Combinators for Impure yet Hygienic Code Generation

Yukiyoshi Kameyama
University of Tsukuba
kameyama@acm.org

Oleg Kiselyov
oleg@okmij.org

Chung-chieh Shan
Indiana University

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 exchange 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 the mature language.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; polymorphism; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords Multi-stage programming; mutable state and control effects; binders; CPS; higher-order abstract syntax; lexical scope

1. Introduction

High-performance computing applications (scientific simulation [15], digital signal processing [31], network routing [1], and many others) require domain-specific optimizations that the typical domain expert performs by hand over and over again to write each specialized program. The manual optimizations are not only excru-

PEPM '14, January 20–21, 2014, San Diego, CA, USA.

ACM 978-1-4503-2619-3/14/01…$15.00.
http://dx.doi.org/10.1145/2543728.2543740

ccshan@indiana.edu

ciating: their correctness is hard to see, they distort the code beyond recognition making it unmaintainable, they require significant and rare expertise, which cannot be reused for similar cases. It is widely recognized in the high-performance community (see [8] 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 [8].

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) or specializations (such as the preferred data layout, or statically known inputs). The pieces may be regarded as a domain-specific language (DSL). Ideally, the pieces should compose: a user could replace one algorithm with a putatively more optimal 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 compilable without errors, so that the end user 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).

The goals of expressivity, composability and static assurances are in conflict, which so far has remained unresolved. (See §5 for 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 and *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. There are two styles of writing generators: with antiquotations, familiar from Lisp, and code-generation combinators [36, 39]. The former are quite more convenient for the user but require assistance from the compiler and the typechecker. Code-generation 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. Here is an illustrative example, to be discussed in detail in §2.5:

```
loop (int 0) (int (m−1)) (int 1) (lam $ \j →
```

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

This is the main part of a generator to produce the code to multiply an $n \times m$ matrix a by a vector v obtaining v'. The combinators vec_get, mat_get and vec_incr generate code to access or increment matrix or vector elements; int generates the code for a given integer, mulM takes two pieces of generated code and produces the code to multiply them, etc. 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 target code from under binders. The combinator loop generates the code for the loop with the given boundaries, the step and the body. 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.)

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 [31] is based on this idea. Generators, therefore, need effects such as IO, exceptions, non-determinism. We are particularly interested in control effects that are necessary for the follow-up optimization, exchanging 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 exchange 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 exchanged, 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 is part of a program with another binder for jj :: Int. The generated code will successfully compile and will even run, but subtly unexpectedly. A similar optimization, also requiring control effects, is loop-invariant code movement, 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 typed-correct and well-scoped *as it is being constructed.*

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

Performing optimization like loop exchange while statically preventing scope extrusion was identified as an open problem by Cohen et al. [8]. 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 compositional, 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). 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-generation 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.

Our library is not yet ready for real-life applications like those supported by MetaOCaml [21], because its syntax is rather heavy. We write `int 1 +: int 2` to generate the expression `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 rank-2 polymorphism.) Moreover, weakening coercions often 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) [23, 25] rather than De Bruijn indices, so bindings in the generator are human-readable;
3. it allows polymorphism over generated environments, so the same generator module can be reused in many environments;
4. 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.

Finally, we propose a new applicative continuation-passing-style (CPS) hierarchy that allows loop exchange and let-insertion across several generated bindings. These tasks cannot be accomplished in the traditional CPS hierarchy [9].

The structure of the paper The next section introduces the interface of our library and explains it on simple examples of code generation with effects. We then turn to hitherto impossible, with static well-scopedness assurances, examples: in §2.3 we store open code in mutable variables across binders, and in §2.4 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. §2.5 describes a case study of using our library for common loop optimizations: loop exchange 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 and is presented in the following conventions: The implicitly quantified type variable repr in type signatures represents a type that is a member of SSym or another 'symantics' class [6]; we almost always omit the corresponding constraint. We assume that the type variables m, i and j are all constrained to be Applicative. Should confusion arise the reader may refer to Figure 1 and 2, 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-generation combinators and illustrates it on a progression of examples. Figure 1 shows the library interface.

