Computational Effects across Generated Binders

Maintaining future-stage lexical scope

Yukiyoshi Kameyama
University of Tsukuba
kameyama@acm.org

Oleg Kiselyov
oleg@okmij.org

Chung-chieh Shan
ccshan@post.harvard.edu

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 prevent experts in the application domain, not in programming languages, from 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.

In an imminently practical code-generation framework, we show how to express arbitrary 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, but also express the lexical scopes of generated binders and prevent mixing up variables with different scopes. This precise notion of lexical scope subsumes the complaints about intuitively wrong example generators in the literature. For the first time, we demonstrate statically safe `let`-insertion across an arbitrary number of binders.

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.' The library is convenient to use thanks to the maturity of Haskell, higher-order abstract syntax, and polymorphism over generated type environments.

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

General Terms Design, Languages

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

1. Introduction

High-performance computing applications (scientific simulation [20], digital signal processing [41], network routing [3], and many others) require domain-specific optimizations (for example, reasoning about complex roots of unity) that the typical domain expert performs by hand over and over again to write each specialized program. To recover code reuse without incurring prohibitive dispatching overhead, the leading approach is to automate and modularize the optimizations in the form of domain-specific code generators. By far the most popular representation of generated code is to use a concrete data type, be it S-expressions or even text strings, that allows arbitrary manipulations without respect for the invariant that all variables be bound, let alone the generated code be well typed. Such 'tree hacking' may be appropriate if used by a professional compiler writer for a fixed, verified code generator, but will exacerbate bugs and stymie maintenance if used by an application programmer who may be an expert in biology but not in compilers.

Therefore, we embrace code generation and disavow tree hacking. We seek a refined representation that guarantees by construction that the generated code is well-scoped and well-typed (and hence will compile without error, so the end user need not look at it). These guarantees are not checks that the generator should perform when it ends but invariants that it should maintain as it runs. This way, the programmer discovers mistakes as early as possible, even while writing just one module of the generator, so we improve our confidence that the generator is correct. In short, we aim for principled code generation in which code is an abstract data type that assures static safety and permits equational reasoning.

For a richer equational theory, we follow MetaOCaml in treating each piece of generated code as a black box, which can be combined with other code but not inspected [45]. Experience shows that this *generative* approach permits many domain-specific optimizations [19, 29] (see §6 for comparison with nominal systems). It also makes it possible to improve efficiency by interleaving code generation and evaluation [16].

The more code generators we want to express, the harder it is to assure static safety and permit equational reasoning [25, 50]. In particular, modular code generators necessarily incur side effects such as exceptions, state, and control [32, 49]: to report failures, to order and memoize generated computations [1, 2, 4, 7, 11–13, 27, 43, 44], to search among alternatives [42], and so on. Effects incurred during generation are treated in the literature in four ways:

1. Some systems allow them without regulation. This invalidates hard-won [46] static safety and stymies equational reasoning.
2. Some systems disallow all effects [34] or disallow effects that manipulate open code [5, 6, 51]. This rules out such useful generation techniques as monadic insertion of `let`-statements and `if`-guards [44] and imperative partial evaluation [14, 48].
3. A simple compromise is to encapsulate effects within binders: although an abstraction's body can be generated using effects, generating the abstraction as a whole must be pure. When effects are expressed (conveniently) in direct style, this compromise means all generated binders delimit all effects [19].

   This compromise allows a wide variety of useful generators but rules out many others. The generation techniques ruled out are roughly those that move open code across generated binders (so generating the abstraction as a whole is impure): loop-invariant code motion, scalar promotion [11], early assert-insertion, and normalization by evaluation for sums [2].

4. Some systems explicitly represent environments for the variables used in generated code: either as records with named labels [10, 21] or as tuples [9]. In the latter case, the variables in the generated code are represented as de Bruijn indices. It is straightforward to add effects to these systems while maintaining type soundness, so we follow this approach here.

In sum, our code generators represent environments explicitly so as to express effects on open code across generated binders. Although such representations have been deemed impractical [46, §1.4], our representation overcomes the obstacles as described below.

Contribution 1: static types for lexical scope Unfortunately, previous representations of generated code with explicit environments do not express, let alone enforce, any notion of lexical scope. Lexical scope is crucial to all modular programming, but what it even means in a code generator is hard to pin down [16, 17]. To this end, Pouillard and Pottier [40] put forth three informal slogans:

1. Name abstraction cannot be violated.
2. Names do not escape their scope.
3. Names with different scopes cannot be mixed.

The first slogan could mean that α-equivalent code generators generate α-equivalent code. However, the notion of α-equivalence among generators is itself elusive. It is easier to interpret the second slogan, to mean that all generated code is well-scoped˜– in particular, the final result is a closed program. However, the literature is rife with intuitively wrong examples (see˜§5.1) that show that it is not enough for the generated code to be merely well-scoped.

Our most important contribution is to use static 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. That is the content of the third slogan. We argue 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.

Contribution 2: imminently practical library for code generation To validate our approach, we built a library of combinators for code generation in Haskell. 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 that MetaOCaml has supported [29], 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 using 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) [31, 35] 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-order or higher-order.

The structure of the paper §2 shows that even the cliche example of `power` already needs to propagate effects beyond a binder. We use the simplicity of the example to introduce the notation and our code generation library, and in §2.4, to demonstrate our key idea˜– hypothetical code generation and the rank-2 types of binding-form generators. §3 turns from mere exceptions to mutable state as the effect, storing open code in mutable variables across binders. This ability prompts the worry that code with unbound variables or with accidental bindings would be generated, but we demonstrate that attempts to write such a faulty generator are flagged as type errors. §4 describes our main example, `let`-insertion across binders. §5 justifies that a well-typed generator always generates well-typed code. We define lexical scope and show how our static types ensure it. We then discuss related work and conclude.

For lack of space, the presented examples are not fully self-contained. For brevity and clarity, we have adopted some conventions for presenting the code. The implicitly quantified type variable `repr` in type signatures represents a type that is a member of `SSym` or another 'symantics' class [8]; we almost always omit the corresponding constraint. We assume that the type variable `m` is constrained to be `Applicative`. We sometimes drop trivial injection-projection functions induced by `newtype`. Appendix A presents our library's public interface, the signatures of all its public functions, in full. The reader may refer to it if notational confusion arises. Furthermore, the complete code is available as the supplementary material to the paper and online at `http://okmij.org/ftp/tagless-final/TaglessStaged/`.

2. Warm-up

We say that the code generator belongs to the *metalanguage* and the generated code belongs to the *target language*. Because our target language is a subset of our metalanguage (as is often the case), we call the metalanguage *staged*: the generator is the present stage and the generated code is the future stage.

