Combinators for Impure yet Hygienic Code Generation

Yukiyoshi Kameyama Oleg Kiselyov Chung-chieh Shan

PEPM 2014
San Diego, CA Jan 20, 2014

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.

Conclusions

Writing programs that generate programs

Code combinator framework in Haskell:

- Expressive:
  any monadic effect in a generator
- Static assurance:
  generated code is well-formed and well-typed, *at all times*
- Implemented in a mature language
- Modular and composable:
  polymorphism over generated environments
- Extensible:
  with more language forms, more target languages
- Relatively convenient: HOAS rather than De Bruijn

The general topic of the paper is metaprogramming: writing programs that generate programs. And to this we contribute a framework, a Haskell library to write Haskell programs that generate code in some target language, a subset of Haskell, OCaml or, say, Javascript. The framework is expressive: any monadic effect can be used in a generator. We'll talk later why we need effects in a generator. The framework statically assures that not only the final generated code but all intermediate results are well-scoped and well-typed. Attempts to generate ill-scoped or ill-typed code are prosecuted right away, and errors are reported in terms of the generator. Our framework is *not* a an experimental language with new syntax and uncertain future. Rather, it is a regular, not a bleeding edge, library in mature Haskell. It is modular and extensible. It allows polymorphism over generated environments. The language of generated code can easily be extended with more features and constants (the paper shows many examples) or changed to be typed or untyped, first- or higher-order.

Because of the HOAS the variable names are human-readable.

Conclusions

Writing programs that generate programs

Code combinator framework in Haskell:

- Expressive:
  any monadic effect in a generator
- Static assurance:
  generated code is well-formed and well-typed, *at all times*
- Implemented in a mature language
- Modular and composable:
  polymorphism over generated environments
- Extensible:
  with more language forms, more target languages
- Relatively convenient: HOAS rather than De Bruijn

All these properties hold simultaneously

Mainly: although there are many frameworks that have one or the other properties, ours is the one that has all these properties simultaneously.

Conclusions

Writing programs that generate programs

Code combinator framework in Haskell:

- Expressive:
- Static assurance:
- Implemented in a mature language
- Modular and composable:
- Extensible:
- Relatively convenient: HOAS rather than De Bruijn

Paper

- explanations, illustrations, examples
- realistic example: loop tiling

The whole code:
`http://okmij.org/ftp/tagless-final/TaglessStaged/`

The paper has a practical bend and tries to show how to *use* the library on many examples. Some are realistic: loop tiling. The full code is available on the web, at this URL, and you are welcome to use it.

Outline

- Paradise gained
  - Lexical scope
  - Lexical scope in generators
  - Uneffectful generators: simple paradise
- Paradise lost
  - Effectful temptation
  - Effects: so much gain, so much loss
  - Representing open code: desperately seeking abstraction
- A glimpse of hope: Applicatives

Lexical scope

That was quick, wasn't it. I have a bit time left and so I will tell you not how you could use the library but how you could write your own, in your preferred, suitable language. I take it back: I don't have that much time. I will show just a few pitfalls you'll most likely to encounter when implementing such library, and one of the main ideas. To compensate for the practical bend of the paper the talk will be more theoretical: I'll be talking not about why the problem is useful but why it is fascinating.

As you can see, the structure of the talk is standard. The paper gives more than a glimpse of hope: it shows the whole solution, albeit with an informal reasoning why it works. The talk will concentrate on suffering. The over-arching idea is lexical scope. What is lexical scope?

Lexical scope

```
(define  (eta  f)  (lambda (x) (f  x)))

(define  test
  (lambda (x) (eta  (lambda (z) (+  z x)))))
```

At the beginning there was Lisp (or Scheme). Giving the following definition of `eta`, what does the `test` expression mean and do? We look at each expression in isolation: `eta` looks like what is says, that is, identity.

Lexical scope

```
(define (eta f) (lambda (x) (f x)))

(define test
 (lambda (x) (eta (lambda (z) (+ z x)) )))
```

Since η-expansion preserves the meaning of the function, we just erase `eta` and see that `test` is just the curried addition.