The target code is treated as an EDSL, a domain-specific language embedded in Haskell. The target language is simply-typed: 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 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, is a Haskell expression for 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 the target addition. The types, inferred by the Haskell compiler, make it clear when a Haskell expression represents target code. (We shall elide the constraint SSym 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 :: ∗ → ∗ representing a concrete realization indexed by the expression's type – is called the "tagless final" approach [6]. The approach encourages several concrete realizations of the target language. Our library provides two: 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 an 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

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

Higher-order fragment

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

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

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

weakens :: Extends m n ⇒ m a → n a
vr = weakens ∘ var
```

Extension: let-expressions

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

let_ :: (m ∘ i) (repr a) → (∀ j. AppPermutable j ⇒
                  (i ∘ j) (repr a) → (m ∘ (i ∘ j)) (repr b))
      → (m ∘ i) (repr b)
```

Figure 1. The interface of hygienic code-generation combinators

"(GHC.Num.+)␣1␣2". (We shall elide the module qualification "GHC.Num." when showing the code.) The generated code can be 'spliced-in' using Template Haskell (TH), or saved into a file and compiled separately.

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 [13]. 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 [38] or $\lambda^{\bigcirc}$ [14]) 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 an Error applicative (an Error monad, which is also an applicative) to throw a String exception. (The reason for using applicatives will become clear soon.)

The task is to specialize this code: 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 a generator for "x". Recall that exponentiation is partial: the generator, just as the ordinary powerF, has to throw an exception if the exponent is negative, thus producing no code. The exception can be caught in the rest of the 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 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 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. However, spowerF does not yet answer our problem. For one, it is not clear where m (repr Int) for the second argument of spowerF should come from. Mainly, the specialized exponentiation should be an Int→ Int function. To fulfill the task, we have to generate the code of a function (whose body will be generated by spowerF).

Our library provides a primitive generator for target functions, lamS (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. [39]. The tagless-final approach uses HOAS too [6]; 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 [6]. 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 main contribution of the paper; it is here where our code generating library differs sharply from the state of the art, such as MetaOCaml or Template Haskell. The main problem to overcome is that the 'effect propagation' threatens well-scopedness. Although no ill-scoped code can result 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, 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 not a monad.) The argument of lam is to generate the body of the function, in the applicative m ∘ i' where i' is itself the composition i ∘ j. The applicatives i and j represent type environments of the target code, or, put differently, the effects of generating target-code variables. The composition i ∘ j is in effect the concatenation of the corresponding type environments. The applicatives i and j must be AppPermutable, with an additional law that makes their composition commute, so to support the familiar exchange rule for the

components of type environments. (AppPermutable applicatives are also closed under composition. The Identity and the Reader applicatives are AppPermutable.) 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 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, AppPermutable 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
addt :: (IO ∘ i)(repr Int) → (IO ∘ i)(repr Int) → (IO ∘ i)(repr Int)
addt x y = trace "generating add" (x +: y)
```

The user-defined addt logs all generated additions; the trace is emitted when the code is generated, rather than when it is executed. 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, Figure 1, we deduce the types

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

where j1 is the AppPermutable 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: 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

```
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 2. Library extension: assertion statement and integer division

```
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. It helps to report the error as soon as possible, when we learn the value of the divisor. We have to move the assertion code.

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))
```

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 end of the generation, the resulting transformer is retrieved and applied to the generated code. To add asser-

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 [38] or a trivial ad hoc extension of $\lambda^{\bigcirc}$ [14]. The code-generation library introduced in §2.1 permits however the manipulation of open code in

any applicative. 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 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.

Our running example is of *assertion-insertion*, a special case of *if-insertion*. It has been described in detail in [14], 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 movement under a binder; a simple modification will move beyond the binder.

We start by extending our DSL assertPos and the integral division (/:), see Figure 2. 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 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 crashes.