We begin with a simplistic example of effectful code generation, permitting the generator to fail and report an error, a character-string carrying exception that could be caught. The failure to finish the already started code generation is common in practice, especially when generation takes input from the user. Since the effect, the exception, carries a text string rather than a piece of code, there is no danger whatsoever of scope extrusion. The system $\lambda^{\oslash}$ of [19] could be trivially ad hoc extended to allow such effects to propagate beyond binders. Mint [51] has done such an extension, in the context of Java code generation. Our present approach, unlike that of [19, 51], does generalize to exceptions and other effects that do carry open code, §3. The simplicity of the example helps us introduce the code generation library and the idea of the approach.

2.1 Code generation library

We will be using the code-combinator approach [47, 52]. For now, we introduce four combinators:

```
class SSym repr where
  int  :: Int -> repr Int
  add  :: repr (Int -> Int -> Int)
  mul  :: repr (Int -> Int -> Int)
  ($$) :: repr (a->b) -> (repr a -> repr b)
infixl 2 $$
```

We define them using the "tagless final" approach [8], considering the future-stage code to be an domain-specific language embedded into Haskell. The language so far has integer literals, application, and two higher-order constants representing addition and multipli-

cation functions. The future-stage language is simply-typed, with the indicated types. Whereas `(1 + 2)::Int`, which is the same as `(+) 1 2`, is a Haskell expression for the present-stage addition, `exS1` of the characteristic type below

```
exS1 :: SSym repr => repr Int
exS1 = add $$ int 1 $$ int 2
```

represents the future-stage `Int` expression adding of two integers. We should have said, however pedantic for now, that `exS1` is the present-stage Haskell expression that, when evaluated, produces a value representing the future-stage addition. The types, inferred by Haskell compiler, make it clear when a Haskell expression represents a future-stage value. (We shall elide the constraint `SSym repr`.) (For the reader familiar with MetaOCaml §B describes the correspondence of our code generation approach and MetaOCaml. Bracket-and-escape syntax for the future-stage code is a syntactic sugar for code combinators, [9].)

The tagless-final approach permits several concrete realizations for our embedded language. For example, we may define instances of `SSym` instantiating `repr` with the type constructors `R` and `C` below:

```
newtype R a = R{unR :: a}
newtype C a = C{unC :: Int -> Exp}
```

The `R`-realization is the identity, ostensibly conflating the future and the present stages. Since Haskell is non-strict, it is more precise to say that the `R`-realization represents future-stage code as the present-stage 'thunk.' The `C`-realization uses Template Haskell (TH) datatype `Exp` that is essentially abstract syntax tree of Haskell code.[1] The values of `Exp` can be pretty-printed or spliced into another Haskell code. We rely on pretty-printing to see what we have generated; for example, instantiating `repr` to `C` in `exS1` as `unC exS1 0` and pretty-printing the resulting `Exp` gives us `"(GHC.Num.+) 1 2"`. (We shall elide the part "GHC.Num." when showing the code.) The `C`-representation, unlike `R`, describes truly future-stage code. Whereas the `R` is the typed generator of the typed code, `C` is the generator of untyped Template-Haskell expressions; `C` is still a typed generator. The representation that is polymorphic over `repr`, such as `exS1` above, abstracts the differences between `R` and `C`, ensuring at the same time that the generated code is well-typed [8] since `R` can generate only well-typed 'code'.

2.2 Lambda and power

We have not yet provided any way to build abstractions. That is the aspect in which our code generating library differs sharply from the state of the art, such as MetaOCaml or Template Haskell or the system of [52]. Here we recall the of the state art; §2.3 demonstrates its shortcomings.

We add a new code generating combinator:

```
class LamPure repr where
  lamS :: (repr a -> repr b) -> repr (a->b)
```

We will be using HOAS, relying on Haskell functions to represent functions bodies of our EDSL. For instance, the twice eta-expanded future-stage addition is represented as

```
exS2 :: repr (Int->Int->Int)
exS2 = lamS(\x -> lamS(\y -> add $$ x $$ y))
```

In HOAS, we use Haskell variables for future-stage variables; we can tell the stage from their type, `Int` or `repr Int`. The great benefit is that we can use human-readable names when we write code generators. For that reason, HOAS is very popular, extensively used by [52] and a few others; the MetaOCaml and TH quotation syntax for future-stage lambda-expressions is essentially syntactic sugar for HOAS ([52] have shown the correspondence, see also App. B). The tagless-final approach uses HOAS too [8]; that paper (and the accompanying code) describe the instances of `LamPure` for our two concrete realizations, `R` and `C`. The latter instance lets us see the generated code; for `exS2`, we get `"\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 too described in [8]. One would not want to write more than a couple of lines of code with deBruijn indices.

We now can write our running example, which is the staged power function. The example is immensely popular, having become a cliche. We shall see it still harbors a few surprises. The ordinary integer power function raises its argument `x` to the `n`-th power:

```
power :: Int -> Int -> Int
power 0 x = 1
power n x = x * power (n-1) x
```

We would like to write the (future-stage) code for `power`, specialized to the presently known value of the exponent. In other words, we want to generate a future-staged function that raises its argument to the a priori known power. Since we know `n` already at the present stage, we should unroll the recursion, leaving only multiplication to the future-stage. Staging the power is straightforward, and described in perhaps any book or paper on partial evaluation

```
spower :: Int -> repr Int -> repr Int
spower 0 x = int 1
spower n x = mul $$ x $$ spower (n-1) x
```

The types again tell the level: the first argument of `spower` is a present-stage integer, but the second argument and the result are future-level. One may view `int`, `$$` as a sort of binding-time annotations. We need to obtain the future-stage argument `x` from somewhere; it is bound at the future-stage:

```
spowern :: Int -> repr (Int -> Int)
spowern n = lamS (\x -> spower n x)
```

The code generated for `spowern 3`

```
"\\x_0 -> (*) x_0 ((*) x_0 ((*) x_0 1))"
```

is indeed as desired: the recursion is unrolled the statically known number of times.

2.3 The faulty power

Alas, the original `power` and its staged variant are partial functions. Evaluating either `power (-1) 2` or `spower (-1)` fails to terminate. The latter is unsatisfactory: we do wish our compiler and code generators always terminate. If a code generator is inherently partial it should report an exception to the calling function, which may catch that exception and handle, e.g., by generating code differently. The fact that the error reporting arises in the cliche power example testifies to the pervasiveness of this concern.

The question is how to report an exceptional condition encountered during code generation. We could call the 'function' `error` – which is denotationally is the same as non-termination and will crash the program (errors cannot be caught in the pure code). A crashed code generator is hardly better than a non-terminating one. The principled approach with the well-defined semantics is to model partiality explicitly, using the so-called Error monad (or Error applicative), lifting all values of the type `t` to `Either ErrMsg t`. We re-write our `power` reporting an error on the negative exponent (and relying on the fact that `Either ErrMsg` is a functor).

