stored in reference cells allocated outside a binder. Therefore, Mint cannot implement the assertion-insertion benchmark .

Swadi et al. [28] and Kameyama et al. [22] described the systems that permit the use of control effects, and hence mutation, restricting them within a binder: the generator of a binder is always pure. The first system used continuation-passing (or, monadic) style, whereas the latter was in direct style. Both systems implement Gibonacci; neither implements faulty power, although the system [22] can be trivially extended for that case (imposing the same restrictions on values thrown from under the binder, as those of Mint). The two systems hit the local optimum, allowing writing moderately complex generators, e.g., [4].

The parallel line of work [34] (and, in the same spirit, [41]) attempts to formalize and make safe the practice of generating code with concrete symbolic names, typical of Lisp. The variable capture is specifically allowed and the lexical scope of the generated code is not assured statically.

Shifting focus away from well-scopedness and concentrating on expressivity and improving the end-user productivity is characteristic of the other line of research. Rompf and Odersky proposed a lightweight approach to staging in Scala [11], which provides an effective way to generate high-performance code. Their goals are quite different from ours: they focus on practical issues on code generation, in particular, how to make the designing, efficiently implementing and using practical DSLs convenient for the end user. We, on the other hand, focus on defining and statically assuring well-scopedness.

The language Terra [42] is a multistage language based on Lua. Although untyped, it assures the absence of unbound variables in the generated code syntactically, by representing any open generated code as a metalanguage function. This approach does not however prevent generation of code with unexpectedly bound variables.

5.4. Contextual systems

In our approach, code generators may produce open target code and have the type that includes the target code typing environment. Moreover, the type contains the ‘names’, represented by applicatives, for free target variables. The latter fact in particular relates our work to the contextual modal type theory [43]. Unlike the latter work, our ‘unquotation’ (which is implicit in the use of code combinators) is much more concise; we also support some polymorphism over environments, and thus, modularity. We also never destruct or pattern-match on code values (see the next section for more discussion). Recording the 'names' of variables in the type of a term also relates our system with record systems with first-class labels [44]. Unlike them, we do not need negative, freshness constraints because our labels j are always chosen fresh by the type checker.

Environment classifiers [23] are an elegant simplification of contextual modal type theories, which indexes open code and contexts by classifiers that stand for extensible sets of free variables (rather than variables themselves). Alas, the classifiers as originally proposed are not precise enough to statically assure well-scoped generated code in the presence of generator effects. The present paper may be viewed as the system of environment classifiers with improved precision.

Rhiger [27] proposed a multi-stage calculus which allows effects in generators with the static guarantee of type safety. Although simple and elegant, his calculus lacks polymorphism over environments, which makes it impossible to implement the examples like those in §2.4 and §5.2.

5.5. Programming with names

The nominal tradition has been extensively reviewed in [33]. Using the latter's criteria, our approach can be classified as using explicit contexts, with 'names' inhabiting every type (the consequence of HOAS), and no costly primitives. The type system ensures not only that a closed generator generates closed code, but also that the code generator preserves the lexical scope.

Our approach has many similarities with that of [33], in particular, their De Bruijn-index implementation. Our environment i is quite like World. The main difference, which explains the others, is the difference in intended applications. We are interested in domain-specific languages for code generation. The programmer building generators from given blocks is not necessarily an expert in the target language; therefore, keeping the generated code abstract and non-inspectable is the advantage. It also enables a richer equational theory (see below). One of the main intended applications for the nominal systems is writing theorem provers, code verifiers, etc. The ability to inspect, traverse and transform terms, which may contain bindings, is a must then.

The framework of [33] provides for generating fresh names, comparing them, and moving them across the worlds. We permit none of that. Our approach is purely generative: the generated code is a black box and cannot be inspected. Comparing variables names for equality or computing the set of free variable of a code value are in principle unimplementable in our approach. The main benefit of the pure generative restriction is simplicity. The framework of [33] required the power of dependent types to ensure *some* of the soundness of dealing with names and so was implemented in Agda. The remaining invariants were not expressed and had to be ensured by an off-line proof of the implementation. Pure FreshML [45], an experimental language, attained the soundness of name manipulation by introducing a specialized logic and expressing logical assertions in types, extending the type checking. One can say the same about Delphin and Beluga [46, 47]. In contrast, we implement our code-generation library in ordinary Haskell. Experience showed that pure generative approach, albeit seemingly restrictive, does not prevent generation of highly optimal code [48, 49].