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

tions therefore, the generator modifies the contents of the cell, composing the current transformer with `assertPos test`. The following code implements the idea, using IO as the generating applicative, and its reference cells IORef.

```
assert_locus :: (m ~ (IO ∘ i)) ⇒
```

```
    (IORef (m (repr a) → m (repr a)) → m (repr a)) → m (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

```
add_assert :: (m' ~ (IO ∘ i)) ⇒
   IORef (m a → m a) → (m a → m a) → m' ()
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) (var y))
```

The generated code demonstrates that assert is inserted before the complex_exp, 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 $\lambda^{\oslash}$[14].

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 :: (m' ~ (IO ∘ i), Extends m m') ⇒
      IORef (m (repr a)) → m (repr a))
      → m' (repr Int) → m (repr Int) → m' (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, m', 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 (var x) (var y)))
```

The generated code shows the assertion y>0 is inserted right after the binding of y, at the earliest possible moment – exactly as desired. Thus assertPos (var y) has really been moved across the binder, lam (\x → ...). If we make a mistake and switch var x and var y as the arguments of guarded_div3, attempting to move assertPos (var x) beyond the binder for x, the type checker reports a problem

```
Could not deduce (j1 ~ j)
Expected type: (∘) i0 j (repr0 Int)
  Actual type: (∘) (i0 ∘ j) j1 (repr0 Int)
In the first argument of `var', namely `x'
In the third argument of `guarded_div3', namely `(var x)'
```

telling us that we have attempted to move x to a smaller binding environment. In other words, the x 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

movement with mutable cells.)

The example is of course simplistic, but easily extensible. For example, by representing the transformer differently, so that the generator, before recording a new assertion, could check if there is already the same or a stronger assertion recorded. The technique thus extends 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 next.

2.4 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 movement. 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 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 [34] and other meta-programming. The paper [14] 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.

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
```

```
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 3. The interface for let-insertion

```
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).

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 since before **let** is inserted, the let-bound variable does not even exist. Seemingly, we cannot generate the **let** body before we generate the binder. The answer to the quandary was found in partial evaluation community long time ago: writing the code or the generator in the continuation-passing style (CPS) [2] (for the detailed explanation, see [5, Section 3.1]) – or else using control operators [20]. Alas, the ordinary CPS [9] cannot be used for let-insertion beyond binders (that is, cannot be used for the invariant code motion) [14]. 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 3. 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, forever preventing let-insertions beyond that point. Some of the let-bound expressions are closed, and could be let-bound outside of 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 genlet 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.

2.5 Loop tiling

This section presents the case study of using our library for common high-performance computing loop optimizations: strip mining, loop exchange and loop tiling. Performing loop exchange with static assurances of well-typedness and well-scopedness of the generated code was posed as an open problem [8]. This section presents the first solution.

Our running example is matrix-vector multiplication[2] (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_incr_ v' i =≪
     mat_get_ a i j ∗ vec_get_ v j))
```

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

This is the standard Haskell implementation of the textbook code for multiplying the matrix a with n rows and m columns by the vector v with the result in the vector v'. 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_incr_ 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 spacial locality, and is one of the basic optimizations in high-performance computing. Tiling is converting each i and j loop into a nested pair of loops, followed by loop exchange, 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 be combining loop tiling with scalar promotion, partial unrolling and other optimizations. Our task thus is to generate ordinary and tiled loop nests modularly and compositionally.

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_incr (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 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', striping 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 a more optimal, 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 §2.4, 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 exchange.

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 exchange loop bodies, moving open code with binders across other binders.

3. Implementation

This section briefly 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 S instances of SSym, LamPure, SymLet, etc. is simple; see the exposition of the tagless-final approach [6] for detailed discussion.

Before showing the implementation of lam, we explain why it has such a complex and strange type. Recall the primitive generator of functions lamS and the example of its use:

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

As evident from the type of lamS, not only Haskell variables represent bound variables in the target code (as exS2 illustrates), but

also the Haskell type environment represents the type environment of the generated code. The former is implicit in Haskell code, and hence is the latter. Therefore, a generator program, which is necessarily a closed Haskell term, is guaranteed to generate closed target code, without unbound variables.