[1] The integer environment counts the level of a lambda-expression, to be considered shortly in the generated code, making sure that the variable names chosen when generating lambda-expressions are distinct within the expression. One may think of it as a weaker version of gensym, or as the annotations of all variable names by their level.

```
type ErrMsg = String
powerF :: Int -> Int -> Either ErrMsg Int
powerF 0 x = Right 1
powerF n x | n > 0 = fmap (x *) (powerF (n-1) x)
powerF _ _ = Left "negative exponent"
```

which we stage as before

```
spowerF :: Int -> repr Int -> Either ErrMsg (repr Int)
spowerF 0 x = Right (int 1)
spowerF n x | n > 0 =
  fmap (mul $$ x $$) (spowerF (n-1) x)
spowerF _ _ = Left "negative exponent"
```

The type of the result tells that the exception is reported during the code generation (not when the generated code is run)˜– which is what we want. It is when we try to complete the example, generating the future-stage function, that we encounter a problem: `\n -> lamS (\x -> spowerF n x)` cannot be typed. Indeed, the type of `(\x -> spowerF n x)` is `repr Int -> Either ErrMsg (repr Int)` whereas the type of `lamS` demands its argument to be of the type `repr Int -> repr Int`. We are fully stuck.

2.4 Hypothetical code generation

We now present our code generation library that solves the above problem. We get a hint by examining the implementation of `lamS` for the `R`-representation of `LamPure` (unlike the `C` representation, it has no phantom types that could be cast away, and so is the strictest.)

```
instance LamPure R where
    lamS f = R $ \x -> unR (f (R x))
```

The argument to `lamS` is a function that gives us the code for the function's body if we give it the code for the bound variable. We have to introduce the future-stage binding-form, the `R (\x -> ...)` part, so we obtain the bound variable, which we can then pass to `f`. That is the source of the problem: we have already committed to yielding the future-stage binding form, *before* we started the generation of the abstraction's body, which may fail to generate any code. The obvious solution then is to generate the code for the body of the abstraction first; only when we have succeeded should we introduce the future-stage binding form. We have to generate the body of the abstraction first, and bind the variable later.

The problem becomes of generating the body of the abstraction without knowing the bound variable. We have to *assume* the bound variable, and generate the code upon the assumption. The future-stage binder will discharge the assumption.

We introduce the type `HV h repr a` representing a future-stage value of the type `a` in the generator environment `h`. The latter records the assumptions of the bound variables. It is essentially the type environment for future-stage variables. The type `HV h repr a` is isomorphic to `h -> repr a`. It is defined more generally

```
newtype J m repr a = J{unJ :: m (repr a)}
type HV h = J ((->) h)
```

as a composition of two type constructors, `(->) h` and `repr`. Defining such a composition in Haskell requires the introduction of an auxiliary `newtype`, `J`, with bijections `J` and `unJ` witnessing the isomorphism between `J m repr` and the composition of `m` and `repr`. We elide these bijections for clarity.

Our goal is to lift our code combinators to the type `HV h repr a`. We do the lifting generically, noting that for each applicative functor [30] `m` and for each member of `SSym repr` their composition is too the member of `SSym`:

```
instance (Applicative m, SSym repr)
  => SSym (J m repr) where
    int = pure . int
    add = pure add
    mul = pure mul
    x $$ y = ($$) <$> x <*> y
```

We shall use this generic instance several times; for now we observe that `(->) h` is an applicative functor.

We also introduce the function to map, contravariantly, the type environment

```
hmap :: (h2 -> h1) -> HV h1 repr a -> HV h2 repr a
hmap f e = \h2 -> e (f h2)
```

and to obtain the future-stage code in the empty environment

```
runH :: HV () repr a -> repr a
runH m = m ()
```

The type environment `h` is, as usual, a sequence of elements representing the type of a corresponding to-be-bound variable. We represent the sequence as a nested tuple, whose elements are of the type future-stage code, for the bound variable, when it becomes known. We introduce a `newtype`-wrapper `H` so to attach a phantom type `s`, to be discussed later. The data constructor `H` is not exported from `TSCore.hs` and not available for the programmer.

```
newtype H r s a = H (r a)

href :: HV (H repr s a,h) repr a
href = \ (H x,h) -> x
```

We have also introduced `href`, to refer to the top assumption (the most-recently 'bound' variable). It, as `H` elimination form discarding our `s`, is also used only internally and not exported. Appendix A enumerates the public interface of our library, which treats the type `H` as abstract.

Before we describe how all this machinery is used and how to represent abstractions, we recall that our goal is effectful code generation. We will represent effects by an applicative functor, `m`: a Haskell value of the type `J m (HV h repr) a` represents a generator, in the generating applicative `m`, of the future-stage expression of the type `a` in the future-stage environment `h`. Since `HV h repr` is a member of `SSym`, so is `J m (HV h repr)` as we have just defined. That code reveals why we represent effects using applicative rather than monad. The generator of an application needs to generate the future-stage operator and the future-stage operand; both generators may incur effect, and the effect of generating operand may depend on the effect of generating the operator; the generator of the operand cannot depend on the *code* for the operator (after all, our approach provides no means of inspecting the generated code). As a syntactic sugar, we introduce an infix operator `(+:)` for the generator of addition

```
x +: y = add $$ x $$ y
```

and ditto for the multiplication `(*:)`. The code for addition of two integers can be generated as `int 1 +: int 2`.

We come to the crucial point: defining the combinator for generating future-stage lambda, letting effects from generating the body propagate. The definition is simple:

```
lam :: (forall s. HV (H repr s a,h) repr a
                -> J m (HV (H repr s a,h) repr) b)
       -> J m (HV h repr) (a->b)
lam f = fmap (\body -> \h ->
                    lamS (\x -> body (H x,h)))
              (f href)
```

We assume the code for the bound variable and pass it to the argument of `lam`, letting it generate the body of the abstraction, incurring an effect. The result is the value of the type `J m (HV`

`(H repr s a,h) repr) b`, which we convert to the desired `J m (HV h repr) (a->b)` using the existing tools. It is here that we use `lamS` to discharge the assumption of the bound variable. That discharge is "pure" and has no effects; that is why `fmap` suffices. We must stress that the key idea was the code generator type, which has the form `m (h -> repr a)` rather than more 'obvious' `(h -> m (repr a))`. The former lets us perform generation effects without knowing of bound variables (without looking at bound variables), which is precisely how staged code generation should proceed.

The type of `lam` is peculiar. First of all, argument of `lam`, the generator for the body, has the type `HV (H repr s a,h) repr a -> ...` rather than `J m (HV (H repr s a,h) repr) a -> ...`. The bound variable is represented as a code value, rather than a potentially effectful code expression. That is to be expected. To use the (potentially) bound variable in code expression, we have to lift it:

```
var :: HV h repr a -> J m (HV h repr) a
var = pure
```

More prominent is the second-rank type of `lam`, reminiscent of that of `runST` [26]. The quantification over type variable `s` prevents the bound variable, or the promise of the bound variable, to be precise, from leaking out of the abstraction.