Lexical scope

Lexical (static) scope

```
(define  (eta  f)  (lambda (x) (f  x)))

(define  test
  (lambda (x) (eta  (lambda (z) (+  z x)))))
```

We have silently assumed lexical scope: just from the look of the code, of each function in isolation, we can deduce what variable is bound where. The modularity and ease of reasoning is the main benefit of lexical scope.

Lexical scope

Dynamic scope of x

```
(define  (eta  f)  (lambda (x) (f  x)))

(define  test-dyn
 (lambda (x) (eta  (lambda (z) (+  z x)))))
```

If x were dynamically bound ("special" in CL), the meaning of our expression would be different. We had to determine it by looking at the expression and eta together. We lose modularity.

Lexical scope in code generator

```
; eta:: (Code a → Code b) → Code (a→ b)
(define (eta f) `(lambda (x) ,(f 'x)))

(define test-code
 `(lambda (x) ,(eta (lambda (z) `(+ ,z x)))))
```

Suppose our goal is to generate code. Lisp has quotation and anti-quotation, which should make it easy. We just insert quotation and anti-quotation markers in obvious places.

Lexical scope in code generator

```
; eta:: (Code a → Code b) → Code (a→ b)
(define (eta f) `(lambda (x) ,(f 'x)))

(define test-code
 `(lambda (x) ,(eta (lambda (z) `(+ ,z x)))))

 ⇝(lambda (x) (lambda (x) (+ x x)))
```

test-code ≡ test-dyn

Lisp-like anti-quotation is not good enough

The result is not at all the curried addition: something different. Curiously, what we have generated has the same meaning as test-dyn, the dynamically scoped interpretation of our example. That's a good point to keep in mind, we come back to it.

Code-generating combinators

```
class SSym repr where
  intS :: Int -> repr Int
  addS :: repr Int -> repr Int -> repr Int
  appS :: repr (a -> b) -> (repr a -> repr b)
  lamS :: (repr a -> repr b) -> repr (a -> b)

eta :: (repr a -> repr b) -> repr (a -> b)
eta f = lamS (\x -> f x)

-- test :: repr (Int -> Int -> Int)
test = lamS (\x -> eta (\z -> addS z x))
```

So, Lisp anti-quotation is not enough for a code generator. What would be a better low-level interface for a code generator? Here is one, of code generating combinators. Here intS generates integer literal code; addS, given the code for two summands generates code for their sum, etc. The parameter repr represents the generated code. It is a constrained type variable, because there may be several representation of the code. That is, we write the code generator once, and by instantiating repr differently we generate code for different target languages, for example, Haskell or Scheme or Javascript. The code representation is indexed by type – so that we, so to speak, type check the generated code now. Therefore, it should compile without type errors.

Our running example takes the shown form, with the signatures inferred (we don't show the type class constraints). The combinator to generate the code of functions, lamS, uses higher-order abstract syntax (HOAS). Therefore, variables in the generated code like x are represented as Haskell variables. It's very convenient.

Implementation

```
data SExp = I Int | A String | L [SExp]

type VarCounter = Int
newtype S x = S{unS :: VarCounter → SExp}

instance SSym S where
  intS  = S ∘ const ∘ I
  addS (S x) (S y) = S $ \v → L [A "+", x v, y v]
  appS (S f) (S x) = S $ \v → L [f  v,  x v]
  lamS f = S $ \v →
      let  name = "x_" ++ show v −− gensym
           body = unS (f (S ∘ const ∘ A $ name)) (succ v)
                in  L [A "lambda", L [A name], body]
```

Here is one implementation for code generating combinators, one instantiation for repr. We generate S-expressions, essentially Lisp code. As any Lisp programmer knows, if we are to generate bindings, we have to use gensym. Here it is.

Paradise

```
eta :: (repr a → repr b) → repr (a→ b)
eta f = lamS (\x → f x)

-- test :: repr (Int → Int → Int)
test = lamS (\x → eta (\z → addS z x))

testS = unS test 0
 ⇝(lambda (x_0) (lambda (x_1) (+ x_1 x_0)))
```

- Generated code is well-typed
- Generated code is well-scoped
- Hygiene
  Lexical scope, modular reasoning

Our test generator then generates the shown code, for curried addition. There are no longer any surprises. So, generated code is well-typed. We reason modularly, locally about the generator. In test, lamS generates a binding for a variable x. The function eta receives the code that includes x as a free variable; that x will be bound by test's lamS. The function eta cannot bind any free variables in the code it receives as an argument. From the look of the generator we figure out, statically, what gets bound where. We thus figure out that every target code variable generated by lamS will be bound exactly by that lamS. The shall be no unbound variables in the complete generated code.