The downside of lamS is that its type does not permit any effects during code generation. To accommodate effects, the type of the abstraction generator should be, at first blush

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

However, this type provides no information so to tell if the moving of open code across a binder results in scope extrusion. Moving open code across binders using mutation or control effects breaks the correspondence between the Haskell type environment and that of the target code. To retain static guarantees of well-scopedness we have to make the target type environment explicit. Suppose Code Γa is such a type of code values with the explicit type environment Γ. We could then give the effectful function generator the following type:

```
lam' :: (Code (Γ;a) a → m (Code (Γ;a) b)) → m (Code Γ(a→ b))
```

where Γ;a extends Γ with the type a. Our lam in Figure 1 is essentially the above lam', with Code Γa realized as i (repr a), where the applicative i represents Γ. Then Code (Γ;a) b is realized as (i ∘ j) (repr b) where the Reader applicative j, being (repr a →), represents type a in the target type environment. Since applicatives compose, (i ∘ j) also has the form of the target type environment. Furthermore, m (Code (Γ;a) b) is m ((i ∘ j) (repr b)), which is (m ∘ (i ∘ j)) (repr b) and (m ∘ (i ∘ j)) is again an applicative if m is. The implementation of our lam is straightforward, keeping in mind that we chose j to be the Reader applicative (repr a→).

```
lam :: (∀ j . AppPermutable j ⇒
        (i ∘ j) (repr a) → (m ∘ (i ∘ j)) (repr b))
       → (m ∘ i) (repr (a→ b))
lam f = fmap lamS (J ∘ fmap unJ ∘ unJ $ f (J ∘ pure $ v))
 where
 v = \repra → repra                          -- bound variable
```

(The reason of the quantification over j will become clear in §4.) It is remarkable that our lam is expressible entirely in terms of the primitive function generator lamS, without looking under its hood. The implementation of let_ is similar. The implementation of var and weaken follows from the Applicative laws.

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

If let-insertion is the only generation effect and if let-insertion is restricted within a binder, then it is easy. The solution, involving delimited continuations, has been known in the partial evaluation community for decades. We can write it in our library as

```
newtype CPS w a = CPS{unCPS :: (a → w) → w}
runCPS :: CPS a a → a
runCPS m = unCPS m id

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

Alas, it is exasperating to generalize this solution from repr a to the more detailed type of code values Code Γa, that is, i (repr a). The straightforward attempt

```
genlet0 e = CPS $ \k → runCPS $ let_ e (\x → k x)
```

fails to type check, because x ostensibly leaks out from the scope

of let_. The real reason for the type-checking failure may be understood as follows. On one hand, if we are to insert **let** across bindings, the environment Γ of genlet0 e should be an extended environment Γ_0 of e, the let-bound expression. Specifically, Γ should be $\Gamma_0;\Gamma'$ where Γ' represents the environment of the bindings crossed by let-insertion. On the other hand, from let_ e (\x → k x) we obtain that x is a bound variable in the target code in the environment $\Gamma_0;b$ where b represents the new binding created by let_. Since the continuation k receives x as an argument, $\Gamma = \Gamma_0;b$. Obviously b and Γ' are not related and don't have to be the same.

The solution is very complex, see the file `TSCPST.hs`. It introduces a new CPS hierarchy, called CPSA, with the following rank-3 type of the applicative CPS transformer:

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

Since the parameter m can be instantiated to be CPSA w' m' again, CPSA generates the hierarchy. Unlike that of Danvy-Filinski [9], 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. A good way to understand CPSA is as a specialization of the following

```
newtype CPSA w m a =
    CPSA{unCPSA ::
           ∀ m1. Extends m m1 ⇒
            (∀ m2. Extends m1 m2 ⇒ m2 a → m2 w) → m1 w}
```

However, to make an instance of Applicative we need the transitivity of Extends, which is very difficult to express in Haskell.

4. Safety properties

This section details static assurances of our generators and gives informal justification. Formal justification is quite involved and is the subject of another paper.

Our library relies on the tagless-final [6] 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 $\forall$repr. (SSym repr, LamPure repr, ...) $\Rightarrow$ repr a in an environment x_i : repr $a, \ldots$ denotes a well-typed target term of the type a in a target-language type environment $x_i : a, \ldots$.

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 $\circ$ i) (repr a) (omitting the constraints on the type variables per our convention), if 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. Mainly, our library statically prevents generation of code with accidentally bound variables. The notion of well-scopedness is subtle; the next section explains.

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 [7, Section 3.3]. The example, unfortunately admitted in the system of [7], 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 problem-

atic 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 ∘ j) ∘ j1)) a -> (m ∘ ((i ∘ j1) ∘ j)) 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 at run time. 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):

