Program Generation With Class

Peter Thiemann and Michael Sperber

Wilhelm-Schickard-Institut, Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
{thiemann,sperber}@informatik.uni-tuebingen.de

Abstract. We have implemented a program generation library for polymorphically typed functional languages with lazy evaluation. The library combinators perform program generation by *partial evaluation*, a technique which allows the generation of highly-customized and efficient specialized output programs from general, parameterized input programs. Previously implemented program generation libraries for polymorphically typed languages have either required dynamic typing or been in the context of specially designed program generation languages. In contrast, our library has been implemented in Gofer, a widely available functional language. Moreover, we exploit multi-parameter constructor classes which are part of Gofer's type system to construct program generators that are *binding-time polymorphic*. An appropriate polymorphic binding-time analysis can provide the necessary type annotations to specify these properties. However, we designed and implemented a minor extension of Gofer's type reconstruction mechanism that can automatically infer binding times.

Partial evaluation [2, 9] is an automatic program transformation technique which improves programs when given part of their input. Typically, a source program `p` has two inputs `ins` and `ind` where `ins` is available early, *statically*, and `ind` is available late, *dynamically*. A partial evaluator `spec` specializes `p` with respect to `ins` such that

$$[\![p]\!]\ \texttt{ins}\ \texttt{ind} = [\![[\![\texttt{spec}]\!]\ \texttt{p}\ \texttt{ins}]\!]\ \texttt{ind}.$$

The intention is that the specialized (or *residual*) program $[\![\texttt{spec}]\!]\ \texttt{p}\ \texttt{ins}$ computes the result more efficiently than the source program applied to the entire input.

Offline partial evaluation stages specialization into a *binding-time analysis* and a *specialization* phase. The binding-time analysis transforms `p` into `p-ann`, given the information that `ins` is statically known—regardless of the particular value of `ins`. It marks each part of `p` as either *static* (executable at specialization time) or *dynamic* (executable at run time; the specializer must generate code for it). Consequently, for offline partial evaluation, `spec` is an interpreter of annotated programs.

In addition to being an aggressive optimization technique, offline partial evaluation is a rather general approach to program generation: The dynamic parts of a program function as templates for the generated programs with the specializer filling in the static parts at program generation time.

One way to implement partial evaluation is to transform `p-ann` into a *generating extension* `p-gen`. The generating extension is a specializer customized to specializing `p` with respect to its static input:

$$[\![\texttt{p}]\!]\ \texttt{ins}\ \texttt{ind} = [\![[\![\texttt{p-gen}]\!]\ \texttt{ins}]\!]\ \texttt{ind}.$$

Either self-application of a partial evaluator or a hand-written *program generator generator* (PGG, cogen) [1,10] can generate such a generating extension. The PGG approach is attractive because of two reasons: Static constructs can run directly rather than having to go through interpretation. Also, the generation extension can exploit the presence of a library of specialization combinators [5,15], each of which defines the behavior of one specific annotated language construct. Effectively, a specialization combinator library allows the annotated program to *be* the generating extension.

Previous specialization combinator libraries [5, 15] are implemented in languages with dynamic typing. A major drawback of existing libraries is that their combinators are binding-time-monovariant: each construct carries one fixed annotation assigned by the binding-time analysis. In contrast, recent developments in binding-time analysis have given rise to *polymorphic binding-time analysis* [4] which does not have that restriction. There is an ad-hoc implementation of a PGG using an extension of that binding-time analysis [3].

To summarize, our contributions are the following:

1. We introduce simple combinators for binding-time-monovariant generating extensions, using standard ML-typed programs.
2. We show how to exploit Gofer's multi-parameter constructor classes [7] to implement binding-time-polyvariant generating extensions.
3. We demonstrate how to extend Gofer's type system to also perform automatic binding-time analysis as a by-product of type reconstruction.

1 Binding-Time-Monovariant Program Generation

ML-style parametric polymorphism is sufficient to express binding-time-monovariant program generation: Each expression in the generating extension only works for one specific set of binding times for its parameters. We motivate the combinators needed with a simple example and show how to implement the remaining combinators required for general program generation.