That's the best properties of the generator framework one can hope for. It all really works out. Paradise.

Effects needed

Generating matrix-matrix multiplication

```
lamS (\mA → lamS (\mB → lamS (\mC →
  partially_unrolled_loop    0 (nrows mA) unroll_factor    (\i  →
  ...
```

But we want effects. Here is a realistic example, of generating matrix-matrix multiplication code. It is incredible how much attention matrix-matrix multiplication gets in HPC. We aim at the optimal code and hence wish to partially unroll loops. The best unroll factor generally depends on the computer, on the amount of cache memory and its organization. So, we need to ask the OS, or the programmer.

Effects needed

Generating matrix-matrix multiplication

```
lamS (\mA → lamS (\mB → lamS (\mC →
    do
    unroll_factor    ← choose [2,3,4]
    partially_unrolled_loop     0 (nrows mA) unroll_factor   (\i  →
   ...
```

Or we just non-deterministically choose a good value out of likely candidates. Hence we generate several versions of the code, to benchmark on the computer in question and choose the fastest. This is indeed how ATLAS or SPIRAL, widely used in HPC, work. So, we need effects: IO or non-determinism. The paper shows more example of needing effects, e.g., loop interchange.

Effects needed

Generating matrix-matrix multiplication

```
lamS (\mA → lamS (\mB → lamS (\mC →
    do
    unroll_factor    ← choose [2,3,4]
    partially_unrolled_loop     0 (nrows mA) unroll_factor   (\i  →
   ...
```

Won't type!

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

Alas, this code won't type. Recall the type of lamS. The generator of the body of lamS must be pure: its return type is just code (repr b), without any monads.

Effects break it

Better lam combinator?

lamM :: **Monad** m ⇒ (repr a → m (repr b)) → m (repr (a→ b))

It seems we need a different generator of functions, lamM, of the shown signature. Alas, it has two problems.

Effects break it

Better lam combinator?

lamM :: **Monad** m ⇒ (repr a → m (repr b)) → m (repr (a→ b))

1. Can't express lamM in terms of lamS

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

First, we can't write lamM in terms of lamS, as we have talked about already.

Effects break it

Better lam combinator?

```
lamM :: Monad m => (repr a -> m (repr b)) -> m (repr (a-> b))
```

1. Can't express lamM in terms of lamS

   ```
   lamS :: (repr  a -> repr  b) -> repr  (a-> b)
   ```

2. Scope extrusion

   ```
   badM = do
     r <- newIORef (intS 0)
     lamM $ \x -> do
                    writeIORef r x
                    return (addS (intS 1) x)
     readIORef r
   ```

Second, it becomes possible to do mischief. What would this code generator produce? An unbound variable. This is a blatant violation of (lexical) scope. It is called scope extrusion: the generated variable extruded out (leaks out) of the scope of the generated binder. Of course noone deliberately writes this code. But very similar code is quite common. For example, we know memoization helps many computations. Memoization also helps code generation. What we memoize is code – often open code. There is the real danger we retrieve code from the memo table past the binder.

Everything breaks. We can prohibit putting open code in mutable cells, as some do. But we really need it, as I just explained with memoization.

Effects break it

Better lam combinator?

```
lamM :: Monad m => (repr a -> m (repr b)) -> m (repr (a-> b))
```

1. Can't express lamM in terms of lamS

   ```
   lamS :: (repr  a -> repr  b) -> repr  (a-> b)
   ```

2. Scope extrusion

   ```
   badM = do
     r <- newIORef (intS 0)
     lamM $ \x -> do
                    writeIORef r x
                    return (addS (intS 1) x)
     readIORef r
   ```

The first seems a minor problem, but it is an indication of bad things to come. It is the snake in our garden. If we can't use lamS, we have to introduce another binding generator in our framework, we have to break the abstraction of building variable names in a given target language. We saw that lamS provides lots of good properties. If we can't use lamS, we may have to give up on those properties.