Here is the first example of our final code generating combinators:

```
ex0 :: J m (HV h repr) (Int -> Int)
ex0 = lam(\x -> int 1 +: var x)
```

(the shown signature has been inferred). We can see the generated code by instantiating `repr` to `C` and `m` to the `Identity` applicative. To re-write `exS2`, with nested lambdas, we need an explicit weakening form

```
weaken :: J m (HV h repr) a -> J m (HV (h',h) repr) a
weaken m = fmap (hmap snd) m
```

witnessing the fact that more bound-variable assumptions can be added at any time. We then write:

```
exA2 = lam(\x -> lam(\y -> weaken (var x) +: var y))
```

The placement of `weaken` in the above code is type-directed. Without `weaken`, the type checker rejects the code, showing in the error message the mismatch of the environments. Therefore, we could have defined `var` through an overloaded function `weakens`, which automatically inserts the necessary number of `weaken` applications (similar to the `lIO` in [22]):

```
var :: Extends h h' => HV h repr a -> J m (HV h' repr) a
var = weakens . pure
```

with the constraint `Extends h h'` witnessing that `h'` is either `h` or an extension of it with more hypotheses. At present, we prefer being explicit, to facilitate formalization. For practical programming, hiding `weaken` is of course preferable.

More interesting use of `weaken` is code transformer, which takes a future-stage expression, which may contain potentially bound variables, and uses it in an expression with more potentially bound variables. This example was described in [18, §2.3].

```
ef :: J m (HV h repr) Int -> J m (HV h repr) (Int -> Int)
ef z = lam (\x -> weaken z +: var x)
ef2 = lam (\x -> lam (\y ->
            ef (weaken (var x) *: var y)))
```

It is unproblematic, the generated code is:

```
"\\x_0 -> \\x_1 -> \\x_2 -> (+) ((*) x_0 x_1) x_2"
```

2.5 Succeeding faulty power

We now complete our running example. We re-write `spowerF` using the more general code combinators. The code is essentially unchanged, only the type is more general, reflecting effectful and hypothetical code generation.

```
spowerAF :: Int -> J (Either ErrMsg) (HV h repr) Int
          -> J (Either ErrMsg) (HV h repr) Int
spowerAF 0 x = int 1
spowerAF n x | n > 0 = x *: spowerAF (n-1) x
spowerAF _ _ = Left "negative exponent"
```

Generating the future-stage function, the power specialized to the given `n`, now type checks, with the following (inferred) signature:

```
spowerAFn :: Int ->
    J (Either ErrMsg) (HV h repr) (Int -> Int)
spowerAFn n = lam (\x -> spowerAF n (var x))
```

According to the type, we obtain code in an applicative `Either ErrMsg`. Supplying the value for `n` and instantiating `repr` to `C` shows the code. In particular, the result of `spowerAFn (-1)` is `Left "negative exponent"`, the expected exception. The problem solved.

3. Moving open code

The warm-up example in §2 was rather simple, and could be implemented with the existing techniques, such as Mint [51] or a trivial ad hoc extension of [19]. The code generation library introduced in §2 permits however the manipulation of essentially 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 [19], which argued that in practice 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 the example, we extend our future-stage language with the form `assertPos`

```
class AssertPos repr where
  assertPos :: repr Int -> repr a -> repr a
```

(the tagless-final approach makes extending the 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` checks to see if the value of `test` is positive. If so, the second argument, `m`, is evaluated and its value is returned. Otherwise, a run-time error is raised and the program is aborted. As we did in §2.4, we define an instance of `AssertPos` for a composition of `repr` with any applicative `m`. We also extend our future-stage language with the integer division operation `(/:)`.

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

```
guarded_div1 :: J m (HV h repr) Int ->
   J m (HV h repr) Int -> J m (HV h 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.

We can accomplish the movement with reference cells. We allocate a reference cell holding a code-to-code transformer, originally identity. We generate code passing the generator the reference cell. After the generator is finished, we retrieve the resulting transformer and apply it to the result of the generated code. The generator may add assertions by modifying the contents of the cell, composing the current transformer with `assertPos test`. The following code implements the idea, using the `IO` as the generating applicative, and its reference cells `IORef` (we could have used the `ST s` or any other monad with reference cells).

```
assert_locus ::
  (IORef (J IO repr a -> J IO repr a) -> J IO repr a)
   -> J IO repr a
assert_locus m = do
  assert_code_ref <- newIORef id
  mv <- m assert_code_ref
  transformer <- readIORef assert_code_ref
  transformer (return mv)
```

We re-define guarded division to insert the positive divisor assertion at the given locus

```
add_assert :: IORef (a -> a) -> (a->a)
     -> J IO repr b -> J IO repr b
add_assert locus transformer m =
  modifyIORef locus ( . transformer) >> m

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 (var y)`, includes the open code. We do store functions that contain open code. The reference cell that accumulates the transformer is used in the example completely inside a binder. There is no risk of scope extrusion then. The above example is implementable in the approach of [19].

Now we can generalize. First we slightly generalize guarded division, inserting the generic `weakens` §2.4. The inferred signature, shown slightly abbreviated, tells the difference

```
guarded_div3 :: (Extends h h1, ...) =>
     IORef (J IO (HV h repr) a -> J IO (HV h repr) a)
     -> J IO (HV h1 repr) Int
     -> J IO (HV h repr) Int
     -> J IO (HV h1 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, `h'`, may be weaker, by an arbitrary amount. The generalized `guarded_div3` can be used in place of `guarded_div2` in the above example. We can also write a more general example

```
exdiv3 = lam (\y -> assert_locus $ \locus ->
          lam (\x ->
            complex_exp +:
              guarded_div3 locus (var x) (var y)))
```

with an extra binding. 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 the function with the open code, `assertPos (var y)` has moved across the binder, `lam (\x -> ...)`. If we make a mistake and switch `var x` and `var y` as the arguments of `guarded_div3`, thus attempting to move `assertPos (var x)` beyond the binder for `x`, the type checker reports a problem

```
Inferred type is less polymorphic than expected
 Quantified type variable ‘s’ is mentioned
  in the environment:
  locus :: IORef (J IO (HV (H repr s Int, h) repr) a
           -> J IO (HV (H repr s Int, h) repr) a)
 In the first argument of ‘lam’, namely
 ‘(\ x -> complex_exp +:
           guarded_div3 locus (var y) (var x))’
```

telling us that the type of `locus` mentions the `s` that is quantified by the `x`'s binding form. In other words, the `x` binding leaks. Scope extrusion indeed becomes a type error. (The generated code is spelled in full as regression tests of the generators, in the code accompanying the article.)

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, we don't have to pass `locus` 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.

4. Inserting `let` across binders

We have come to the ultimate application, `let`-insertion, or the generation of code containing explicit sharing of the results of some sub-expressions, thus eliminating duplication of the code for these sub-expressions. If the generated code is imperative, controlling code duplication is not only desirable but necessary. For that reason, `let`-insertion is used extensively in partial evaluation, staging [44] and other meta-programming. It has long been discovered in the partial evaluation community that the principled (rather than tree hacking) `let`-insertion requires writing the code or the generator in the continuation-passing style [4] (see detailed explanation in [7, Section 3.1]) or else use control operators [27]. Continuation-passing style cannot insert future-stage `let` beyond the closest future-stage binder without risking scope extrusion. Likewise, using control operators with the restriction to ensure the absence of scope extrusion [19] keeps inserted `let` under the closest binder. However, `let`-insertion across binders may be necessary; the latter paper described several such cases as open problems. We now demonstrate the solution, with the same safety guarantees.

First we add the future-stage `let_` to our DSL, which, from the signature, below, looks like a combination of `lam` and application˜– which is what `let` is.