```
exCX :: ((m ∘ ((i ∘ (->) (repr a)) ∘ (->) (repr b))) (repr b)
      -> (m ∘ ((i ∘ (->) (repr a)) ∘ (->) (repr b))) (repr c))
     -> (m ∘ i) (repr (a -> b -> c))
```

The type says that the argument of exCX maps target code valid in the environment with at least two slots into target code in the same environment – or in the environment of the same structure. If we instantiate the type variables appropriately, swapping two slots in the 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 [30] 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 [7] 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.)

Let us take another example of an effectful code generator, from Kim et al. [16, §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 [7], 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 generate ill-scoped code. (Incidentally, since the let-insertion and loop-exchange code is written in terms of lam and similar let_, their safety follows.) We may liken the higher-rank type of lam to that of runST in the ST monad [19]. In fact, our ill-scoped examples are akin to those [19, §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 [19], 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 [19] 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 provides much more convenient, compared to Exp, way of building code, 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. So, when effectful generators produced wrong code (inserted a binding to a wrong place), it could be quite difficult to figure out who did it (keeping in mind that identifiers in the generated code are all obfuscated, and the generated code is typically long. 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 mixup (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.

Recently TH has introduced typed quoted expressions TExp, which is 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.

We demonstrate the code-generation library that has the same static guarantees as TExp, and permits all effects of the Q monad, including arbitrary IO (and even control effects). Incidentally, our work points out that the Q monad is redundant. Our library permits storing open expressions in mutable cells and communicating open expressions in exceptions or control effects, across binders – and yet statically ensures well-typedness and well-formedness. To our knowledge, nobody was able to demonstrate that before.

5.2 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 [11, 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 [34], an epitome of code generation with memoization; (3) assert-insertion beyond the closest binder, §2.3; (4) let-insertion beyond the binder, §2.4. Only the present work implements all four benchmarks; even assert-insertion was not reachable before with statically assured generators.

The work [12] presents a type-and-effect system for metaprogramming 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. [3, 4] permitted mutations but only of the closed code; the approach cannot therefore implement the Gibonacci benchmark.

Mint [38] 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 Gibonacci benchmark.

Swadi et al. [34] and Kameyama et al. [14] 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 [14] 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., [5].

The parallel line of work [16] (and, in the same spirit, [22]) attempts to formalize and make safe Lisp practice of generating code with concrete symbolic names. The variable capture is specifically allowed and the lexical scope of the generated code is not assured statically.

Rompf and Odersky proposed a lightweight approach to staging in Scala [33], 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 [10] 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.

5.3 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', or the type-level j proxies, for term's free target variables. The latter fact in particular relates our work to the contextual modal type theory [24]. Unlike the latter work, our 'unquotation' (which is implicit in the use of cogen 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).

Environment classifiers [35] 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 [32] 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 so that a code cannot be re-used in different environments.

5.4 Programming with names

The nominal tradition has been extensively reviewed in [30]. 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 [30], in particular, their De Bruijn-index implementation. Our environment i is quite like World. The main difference, which explains the others, is the different foci of ours and nominal approaches. 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 [30] provides for the generation of fresh names, comparing them, 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, computing the set of free variables of a term are in principle unimplementable in our approach. The main benefit of the pure generative restriction is simplicity. The framework of [30] 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 [29], 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 [26, 28]. In contrast, we imple-

ment our code-generation library in ordinary Haskell. Experience showed that pure generative approach, albeit seemingly restrictive, does not prevent generation of highly optimal code [17, 18].

The system of [30] and the other nominal systems reviewed therein do not specify how and if they permit let-insertion across binders while ensuring lexical scope. Perhaps solving this problem requires additional primitives or environment polymorphism.