The main benefit of generative approach is a richer equational theory: as argued in [50, 51] allowing inspection of the generated code makes the equational theory trivial. Indeed, if one could compute the set of free variables of a term, one could distinguish two β-equivalent terms, lam (\x → int 1) $$ var y and int 1. (Our library has a function to show the code, but its type is not polymorphic in repr.)

The system of [33] and the other nominal systems reviewed therein do not specify how and if they permit let-insertion across binders while ensuring lexical scope.

We should specifically contrast our approach with well-scoped De Bruijn indices [32]. Although the approach ensures that all variables in the generated code are bound, the binding may be unanticipated, see §4.1. The problem was indicated in the review of [33], although it has been pointed out by [34] and already in [32]. Although our representation of target environment by nested ∘ is reminiscent of the well-scoped De Bruijn-index approach, our use of rank-2 types for future-stage binders prevents unintended permutations of the environment or forgetting to add weaken and ensure that 'variable references', represented as projections from the environment, always match their environment slot.

Interestingly, Chen and co-authors [32] gave up on HOAS (which was used by the authors in [7]) because "In general, it seems rather difficult, if not impossible, to manipulate open code in a satisfactory manner when higher-order code representation is chosen." Second, HOAS representation makes it possible to write code that does "free variable evaluation, a.k.a. open code extrusion". The authors use De Bruijn indices, however cumbersome they are for practical programming (which the authors admit and try to sugar out). The sugaring still presents the problems (reviewed in §4.1). We demonstrate how to solve both problems, manipulation of open code and prevention of free variable elimination, without giving up conveniences of HOAS.

5.6. Hygienic macros

The long tradition of code generation, or macros, in Lisp systems has long pointed out the danger of variable capture and the need to maintain the *hygiene* of macro-expansion [52]. Alas, defining what it means precisely has been elusive [53, 54]. The latter papers argue that a type system for macros is necessary to define and maintain lexical scope. The macro system of Herman and Wand [53] is, like ours, purely generative.

6. Conclusions

We have presented the so far most expressive, yet statically safe code generation approach. It permits arbitrary effects during code generation, including those that store or move open code. For the first time we demonstrate let-insertion across an arbitrary number of generated binders and loop interchange while statically assuring that the generated code is well-typed and contains no unbound variables or unexpectedly bound ones. A generator or even a generator fragment that would violate these assurances is rejected by the type checker.

We have fulfilled the dream of Taha and Nielsen [23]: "that the notion of classifiers will provide a natural mechanism to allow us to safely store and communicate open values, which to date has not been possible." Our approach is to make classifiers more precise, associating them with each binding rather than a set of bindings. Classifiers, or quantified type variables, act as names for free variables; the quantification scopes of these type variables correspond to the binding scopes of the respective generated variables. In other words, *the generator type tells the scope of generated variables.*

Although our approach makes the 'names' of free variables apparent in the types of open code, it avoids the common drawback of context calculi: the need to state freshness-of-names constraints. They are implicit and enforced by the type checker. Although our approach exposes target-code binding environments in the types of the generator, it permits environment polymorphism and statically prevents weakening too little or too much. Our approach further departs from statically scoped De Bruijn indices by permitting human-readable names for the variables. In fact, our approach vindicates HOAS, which has been regarded as unsuitable for assured and expressive code generation.

We have implemented the approach as a Haskell library. It may be regarded as a blueprint for a safe subset of Template Haskell. The approach can be implemented in any other language with first-class polymorphism, such as OCaml. Our use of mature languages, our guarantee that the generated code compiles, the human-readable variable names afforded by HOAS, and the generator modularity enabled by environment polymorphism together let domain experts today implement efficient domain-specific languages.

As for theory, we introduced a novel, applicative CPS hierarchy that does not treat abstraction as a value, permitting effects to extend past a binder. That result has many implications, for example, for the analysis of quantifier scope in linguistics.

Acknowledgements. We are very grateful to Simon Peyton Jones, Nicolas Pouillard, Nathaniel W. Filardo and Jun Inoue for many helpful comments. We appreciate helpful discussions with Atsushi Igarashi, and Didier Rémy. We thank anonymous reviewers for very many suggestions. The support of JSPS, under the Kakenhi grant 25280020, is gratefully acknowledged.

[1] G. Keller, H. Chaffey-Millar, M. M. T. Chakravarty, D. Stewart, C. Barner-Kowollik, Specialising Simulator Generators for High-Performance Monte-Carlo Methods, in: PADL, LNCS, 2008.

[2] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, SPIRAL: Code Generation for DSP Transforms, Proceedings of the IEEE 93 (2) (2005) 232–275.