```
let_ :: J m (HV h repr) a
      -> (forall s. HV (H repr s a,h1) repr a
             -> J m (HV (H repr s a,h) repr) b)
      -> J m (HV h repr) b
```

Whereas the generator using Haskell's let

```
let x = int 1 +: int 2 in x *: x
-- "(*) ((+) 1 2) ((+) 1 2)"
```

produces the code (shown underneath in the comments) with the obvious code duplication, the generator relying on future-stage `let`

```
let_ (int 1 +: int 2) $ \x -> var x *: var x
-- "let z_0 = (+) 1 2\n 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).

Second, we should write the generator in the continuation-passing style, or in the applicative `CPS w` (which is the standard Haskell delimited continuation monad, taken here as Applicative):

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

The let-insertion primitive, to be called `genlet'` here, has been well-explained in [7, 44]. Re-writing it in our library is straightforward:

```
genlet' e = CPS $ \k ->
             runCPS $ let_ e (\x -> pure (k x))
```

Alas, in our library, which is more general and precise, this code does not type-check. Before we get to the problem let us show that at least the intention is correct, by comparing with `gennolet`:

```
gennolet :: J (CPS w) (HV h repr) a
            -> J (CPS w) (HV h repr) a
gennolet e = CPS $ \k -> unCPS e (\v -> k v)
```

which is the identity function (`gennolet e` is two η-expansions away from `e`). The expression `gennolet e` evaluates the generator `e` and passes the result `v` to the continuation of `gennolet e`. According to its type, `v` is a future-stage *expression*, potentially quite complex. The expression `genlet' e` too evaluates `e`; it generates the future-stage let-expression binding the code generated by `e` to a fresh, future-stage variable, passing the code of that variable to the continuation `k`; the continuation thus receives an atomic future-stage expression (the variable reference).

Let us now examine the typing problem of `genlet'`. The continuation `k` has the type `a->w`, or, more explicitly, `HV ha repr a -> HV hw repr w`. The continuation thus is a transformer for future-stage values (in general, `ha` is different from `hw`: the transformed code may have different environment). The type of `let_` shows that the body of `let_` has the environment `(H repr s a,h)` with a private slot `H repr s a` for the let-bound variable. Here lies the problem: first, we need to find a way to 'convert' the captured continuation from the type `HV ha repr a -> HV hw repr w` to `HV (H repr s a, ha) repr a -> HV (H repr s a, hw) repr w`. Second, we have to ensure that the 'weakened continuation' never looks at the slot `H repr s a` in the environment but merely passes it along. Only then the Haskell type-checker will be satisfied that the 'name' of the let-bound variable (the quantified type variable `s`, to be precise) does not 'leak'.

The general solution to this problem, allowing safely layering delimited control and let-insertion on top of other effects such as exceptions, tracing, or another (outer) level of delimited control, shown in the accompanying code (file `TSCPST.hs`), is quite complex. We describe here a simplified version, to convey general intuitions. The simplification is only valid for (single-level), pure `CPS` type, rather than for the `CPS` transformer. The trick is to pass the `H repr s a` slot for the let-bound variable using the metalanguage (Haskell) environment. Examining the ill-typed definition `genlet'` and noting the type of `x`, which is `HV (H repr s a,hx) repr a` (we can chose `hx` to be anything, for example, `()`) and the desired type for `k x`, which is `HV (H repr s a,h) repr b` points out the way to 'route' `H repr s a` around `k`, thus fixing the code:

```
genlet :: J (CPS (HV hw repr w)) (HV hw repr) a
     -> J (CPS (HV hw repr w)) (HV ha repr) a
genlet e = CPS $ \k -> runCPS $ let_ e (\x ->
   pure $ \ (h1,hw) -> k (\ha -> x (h1,())) hw)
```

No 'weakening' of k is required then. What remains is to define a convenient combinator to mark the place where `let` is to be inserted:

```
reset :: J (CPS (repr a)) repr a -> J (CPS w) repr a
reset m = pure $ runCPS m
```

We now show a few examples of let-insertion across the binders, the simplest being

```
reset $ 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-insertion point, marked by `reset`, may be arbitrarily number of binders away from the `genlet` expression:

```
reset $ 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 `reset`, which is outside the `x`'s binder.

```
reset $ lam (\x ->
  (lam (\y -> var y +: weaken (var x) +:
              genlet (var x +: int 3))))

  Inferred type is less polymorphic than expected
    Quantified type variable 's' escapes
  In the first argument of 'lam', namely
      '(\ x -> (lam ...))
```

The type checker reports the error, pointing out the binder whose variable escapes. We must move the insertion point within that binder, moving the reset:

```
lam (\x ->
  reset (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` expression and even nest them:

```
lam (\x -> reset (lam (\y ->
   int 1 +: genlet (var x +: genlet (int 3 +: int 4))
   +: genlet (int 5 +: int 6))))
-- \x_0 -> let z_1 = (+) 3 4 in
--         let z_2 = (+) x_0 z_1 in
--         let z_3 = (+) 5 6 in
--         \x_4 -> (+) ((+) 1 z_2) z_3
```

The generated code shows the consequences of our simplification: since one of the let-bound expressions contain the variable `x`, we must insert `reset` under the binder for `x`, ever preventing let-insertion 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 CPS hierarchy [12], obtained by generalizing CPS to CPS transformer

```
newtype CPST w m a = CPS{unCPS :: (a -> m w) -> m w}
```

(for Applicative m) and iterating these transformers. Unfortunately, our simple fix for genlet does not generalize to CPS transformer. The fix relied on the fact that the continuation k mapped a future-stage code value HV ha repr a to another a future-stage code value. In other words, the application k x has no (visible) control effects. That is no longer true for the CPS transformer: the result of k x is an effectful expression.

The accompanying code shows a general solution, with the following rank-3 type of the applicative CPS transformer:

```
newtype CPSA w m a =
    CPSA{unCPSA :: forall hw.
          (forall h1. m (h1->hw->a) -> m (h1->hw->w))
            -> m (hw -> w)}
```

Before looking at the code, the reader is encouraged to derive an applicative instance for CPSA w m as an exercises. The type arguments w and a are intended to be HV types. The earlier example of nested genlet now looks as follows