We should specifically contrast our approach with well-scoped De Bruijn indices [7]. 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 [30], although it has been pointed out by [16] and already in [7]. Although our representation of target environment by nested $\circ$ 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 gave up on HOAS (which was used by the authors in [39]) 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. [7] need type annotations even for local defi-

nitions. Also, [7] acknowledge that their language is experimental and integrating to the full-fledged language is left for future work.

6. Conclusions

We have presented the so far most expressive 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 exchange 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 [35]: "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 fur-

ther 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 demonstrated an 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.

Acknowledgments

We are very grateful to Simon Peyton Jones, Nicolas Pouillard and Nathaniel W. Filardo for many helpful comments. We appreciate helpful discussions with Atsushi Igarashi, Jun Inoue and Didier Rémy.

References

[1] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Computer Communication Review*, 29(4):123–134, 1999.

[2] A. Bondorf. Improving binding times without explicit CPS-conversion. In *Lisp & Functional Programming*, pages 1–10, 1992.

[3] C. Calcagno, E. Moggi, and W. Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *ICALP*, number 1853 in LNCS, pages 25–36, 2000.

[4] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3):545–571, May 2003.

[5] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, 76(5):349–375, 2011.

[6] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[7] C. Chen and H. Xi. Meta-programming through typeful code representation. *Journal of Functional Programming*, 15(6):797–835, 2005.

[8] A. Cohen, S. Donadio, M. J. Garzarán, C. A. Herrmann, O. Kiselyov, and D. A. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, Sept. 2006.

[9] O. Danvy and A. Filinski. Abstracting control. In *Lisp & Functional Programming*, pages 151–160, 27–29 June 1990.

[10] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and 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*, pages 105–116. ACM, 2013. .

[11] D. Dussart and P. Thiemann. Imperative functional specialization. Technical Report WSI-96-28, Universität Tübingen, July 1996.

[12] H. Eo, I.-S. Kim, and K. Yi. Type and effect system for multi-staged exceptions. In *APLAS*, number 4279 in LNCS, pages 61–78, 2006. ISBN 3-540-48937-1.

[13] A. P. Ershov. On the partial computation principle. *IPL: Information Processing Letters*, 6, 1978.

[14] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shifting the stage: Staging

with delimited control. In *PEPM*, pages 111–120, New York, 2009. ACM Press. ISBN 978-1-60558-327-3.

[15] G. Keller, H. Chaffey-Millar, M. M. T. Chakravarty, D. Stewart, and C. Barner-Kowollik. Specialising simulator generators for high-performance Monte-Carlo methods. In *PADL*, LNCS, 2008.

[16] I.-S. Kim, K. Yi, and C. Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *POPL*, pages 257–268, 2006. ISBN 1-59593-027-2.

[17] O. Kiselyov and W. Taha. Relating FFTW and split-radix. In *ICESS*, number 3605 in LNCS, pages 488–493, 2005. ISBN 3-540-28128-2.

[18] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT*, pages 249–258, 27–29 Sept. 2004.

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

[20] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *Lisp & Functional Programming*, pages 227–238, 1994.

[21] C. Lengauer and W. Taha, editors. *Special Issue on the 1st MetaOCaml Workshop (2004)*, volume 62(1) of *Science of Computer Programming*, Sept. 2006.

[22] G. Mainland. Explicitly heterogeneous metaprogramming with metahaskell. In *ICFP*, pages 311–322, New York, 2012. ACM Press. .

[23] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, Washington, DC, 1987. IEEE Computer Society Press. ISBN 0-8186-0799-8.

[24] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3):23:1–49, June 2008.

[25] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI '88*, volume 23(7) of *ACM SIGPLAN Notices*, pages 199–208, New York, 22–24 June 1988. ACM Press.

[26] B. Pientka. A type-theoretic foundation for programming with higher-

order abstract syntax and first-class substitutions. In *POPL*, pages 371–382, 2008.

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

[28] A. Poswolsky and C. Schürmann. System description: Delphin - A functional programming language for deductive systems. *Electr. Notes Theor. Comput. Sci*, 228:113–120, 2009.

[29] F. Pottier. Static name control for freshML. In *LICS*, pages 356–365. IEEE Computer Society, 2007.

[30] N. Pouillard and F. Pottier. A fresh look at programming with names and binders. In *ICFP*, pages 217–228, New York, 2010. ACM Press.

[31] 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, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[32] M. Rhiger. Staged computation with staged lexical scope. In *ESOP*, pages 559–578, 2012.

[33] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[34] K. Swadi, W. Taha, O. Kiselyov, and E. Pašalić. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, pages 160–169, 2006. ISBN 1-59593-196-1.

[35] W. Taha and M. F. Nielsen. Environment classifiers. In POPL [27], pages 26–37.

[36] P. Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[37] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state. `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz`, 1999.

[38] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *PLDI '10*, New York, 2010. ACM Press.

[39] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In POPL [27], pages 224–235.