Why do effects break it?

```
eta :: (repr a → repr b) → repr (a→ b)
eta f = lamS (\x → f x)

-- test :: repr (Int → Int → Int)
test = lamS (\x → eta (\z → addS z x))
```

Uneffectful code generators

- Haskell variables for target code variables
- Haskell *dynamic* environment (of lamS invocations) for target *type* environment
- Any open code is always within the region of its lamS

Let's look again at our example: eta receives the open code in this program, but its type is just repr Int. Closed code, intS 1, would have the same type.

In the example on the slide it didn't matter: that's why the uneffectful life was so wonderful. We were assured that all free variables will be bound, by their intended binders (by the same lamS that introduced the variables), because of the strict region discipline of lamS. Any open code is manipulated only within the region of lamS that introduced the variables.

The effects break this correspondence, which causes all the problems.

Being explicit

(open) code representation: $\gamma \to \mathsf{repr\ a}$

```
intH :: Int  → (γ → repr  Int )
intH = const ∘ intS

addH :: (γ → repr  Int ) → (γ → repr  Int ) → (γ → repr  Int )
addH x y = \γ → addS (x γ) (y γ)

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

We've got to differentiate open and closed code in types, to give the type checker enough information to detect scope extrusion. We make the type environment of the target code explicit. So, the potentially open code now has the following representation, where γ is the environment for the free target code variables that may be in the code. The signature for lamH shows how we extend the environment when we introduce a new variable in the generated code.

All these new functions etc. are expressed entirely in terms of intS, addS, etc. That's a good thing: we maintain the abstraction provided by repr and especially of lamS, of the actual process of generating a variable in the target code. These new functions are still uneffectful. But now they are generalize to effects.

Being explicit and adding effects

Effectful generator of open code: $\mathsf{m}(\gamma \to \mathsf{repr\ a})$

```
intE :: Monad m ⇒ Int → m (γ → repr Int)
intE = return ∘ const ∘ intS

addE :: Monad m ⇒
          m (γ → repr Int) → m (γ → repr Int) → m (γ → repr Int)
addE e1 e2 = liftM2 (\x y γ → addS (x γ) (y γ)) e1 e2

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

Ugly! Desperately seeking abstraction

Now we introduce effects. We can generate open code and have effects while doing this. lamE is expressed entirely in terms of lamS. It is clear from this code how ugly it is. All these explicit γs everywhere. Exposing the representation of the environment makes it ripe for abuse: a programmer can write a function to reshuffle the environment. We need to hide the environment plumbing. We need abstraction.

The first hint comes from the type of lamE: we only need m to be a functor.

Let's look at this type: $\mathsf{m}(\gamma \to \mathsf{repr}\ \mathsf{Int})$. The type in parentheses looks like the Reader Monad. So, the whole type looks like a composition of m and the reader monad. Alas, the composition of two monads is not a monad in general, so this leads us nowhere. But! Any monad is an applicative, and applicatives compose!

Applicative

Representing a computation that produces the value of the type α and may have an effect

	Monads	**Applicatives**
Representation type	$m\ \alpha$	$i\ \alpha$
General Introduction	return : $\alpha \to m\alpha$	pure : $\alpha \to i\alpha$
Composing principle	$m\alpha \to (\alpha \to m\beta) \to m\beta$	$i(\alpha \to \beta) \to i\alpha \to i\beta$
	let-binding with 'side-effects'	Function application with 'side-effects'

What are applicatives? You might have heard of monads, an obscure philosophical concept borrowed as a joke into Category theory and rising to prominence through hardly countable monad tutorials. Applicative is a simpler version of monads. For one, 'applicative' has been in English language longer (by about 30 years, according to OED: OED quotes 1607 for applicative).

Both applicatives and monads represent a computation that produces the value of the type α and may have an effect.