```
lam (\x -> reset (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
```

generating code in which different let-bound expressions are moved to different places, as far a spossible, crossing a number of future-stage binders, including the binders introduced by earlier genlet.

5. Safety properties

We state the static safety properties of our code generators.

Proposition 1 If the evaluation of an expression e :: J m (HV () repr) a terminates and yields the value v :: repr a, then the top-level Template-Haskell splice $(unC v 0) evaluates without errors and produces an expression of the type a.

That is, a well-typed generator produces only well-typed code, specifically, a well-typed generator in the empty environment produces a well-typed closed code, which compiles, or can be spliced in, with no errors. The proof is along the lines of [8]. Here is the outline: first we verify by inspection that the Template Haskell code produced in the C realization of repr corresponds to the 'code' produced in the R realization. That is, the top-level splice of the result of unC (int n) 0 is the same as unR (int n) for any integer n. Likewise for other primitive forms. Because Haskell enjoys subject reduction and R involves no staging, a well-typed generator (polymorphic over repr) will produce well-typed Haskell code with repr instantiated to˜R. Thus the same generator, with repr instantiated to˜C, will produce well-typed Template Haskell code.

5.1 Non-examples of 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. In the next section, we enforce a notion of lexical scope using static types.

As the first example of intuitively wrong behavior we use the one from [9, Section 3.3]. The example, unfortunately admitted in the system of [9], exhibits the problem that bindings "vanish or occur 'unexpectedly' ". The example can be translated to our library:

```
exCX f = unsafeLam(\y -> unsafeLam (\x -> f (var x)))
```

where unsafeLam is the *unsafe* version of the lam future-stage abstraction constructor, without the higher-rank type (without the forall s). 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"

exCX_c2 = exCX (fmap.hmap $ \(y,(x,z)) -> (x,(y,z)))
-- "\\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 (HV (H repr s1 b,(H repr s a,h)) repr b)
    -> m (HV (H repr s1 b,(H repr s a,h)) repr c))
  -> m (HV h repr (a -> b -> c))
```

The type says that the argument of exCX maps future-stage code valid in the environment with at least two slots into future-stage 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 future-stage variable should correspond to *some* slot in the (explicit) future-stage environment, we lose lexical scoping for the generated code. We cannot statically tell the correspondence between binding and reference occurrences of future-stage variables. We thus give further, clearer evidence for the argument of Pouillard and Pottier [40] 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 [9] 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 s. Because the openness of the future-stage code is apparent in type and the environment slots contain s, transformers of open code must have higher-rank. We must give an explicit signature. For example, we can specify

```
exCX2 :: (forall h. m (HV h repr b) -> m (HV h repr c))
     -> m (HV h repr (a -> b -> c))
exCX2 f = lam(\y -> lam (\x -> f (var x)))
```

that the function f does not even depend on the environment. We may apply exCX2 to the identity function but we cannot write the analogue of exCX_c2. We may try to give the signature that specifically permits the environment-shuffling f:

```
-- ill-typed!
exCX4 :: (forall h1 h2. m (HV (h1,(h2,h)) repr a)
          -> m (HV (h2,(h1,h)) repr a))
     -> m (HV h repr (a -> a -> a))
```

It is rejected by the type checker because of the attempt to identify the `s` associated with˜`x` and the `s` associated with˜`y`. These two `s` are independently quantified and not unifiable. Thus, our system disallows `exCX_c2`. The mapping between bound and reference occurrences of the variables is statically apparent in our system. Since we identify future-stage variables with quantified type variables `s`, the scope of future-stage variables is the quantification scope of the corresponding `s` 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. [21, §6.4]. Written with our library, it is as follows:

```
exKYC1 :: IO (HV h 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 variable `y` is captured by the new abstraction in `g`; if the programmer used the system of Chen and Xi [9], 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, is an offence. We can only write `exKYC1` if we deliberately break our library; inserting even one regular, safe `lam` provokes the ire of the type checker.

The file `Unsafe.hs` in the accompanying code has the complete code for these examples. The file describes a third example, which uses a control effect (throwing an exception) to smuggle a bound variable beyond its binder. Unlike `exKYC1` above, the smuggling is far less textually obvious and so is easy for a human programmer to overlook. The problem is detected by the type checker, if we use `lam` rather than the deliberately broken `unsafeLam`.

5.2 Lexical scope

We now make precise the notion of lexical scope that was intuitively violated in the examples of the previous section. We instrument code generators with integer labels. We introduce an instrumented `lamS` from §2.2 that accepts a label; we also add a form to the target language that checks if a code value has the expected label (crashing the program otherwise).

```
type Label = Int
class LamLPure repr where
  lamSL :: Label -> (repr a -> repr b) -> repr (a->b)
  check_label :: Label -> repr a -> repr a
```

The label assignment is done by the instrumented `lam` form, the effectful code generator.

```
lamL :: (forall s. HV (H repr s a,h) repr a
    -> J (State Label m) (HV (H repr s a,h) repr) b)
   -> J (State Label m) (HV h repr) (a->b)
lamL f = do
  l <- newLabel
  fmap (\body -> \h -> lamSL l (\x -> body (H x,h)))
       (f (hrefL l))
hrefL l = \ (H x,h) -> check_label l x
```

A fresh label is obtained before generating the abstraction body, and used to label the variable to be used in the body, and the generated abstraction itself. For example, the instrumented generator

```
exL0 :: J (State Label m) (HV h repr)
          (Int -> Int -> Int)
exL0 = lamL(\x -> lamL(\y -> weaken (var x) +: var y))
```

yields a `repr (Int -> Int -> Int)` code value