[3] A. Begel, S. McCanne, S. L. Graham, BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture, SIGCOMM Computer Communication Review 29 (4) (1999) 123–134.

[4] J. Carette, O. Kiselyov, Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code, Science of Computer Programming 76 (5) (2011) 349–375, doi:\bibinfo{doi}{10.1016/j.scico.2008.09.008}.

[5] A. Cohen, S. Donadio, M. J. Garzarán, C. A. Herrmann, O. Kiselyov, D. A. Padua, In Search of a Program Generator to Implement Generic Transformations for High-Performance Computing, Science of Computer Programming 62 (1) (2006) 25–46.

[6] P. Thiemann, Combinators for Program Generation, Journal of Functional Programming 9 (5) (1999) 483–525.

[7] H. Xi, C. Chen, G. Chen, Guarded Recursive Datatype Constructors, in: [55], 224–235, 2003.

[8] T. Rompf, N. Amin, A. Moors, P. Haller, M. Odersky, Scala-Virtualized: linguistic reuse for deep embeddings, Higher-Order and Symbolic Computation (Sep.), doi:\bibinfo{doi}{10.1007/s10990-013-9096-9}.

[9] J. Carette, O. Kiselyov, C.-c. Shan, Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages, Journal of Functional Programming 19 (5) (2009) 509–543.

[10] C. McBride, R. Paterson, Applicative Programming with Effects, Journal of Functional Programming 18 (1) (2008) 1–13.

[11] T. Rompf, M. Odersky, Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs, Commun. ACM 55 (6) (2012) 121–130, doi:\bibinfo{doi}{10.1145/2184319.2184345}.

[12] J. Yallop, L. White, Lightweight Higher-Kinded Polymorphism, in: FLOPS, no. 8475 in LNCS, Springer, 119–135, doi:\bibinfo{doi}{10.1007/978-3-319-07151-0}, 2014.

[13] O. Kiselyov, Typed Tagless Final Interpreters, in: Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming, SSGIP'10, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-32201-3, 130–174, doi:\bibinfo{doi}{10.1007/978-3-642-32202-0_3}, 2012.

[14] C. Lengauer, W. Taha (Eds.), MetaOCaml Workshop 2004, vol. 62(1) of *Science of Computer Programming*, 2006.

[15] G. Huet, B. Lang, Proving and Applying Program Transformations Expressed with Second-Order Patterns, Acta Informatica 11 (1978) 31–55.

[16] D. Miller, G. Nadathur, A Logic Programming Approach to Manipulating Formulas and Programs, in: S. Haridi (Ed.), IEEE Symposium on Logic Programming, IEEE Computer Society Press, Washington, DC, ISBN 0-8186-0799-8, 379–388, 1987.

[17] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic 5 (2) (1940) 56–68.

[18] O. Danvy, A. Filinski, Abstracting Control, in: Lisp & Functional Programming, 151–160, 1990.

[19] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. N. Swadi, W. Taha, Implicitly Heterogeneous Multi-Stage Programming, New Generation Computing 25 (3) (2007) 305–336, doi:\bibinfo{doi}{10.1007/s00354-007-0020-x}.

[20] A. P. Ershov, On the Partial Computation Principle, IPL: Information Processing Letters 6.

[21] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, W. Taha, Mint: Java Multi-stage Programming Using Weak Separability, in: PLDI '10, ACM Press, New York, 2010.

[22] Y. Kameyama, O. Kiselyov, C.-c. Shan, Shifting the Stage: Staging with Delimited Control, in: PEPM, ACM Press, New York, ISBN 978-1-60558-327-3, 111–120, 2009.

[23] W. Taha, M. F. Nielsen, Environment Classifiers, in: [55], 26–37, 2003.

[24] J. Hage, DOMain Specific Type Error Diagnosis (DOMSTED), Tech. Rep. UU-CS-2014-019, Department of Information and Computing Sciences, Utrecht University, 2014.

[25] B. Heeren, J. Hage, S. D. Swierstra, Scripting the type inference process, in: ICFP, 3–13, doi:\bibinfo{doi}{10.1145/944705.944707}, 2003.

[26] B. Heeren, J. Hage, Type Class Directives, in: PADL, vol. 3350 of *LNCS*, 253–267, doi:\bibinfo{doi}{10.1007/978-3-540-30557-6_19}, 2005.

[27] M. Rhiger, Staged Computation with Staged Lexical Scope, in: ESOP, 559–578, 2012.