Applicative

Representing a computation that produces the value of the type α and may have an effect

	Monads	**Applicatives**
Representation type	$m\ \alpha$	$i\ \alpha$
General Introduction	$\text{return} : \alpha \to m\alpha$	$\text{pure} : \alpha \to i\alpha$
Composing principle	$m\alpha \to (\alpha \to m\beta) \to m\beta$	$i(\alpha \to \beta) \to i\alpha \to i\beta$
	let-binding with 'side-effects'	Function application with 'side-effects'

- All monads are applicatives, but not vice versa
- Applicatives compose, monads generally not

A glimpse of hope

Effectful generator of open code: i (repr a)

```
int :: Applicative i ⇒ Int → i (repr Int)
int = pure ∘ intS

add :: Applicative i ⇒
       i (repr Int) → i (repr Int) → i (repr Int)
add x y = addS <$> x <*> y

lam :: Functor i ⇒
       (∀ j. Applicative j ⇒ j (repr a) → (i ∘ j) (repr b)) →
       i (repr (a→ b))
lam f = fmap lamS (unJ (f var))
 where var = \x → x          -- j is the Reader applicative
```

Now, i (repr a) is the type of the generator that produces potentially open code and has some effects. Look how pretty it is. Plumbing is hidden, especially in generators like add who could care less what environment the code is in. Everything is expressed in terms of intS, lamS, etc. – and in a very simple way. Since a composition of two applicatives is an applicative, we hide the structure of the environment and even its length. The higher-rank type of lam prevents any mix-up and any attempts to 'permute' the environment.

A glimpse of hope, close-up

```
newtype (i ∘ j) a = J{unJ:: i (j a)}
liftJ  :: (Applicative m, Applicative i) ⇒ m a → (m ∘ i) a
liftJ  = J ∘ fmap pure

var :: Applicative m ⇒ i (repr a) → (m ∘ i) (repr a)
var = J ∘ pure
```

Here is how applicative composition is defined. We also need weakening, liftJ.

A glimpse of hope, close-up

```
eta_eff :: (Applicative i, m ~ (IO ∘ i)) ⇒
  (∀ j. Applicative j ⇒ (m ∘ j) (repr a) → (m ∘ j) (repr b)) →
  m (repr (a→ b))
eta_eff f = lam (\x → (liftJ . liftJ $ putStrLn "in␣eta") *> f (var x))

test_eff :: IO (repr (Int → Int → Int))
test_eff = lam (\x → liftJ (putStrLn "in␣test") *>
                     eta_eff (\z → add z (liftJ (var x))))
```

Here's our running example again, but with effects, tracing – in the time-honored tradition of printf debugging. It doesn't look too bad: we have to add weakening (liftJ) here and there, but overall it is not too bad. The weakening can be automated in many cases.

A glimpse of hope, no scope extrusion

```
bad :: IO (repr (Int → Int))
bad = do
  r ← newIORef ⊥
  lam $ \x → liftJ (writeIORef r x) *>
                   add (int 1) (var x)
```

```
Couldn't match expected type 'a0' with actual type 'j (repr Int)'
      because type variable 'j' would escape its scope
    This (rigid, skolem) type variable is bound by
      a type expected by the context:
        Applicative j ⇒ j (repr Int) → (∘) IO j (repr Int)
    The following variables have types that mention a0
      r :: IORef a0 (bound at /home/oleg/temp/beyond-talk.hs:202:3)
    In the second argument of `($)', namely
      '\ x → liftJ (writeIORef r x) *> add (int 1) (var x)'
```

The bad example causes the error even if we don't retrieve the stashed away free variable. The mere act of stashing it away is the problem. The error message is quite precise and informative. It directly tells us that something was trying to escape its scope.

Conclusions

- Code generation complicates the notion of scope
- Effects complicate even more

Code generation: think Applicative

More in the paper

- the real framework of code generation
- bigger and better examples
- control effects and let-insertion
- new applicative CPS hierarchy
- informal justifications, lexical scope and α-convertibility

`http://okmij.org/ftp/tagless-final/TaglessStaged/`

We come again to conclusions. We've been talking about how much code generation complicates the notion of variable scope, and how much effects mess it all up.

The main message is that if you think of code generation, think applicative, not a monad.

The paper also proposes 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. I did not say much about let-insertion and loop-exchange, and nothing at all about that CPS hierarchy. The paper does, and even gives an exercise that you might enjoy doing.