```
lamSL 0 (\v0 -> lamSL 1 (\v1 ->
   add $$ (check_label 0 v0) (check_label 1 v1)))
```

which, upon instantiation of `repr` to `C`, produces the TH code and checks the labelling, that `v0` is indeed bound by a binder labelled with 0, etc. One may regard that checking as a verification pass after the generation has completed. In the example above, the verification succeeds; that is, the `check_label` assertions all succeed. If, however, we use the instrumented interpreter˜– and the deliberately broken `unsafeLamL`˜– with the examples `exCX_c2` and `exKYC1` of the previous section, the verification fails because some variables turn out with unexpected labels.

The verification formalizes our intuitions, testing that the labelling of binders and bound variables performed before the code generation is preserved in the generated code. Successful verification assures us that all bindings in the generated code meet our expectations; that is, the correspondence of future-stage variables to their binding forms is preserved during the generation, no matter the effects. The verification pass takes place after the code has been generated. One desires however to statically assure its success. Our type system does so using the rank-2 type of `lam`.

Proposition 2 If the evaluation of an expression `e :: J (State Label m) (HV () repr) a` terminates and thus yields the future-stage code value `v :: repr a`, then the verification of `v` succeeds.

Since well-typed generators yield verifiable code, we can dispense with the verification and erase the labels.

The proof of the proposition depends on the correspondence between a fresh label and the quantified type variable `s`, seen in the `lamL` code. Binders and variables are labeled with `l`, and so are their types with the corresponding `s`. The type checker ensures that each `lamL` correspond to unique `s`, unifiable only with itself. Finally we notice that the `s` elimination forms, `href` and `hrefL`, are used only in `lam`, not exported from `TSCore` and not available to the user of our library.

6. Related work

6.1 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 [14, 48], 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 the 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 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 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): throwing simple exceptions in code generators; (2) Gibonacci [44], an epitome of code generation with memoization; (3) assert-insertion beyond the closest binder, §3; (4) let-insertion beyond the binder, §4. Only the present work implements all four benchmarks; even assert-insertion was not reachable before with statically assured generators.

The work [15] presents a type-and-effect systems for meta-programming with exceptions, allowing exception propagation beyond future-stage 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. [5, 6] permitted mutations but only of the closed code; the approach cannot therefore implement the Gibonacci benchmark.

Mint [51] 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. [44] and Kameyama et˜al. [19] 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 [19] can be trivially extended for that case (imposing the same restrictions on values are thrown from under the binder as those of Mint). The two systems hit the local optimum, allowing writing moderately complex generators, e.g., [7].

The parallel line of work [21] attempts to formalize and make safe Lisp practice of generating code with concrete symbolic names. The variable capture is specifically allowed and future-stage lexical scope is not assured statically.

6.2 Contextual systems

In our approach, code generators may produce open future-stage code and have the type that includes the future-stage typing environment. Moreover, the type contains the 'names', or the type-level `s` proxies, for term's free future-stage variables. The latter fact in particular relates our work to the contextual modal type theory [33]. 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 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 [28]. Unlike them, we do not need negative, freshness constraints because our labels `s` are always chosen fresh by the type checker.

Environment classifiers [46] is an elegant simplification of contextual model 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.

6.3 Programming with names

The nominal tradition has been extensively reviewed in [40]. 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 §5.2.

Our approach has many similarities with that of [40], in particular, their de Bruijn-index implementation. Our environment `h` is quite like `World`, used to parameterize future-stage terms and their types. We use the concrete representation of a world as a nested tuple; hence, world inclusion is apparent and can be decided by Haskell type checker, rather than taken to be a primitive as it is in [40]. We both use rank-2 type for the future-stage binding form. Pouillard and Pottier's $\texttt{import}_{\subseteq}$ is quite like our `weaken`, and $\subseteq -$`trans` corresponds to our `Extends`.

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 generator 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 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 [40] 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 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 first benefit of the pure generative restriction is simplicity. The framework of [40] required the power of dependent typed 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 [39], 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 [36, 38]. In contrast, we implement our code generation library in ordinary Haskell. Safety depends only on limiting the access to the `H` data constructor, so that the phantom `s` type cannot be cast away.

The main benefit of generative approach is a richer equational theory: as argued in [45, 50] 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`.) Experience showed that pure generative approach, albeit seemingly restrictive, does not prevent generation of highly optimal code [23, 24].

The system of [40] 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, [9]. Although the approach ensures that all variables in the generated code are bound, the binding may be unanticipated, see §5.1. The problem was indicated in the review of [40], although it has been pointed out by [21] and already in [9]. Although our representation of future-stage environment by nested tuples is reminiscent of well-scoped deBruijn-index approach, our use of rank-2 type for future-stage binders prevents unintended per-

mutations of the tuples 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 [52]) 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 §5.1). We demonstrate how to solve both problems, manipulation of open code and prevention of free variable elimination, without giving up conveniences of HOAS. [9] need type annotations even for local definitions. Also, [9] acknowledge that their language is experimental and integrating to the full-fledged language is left for future work.

6.4 Hygienic macros

The long tradition of code generation, or macros, in Lisp systems has long pointed out the dangers of variable capture and the need to maintain the *hygiene* of macro-expansion [25]. Alas, defining what it means precisely has been elusive [16, 17]. 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 [17] is, like ours, pure generative.

7. Conclusions

We have presented the 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 safe `let`-insertion across an arbitrary number of generated binders. The approach statically assures 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 [46]: "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, *present-stage types tell future-stage scopes.*

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 future-stage 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 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.

References

[1] Balat, Vincent, and Olivier Danvy. 2002. Memoization in type-directed partial evaluation. In *GPCE*, 78–92. LNCS 2487.

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

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

[4] Bondorf, Anders. 1992. Improving binding times without explicit CPS-conversion. In *Lisp & functional programming*, 1–10.

[5] Calcagno, Cristiano, Eugenio Moggi, and Tim Sheard. 2003. Closed types for a safe imperative MetaML. *Journal of Functional Programming* 13(3):545–571.

[6] Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2000. Closed types as a simple approach to safe imperative multi-stage programming. In *ICALP*, 25–36. LNCS 1853.

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

[8] Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19(5):509–543.

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

[10] Choi, Wontae, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. 2011. Static analysis of multi-staged programs via unstaging translation. In *POPL '11: Conference record of the annual ACM symposium on principles of programming languages*, ed. Thomas Ball and Mooly Sagiv, 81–92. New York: ACM Press.

[11] Cohen, Albert, Sébastien Donadio, María˜Jesús Garzarán, Christoph˜Armin Herrmann, Oleg Kiselyov, and David˜A. Padua. 2006. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming* 62(1):25–46.

[12] Danvy, Olivier, and Andrzej Filinski. 1990. Abstracting control. In *Lisp & functional programming*, 151–160.

[13] ———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

[14] Dussart, Dirk, and Peter Thiemann. 1996. Imperative functional specialization. Tech. Rep. WSI-96-28, Universität Tübingen.

[15] Eo, Hyunjun, Ik-Soon Kim, and Kwangkeun Yi. 2006. Type and effect system for multi-staged exceptions. In *APLAS*, 61–78. LNCS 4279.

[16] Herman, David. 2010. A theory of typed hygienic macros. Ph.D. thesis, Northeastern University, Boston, MA.

[17] Herman, David, and Mitchell Wand. 2008. A theory of hygienic macros. In *ESOP '08: Proc. european symp. on programming*.

[18] Kameyama, Yukiyoshi, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the stage: From staged code to typed closures. In *PEPM*, 147–157.

[19] ———. 2009. Shifting the stage: Staging with delimited control. In *PEPM*, 111–120. New York: ACM Press.

[20] Keller, Gabriele, Hugh Chaffey-Millar, Manuel M.˜T. Chakravarty, Don Stewart, and Christopher Barner-Kowollik. 2008. Specialising simulator generators for high-performance Monte-Carlo methods. In *PADL*. LNCS.

[21] Kim, Ik-Soon, Kwangkeun Yi, and Cristiano Calcagno. 2006. A polymorphic modal type system for Lisp-like multi-staged languages. In *POPL*, 257–268.

[22] Kiselyov, Oleg, and Chung-chieh Shan. 2008. Lightweight monadic regions. In *Symposium on Haskell*, ed. Andrew Gill, 1–12.

[23] Kiselyov, Oleg, Kedar˜N. Swadi, and Walid Taha. 2004. A methodology for generating verified combinatorial circuits. In *EMSOFT*, 249–258.

[24] Kiselyov, Oleg, and Walid Taha. 2005. Relating FFTW and split-radix. In *ICESS*, ed. Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu, 488–493. LNCS 3605.

[25] Kohlbecker, Eugene, Daniel˜P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *LFP*, 151–161. New York: ACM Press.

[26] Launchbury, John, and Simon˜L. Peyton˜Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

[27] Lawall, Julia˜L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Lisp & functional programming*, 227–238.

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

[29] Lengauer, Christian, and Walid Taha, eds. 2006. *Special issue on the 1st MetaOCaml workshop (2004)*, vol. 62(1) of *Science of Computer Programming*.

[30] McBride, Conor, and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18(1):1–13.

[31] Miller, Dale, and Gopalan Nadathur. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE symposium on logic programming*, ed. Seif Haridi, 379–388. Washington, DC: IEEE Computer Society Press.

[32] Moggi, Eugenio. 1991. Notions of computation and monads. *Information and Computation* 93(1):55–92.

[33] Nanevski, Aleksandar, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *Transactions on Computational Logic* 9(3):23:1–49.

[34] Peyton˜Jones, Simon˜L. 2010. New directions for Template Haskell. `http://hackage.haskell.org/trac/ghc/blog/Template\%20Haskell\%20Proposal`.

[35] Pfenning, Frank, and Conal Elliott. 1988. Higher-order abstract syntax. In *PLDI '88*, vol. 23(7) of *ACM SIGPLAN Notices*, 199–208. New York: ACM Press.

[36] Pientka, Brigitte. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*, 371–382.

[37] POPL. 2003. *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*.

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

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

[40] Pouillard, Nicolas, and François Pottier. 2010. A fresh look at programming with names and binders. In *ICFP*, 217–228. New York: ACM Press.

[41] Püschel, Markus, José M.˜F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan˜W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert˜W. Johnson, and Nick Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE* 93(2):232–275.

[42] Solar-Lezama, Armando, Liviu Tancau, Rastislav Bodík, Sanjit˜A. Seshia, and Vijay˜A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*, 404–415. New York: ACM Press.

[43] Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation* 14(2–3):101–142.

[44] Swadi, Kedar, Walid Taha, Oleg Kiselyov, and Emir Pašalić. 2006. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM*, 160–169.

[45] Taha, Walid. 2000. A sound reduction semantics for untyped CBN multi-stage computation. In *PEPM*, 34–43.

[46] Taha, Walid, and Michael˜Florentin Nielsen. 2003. Environment classifiers. In [37], 26–37.

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

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

[49] Wadler, Philip˜L. 1992. The essence of functional programming. In *POPL*, 1–14.

[50] Wand, Mitchell. 1998. The theory of fexprs is trivial. *Lisp and Symbolic Computation* 10(3):189–199.

[51] Westbrook, Edwin, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java multi-stage programming using weak separability. In *PLDI '10*. New York: ACM Press.

[52] Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In [37], 224–235.

A. Public interface of code generation library

The public interface of our code generation library: 'Staged Haskell'