[28] K. Swadi, W. Taha, O. Kiselyov, E. Pašalić, A Monadic Approach for Avoiding Code Duplication When Staging Memoized Functions, in: PEPM, ISBN 1-59593-196-1, 160–169, 2006.

[29] A. Bondorf, Improving Binding Times Without Explicit CPS-Conversion, in: Lisp & Functional Programming, 1–10, 1992.

[30] J. L. Lawall, O. Danvy, Continuation-Based Partial Evaluation, in: Lisp & Functional Programming, 227–238, 1994.

[31] M. Fluet, J. G. Morrisett, Monadic Regions, Journal of Functional Programming 16 (4–5) (2006) 485–545.

[32] C. Chen, H. Xi, Meta-Programming Through Typeful Code Representation, Journal of Functional Programming 15 (6) (2005) 797–835.

[33] N. Pouillard, F. Pottier, A Fresh Look at Programming with Names and Binders, in: ICFP, ACM Press, New York, 217–228, 2010.

[34] I.-S. Kim, K. Yi, C. Calcagno, A Polymorphic Modal Type System for Lisp-Like Multi-staged Languages, in: POPL, ISBN 1-59593-027-2, 257–268, 2006.

[35] J. Launchbury, S. L. Peyton Jones, State in Haskell, Lisp and Symbolic Computation 8 (4) (1995) 293–341.

[36] P. Thiemann, D. Dussart, Partial Evaluation for Higher-Order Languages with State, 1999.

[37] D. Dussart, P. Thiemann, Imperative Functional Specialization, Tech. Rep. WSI-96-28, Universität Tübingen, 1996.

[38] H. Eo, I.-S. Kim, K. Yi, Type and Effect System for Multi-staged Exceptions, in: APLAS, no. 4279 in LNCS, ISBN 3-540-48937-1, 61–78, 2006.

[39] C. Calcagno, E. Moggi, W. Taha, Closed Types as a Simple Approach to Safe Imperative Multi-Stage Programming, in: ICALP, no. 1853 in LNCS, 25–36, 2000.

[40] C. Calcagno, E. Moggi, T. Sheard, Closed Types for a Safe Imperative MetaML, Journal of Functional Programming 13 (3) (2003) 545–571.

[41] G. Mainland, Explicitly heterogeneous metaprogramming with MetaHaskell, in: ICFP, ACM Press, New York, 311–322, doi:\bibinfo{doi}{10.1145/2398856.2364572}, 2012.

[42] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, J. Vitek, Terra: a multi-stage language for high-performance computing, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, ACM, 105–116, doi:\bibinfo{doi}{10.1145/2499370.2462166}, 2013.

[43] A. Nanevski, F. Pfenning, B. Pientka, Contextual Modal Type Theory, Transactions on Computational Logic 9 (3) (2008) 23:1–49.

[44] D. Leijen, First-class labels for extensible rows, Tech. Rep. UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, 2004.

[45] F. Pottier, Static Name Control for FreshML, in: LICS, IEEE Computer Society, 356–365, doi:\bibinfo{doi}{10.1109/LICS.2007.44}, 2007.

[46] A. Poswolsky, C. Schürmann, System Description: Delphin - A Functional Programming Language for Deductive Systems, Electr. Notes Theor. Comput. Sci 228 (2009) 113–120, doi:\bibinfo{doi}{10.1016/j.entcs.2008.12.120}.

[47] B. Pientka, A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions, in: POPL, 371–382, 2008.

[48] O. Kiselyov, K. N. Swadi, W. Taha, A Methodology for Generating Verified Combinatorial Circuits, in: EMSOFT, 249–258, 2004.

[49] O. Kiselyov, W. Taha, Relating FFTW and Split-Radix, in: ICESS, no. 3605 in LNCS, ISBN 3-540-28128-2, 488–493, 2005.

[50] W. Taha, A Sound Reduction Semantics for Untyped CBN Multi-Stage Computation, in: PEPM, 34–43, 2000.

[51] M. Wand, The Theory of Fexprs is Trivial, Lisp and Symbolic Computation 10 (3) (1998) 189–199.

[52] E. Kohlbecker, D. P. Friedman, M. Felleisen, B. Duba, Hygienic Macro Expansion, in: LFP, ACM Press, New York, ISBN 0-89791-200-4, 151–161, 1986.

[53] D. Herman, M. Wand, A Theory of Hygienic Macros, in: ESOP '08: Proc. European Symp. On Programming, 2008.

[54] D. Herman, A Theory of Typed Hygienic Macros, Ph.D. thesis, Northeastern University, Boston, MA, 2010.

[55] POPL, POPL '03: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, 2003.