1.1 Motivation

Consider the `power` function, written in Gofer:

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

With a fixed `n` at, say, `2`, a generating extension based on `power` should come up with `power2`.

```
power2 :: Int -> Int
power2 x = x*x*1
```

1.2 Simple Program Generation

The construction of an appropriate generator requires a binding-time analysis. Since `n` is static the test `n==0` and hence the conditional are also static. The same holds for the subtraction `n-1`. Everything else is dynamic. Therefore, a generating extension would look like this:

```
power_gen_SD :: Int -> Code -> Code
power_gen_SD n x = if n==0 then liftD 1
                           else multD x (power_gen_SD (n-1) x)
```

Here, `liftD 1` converts the generation-time value `1` to an expression whose value is 1 and `multD` constructs the code for a multiplication. Given suitable definitions for `liftD` and `multD` (shown in Section 1.4) the expression `power_gen_SD 2 (MkVar "x")` evaluates to a representation of `x*x*1`.

1.3 Representing Code

Program generation requires a code representation. The `Code` datatype represents a small applied lambda calculus:

```
data Code =
   MkInt Int                -- integer constants
 | MkVar String             -- variables
 | MkLam String Code        -- abstraction
 | MkApp Code Code          -- application
 | MkIf Code Code Code      -- conditional
```

Names like `+`, `*`, etc. are implicitly bound to appropriate definitions. The function `multiApp` simplifies the construction of code generating functions.

```
multiApp :: [Code] -> Code
multiApp = foldl1 MkApp
```

The paper uses standard Gofer syntax to represent generated programs.

Code generation will ultimately need to generate fresh names in the output code. Hence, the specialization combinators will return instances of a code generation monad rather than just a `Code` object. The code generation monad `CodeGen a` is a supply for fresh names as an instance of a state transformer with unit and bind operations `resultC` and `bindC`:

```
data CodeGen a = CodeGen (NameSupply -> (a, NameSupply))
type NameSupply = Int
resultC :: c -> CodeGen c
resultC c = CodeGen (\n -> (c, n))
bindC :: CodeGen a -> (a -> CodeGen b) -> CodeGen b
bindC (CodeGen m) f = CodeGen ((\(c, n) ->
                              let CodeGen m' = f c in m' n) . m)
runC :: CodeGen a -> a
runC (CodeGen st) = fst (st 0)
newVar :: CodeGen String
newVar = CodeGen ((\n -> ('v':show n, n)) . (+1))
```

The `newVar` operation generates a fresh variable name. The function `runC` runs a code-generating computation and returns the resulting code. The transition to the `CodeGen` monad incurs that all dynamic constructs need to return a `CodeGen Code` instead of `Code`.

```
mkInt = resultC . MkInt
mkVar = resultC . MkVar
mkLam v e = resultC (MkLam v e)
mkApp e1 e2 = resultC (MkApp e1 e2)
mkIf e1 e2 e3 = resultC (MkIf e1 e2 e3)
multiAppC = resultC . multiApp
```

The appropriate instance declarations make `CodeGen` into a proper monad so that `do` notation works for it:

```
instance Functor CodeGen where
  map f (CodeGen t) = CodeGen (\n -> let (x, n') = t n in (f x, n'))

instance Monad CodeGen where
  result = resultC
  bind = bindC
```

1.4 Monovariant Specialization Combinators

The `power_gen_SD` generator in Section 1.1 requires definitions for `liftD` which constructs a residual constant from an integer, and `multD` which constructs a residual multiplication:

```
liftD :: Int -> CodeGen Code
liftD = mkInt
multD :: CodeGen Code -> CodeGen Code -> CodeGen Code
multD x y = do { c1 <- x; y <- c2; multiAppC [MkVar "*", c1, c2] }
```

More advanced program generation requires a dynamic conditional operator `ifD`:

```
ifD :: CodeGen Code -> CodeGen Code -> CodeGen Code -> CodeGen Code
ifD i t e = do { c0 <- i; c1 <- t; c2 <- e; mkIf c0 c1 c2 }
```

The `ifD` combinator allows the formulation of dynamic recursion. However, its naive use in a realistic settings (using Gofer recursion to model dynamic recursion) immediately leads to non-termination. Hence, dynamic recursion requires a `fixD` operator for dynamic fixpoints, and, for good measure, `lambdaD` for dynamic abstraction.