```
class SSym repr where
  int :: Int -> repr Int
  add :: repr (Int -> Int -> Int)
  mul :: repr (Int -> Int -> Int)
 ($$) :: repr (a->b) -> (repr a -> repr b)
infixl 2 $$

class LamPure repr where
  lamS :: (repr a -> repr b) -> repr (a->b)

newtype J m repr a = J{unJ :: m (repr a)}

newtype R a = R{unR :: a}
newtype C a = C{unC :: VarCounter -> Exp}
runCS :: C a -> String

type HV h = J ((->) h)
newtype H r s a -- abstract

hmap :: (h2 -> h1) -> HV h1 repr a -> HV h2 repr a

weaken :: Applicative m =>
          J m (HV h repr) a -> J m (HV (h',h) repr) a

(+:),(*:) :: (SSym repr) =>
              repr Int -> repr Int -> repr Int

lam :: (Functor m, SSym repr, LamPure repr) =>
       (forall s. HV (H repr s a,h) repr a ->
                  J m (HV (H repr s a,h) repr) b)
       -> J m (HV h repr) (a->b)
var :: (Applicative m, SSym repr) =>
       HV h repr a -> J m (HV h repr) a

class Extends h h' where
  weakens :: Applicative m =>
             J m (HV h repr) a -> J m (HV h' repr) a

runC :: Applicative i => J i (HV () C) a -> i String
runR :: Applicative i => J i (HV () R) a -> i a

class AssertPos repr where
    assertPos :: repr Int -> repr a -> repr a

class SymDIV repr
(/:) :: (Applicative m, SSym repr, SymDIV repr) =>
        J m (HV h repr) Int -> J m (HV h repr) Int
        -> J m (HV h repr) Int

class SymLet repr where
let_ :: (SSym repr, SymLet repr, Applicative m) =>
      J m (HV h repr) a
      -> (forall s. HV (H repr s a,h1) repr a ->
            J m (HV (H repr s a,h) repr) b)
      -> J m (HV h repr) b

type Label = Int
class LamLPure repr
newtype State l m a = State{unState :: l -> m (a,l)}
lamL :: (Functor m,Monad m,SSym repr,LamLPure repr) =>
   (forall s. HV (H repr s a,h) repr a
     -> J (State Label m) (HV (H repr s a,h) repr) b)
   -> J (State Label m) (HV h repr) (a->b)
runCL :: (Functor m, Monad m) =>
         J (State Label m) (HV () C) a -> m String
```

B. Connection with MetaOCaml

Our code combinators can be expressed using brackets, escapes and cross-staged persistence (for selected types, e.g. `Int` in case of `int`) of MetaOCaml.

```
let int x    =  <x>
let add      =  <(*)>
let ($$) x y =  <~x ~y>

let lamS f   =  <fun x -> ~(f <x>)>
```

Like MetaOCaml, we also keep track of the (future stage) environment, distinguishing a closed expression like `<1>` from the open expression `<x>`. We keep a far detailed track than does MetaOCaml, tracking each free variable rather than a set of variables of the same classifier. Thus our type `HV h repr a` is roughly equivalent to `('h,'a) code` value of MetaOCaml. Roughly because our `h` has a lot more structure than just the single environment classifier.

We are more precise than MetaOCaml in another aspect, keeping track of effects of code generation as well. In MetaOCaml, as in OCaml in general, all expressions are effectful. Therefore, our type `m (HV h repr a)` corresponds to a general MetaOCaml expression, with effects, that will generate the code of type `a` in the future-stage environment `h`.