`fixD` and `lambdaD` make use of higher-order abstract syntax [11, 12, 15] which delegates all questions of binding and scoping to the Gofer interpreter. Without higher-order abstract syntax, program generators would have to encode all variable references.

```
lambdaD f = do { v <- newVar; b <- f (mkVar v); mkLam v b }
applyD f a = do { f0 <- f; a0 <- a; mkApp f0 a0 }
fixD f = do { v <- newVar; b <- f (mkVar v);
              mkApp (MkVar "fix") (MkLam v b)
            }
```

The functions `lambdaD` and `fixD` acquire fresh variable names before constructing the respective bodies. The construction happens by applying the function `f` to a code generator which constructs a residual variable reference to the fresh variable. Finally, the construct itself is generated.

Now `runC (power_gen_DS (MkVar "n") 7)` evaluates to:

```
fix (\v1 -> \v2 -> if v2==0 then 1 else 7*v1 (v2-1))
```

2 Polyvariant Specialization Combinators

The combinators that we have developed so far are monovariant with respect to their binding-time properties. This means that source functions which need to generate code for several different binding times for their arguments require multiple generators for the same function. Polymorphic binding-time analysis [4,6] addresses this shortcoming. It attaches a symbolic binding-time value to each expression. We use constructor classes to propagate the necessary binding-time information at generation time.

The goal is to use a single program generator function `power_gen_XX` which can be used at all of the following types: `Int` $\to$ `Int` $\to$ `Int`, `Int` $\to$ `Code` $\to$ `Code`, `Code` $\to$ `Int` $\to$ `Code`, and `Code` $\to$ `Code` $\to$ `Code`.

The additional requirements for polyvariant specialization combinators are:

- an expression `lift 1` in the program must stand both for an integer and for code,
- the multiplication operator `*` must stand once for multiplication and once for a code constructor,
- a `cond` function for conditionals must be able to accept code as a condition,
- an equality function that can generate code as well as a boolean result.

2.1 Lifting

A type class `Liftable` with instances for `Int` and `CodeGen Code`. This class requires multiple parameters, and is therefore specific to Gofer.

```
class Liftable a b where
  lift :: b -> a
```

Obviously, the identity function covers all cases where `lift x` must stand for `x`. Integers lift to code by virtue of the appropriate code constructor.

```
instance Liftable a a where
  lift = id
instance Liftable (CodeGen Code) Int where
  lift = mkInt
```

2.2 Primitive Operations

For primitive operations on numbers, code generators need to be an instance of the type class `Num`. Two auxiliary functions define unary and binary operators.

```
unop  name x   = do { c1 <- x; multiAppC [MkVar name, c1] }
binop name x y = do { c1 <- x; c2 <- y; multiAppC [MkVar name, c1, c2] }

instance Num (CodeGen Code) where
  x + y = binop "(+)" x y
  x - y = binop "(-)" x y
  x * y = binop "(*)" x y
  x / y = binop "(/)" x y
  negate x = unop "negate" x
  fromInteger i = mkInt i
```

2.3 Conditional

Like the definition `Liftable` of `lift`, there is a type class `Conditional` for the conditional. `Conditional a b` denotes that there is a conditional `cond` where `a` is the result type of the test, and `b` is the result type of the conditional itself:

```
class Conditional a b where
  cond :: a -> b -> b -> b
```

The first instance describes the static conditional which is just an `if` expression.

```
instance Conditional Bool b where
  cond i t e = if i then t else e
```

The second instance describes the dynamic conditional which generates code from code:

```
instance Conditional (CodeGen Code) (CodeGen Code) where
  cond = ifD
```

2.4 Equality

Unlike the case for `Num` above, the standard class `Eq` is unsuitable for characterizing comparisons because the result type of `Eq` comparisons is always `Bool`. However, in the context of code generation, it may have to return code. Again, a multi-parameter type class is required.

```
class Equality a b where
  eq :: a -> a -> b

instance Eq a => Equality a Bool where
  eq x y = x == y
instance Equality (CodeGen Code) (CodeGen Code) where
  eq x y = binop "(==)" x y
```

2.5 Functions

The definition of a class for functions `lambda` and `apply` needs to abstract from the function type constructor in order to give sensible typings in the standard function case. In this case a multi-parameter constructor class is necessary.

```
class Function f a b where
  apply  :: f a b -> a -> b
  lambda :: (a -> b) -> f a b
```

The instance that covers standard functions implements the `apply` function by application and `lambda` by the identity.

```
instance Function (->) a b where
  apply f x = f x
  lambda f = f
```

The instance for code is slightly more problematic since it requires a *binary* type constructor for code:

```
data CodeGen2 a b = MkCodeGen2 (CodeGen Code)
fromCodeGen2 :: CodeGen2 a b -> CodeGen Code
fromCodeGen2 (MkCodeGen2 x) = x

instance Function CodeGen2 (CodeGen Code) (CodeGen Code) where
  apply f a = do { x1 <- fromCodeGen2 f; x2 <- a; mkApp x1 x2 }
  lambda f  = MkCodeGen2 (
                do s <- newVar
                   e <- f (mkVar s)
                   mkLam s e
              )
```

2.6 Fixpoints

Using this encoding for functions a fixpoint operator becomes feasible:

```
class Function f a a => Fix f a where
  fix :: f a a -> a
  fix g = apply g (fix g)

instance Fix (->) a

instance Fix CodeGen2 (CodeGen Code) where
  fix g = do { c <- fromCodeGen2 g;  mkApp (MkVar "fix") c }
```

2.7 Synthesis

With the now complete combinator library, a maximally polymorphic generator for `power` is straightforward:

```
power_gen_XX n x = apply (fix (\f -> lambda (\n ->
                   cond (eq n (lift 0))
                        (lift 1)
                        ((lift x) * (apply f (n - (lift 1))))))) n
```

It has the following type which Gofer infers automatically:

```
power_gen_XX :: (Fix (->) (f n r), Conditional b r, Num r,
                 Function f n r, Num n, Liftable r x, Liftable r Int,
                 Equality n b, Liftable n Int) => n -> x -> r
```

Gofer can use `power_gen_XX` at all the types advertised above. Hence, it can evaluate all of the following expressions.

```
power_gen_XX 3 2
  ⟹  8
runC (power_gen_XX 2 (mkVar "x"))
  ⟹ x*x*1
runC (power_gen_XX (mkVar "n") 4)
  ⟹ fix \f -> \n -> if n==0 then 1 else 4*f (n-1)
runC (power_gen_XX (mkVar "n") (mkVar "x"))
  ⟹ fix \f -> \n -> \x -> if n==0 then 1 else x*f (n-1)
```

Our approach to overloading the function type constructor generalizes to overloading product and sum construction. The actual class and instance declarations are very similar to the declarations for class `Function` and have therefore been omitted from the paper.

3 Improvement

Unfortunately, the above combinators often lead to program generators for which the Gofer type inference engine cannot infer a type fully automatically. Specifically the multi-parameter class `Function` is a major source of ambiguity. For example, the expression

```
  identity = lambda (\z -> z)
```

has type `identity :: Function f a a => f a a`. If we `apply identity` to anything, the type checker can probably resolve `a` to `Int` or `CodeGen Code`. However, it is not able to determine the correct instance to use since it cannot infer `f`. Conversely, the overloading in the expression `runC (fromCode2 identity)` cannot be resolved, either. Here, `f` is instantiated to `CodeGen2`, but it is impossible to determine an instantiation for `a`.

We use the technique of *improvement* to make type inference more effective. Improvement [8] allows the type checker to instantiate type variables in contexts if there is only instance that fits the already instantiated type variables. The goal is to perform subsequent context simplification. In our second case, where `f` is instantiated to `CodeGen2` the only remaining possibility for `a` is `(CodeGen Code)`. Hence, improvement solves the second problem.

The first problem is not always amenable to improvement. If `a` is instantiated to `Int`, say, then the only possible instantiation for `f` is `->`. If `a` is `CodeGen Code` then both instantiations of `f` (function or code) are possible and improvement cannot help. To deal with this situation, we have enhanced the type checker by a method which is standard practice in binding-time analysis. First, we use improvement as far as possible to propagate the dynamic binding time. If that is no longer possible we start instantiating the occurrences of `f` in a `Function a b` context to `->` until all type variables of higher kind are instantiated. We proceed accordingly for other similar contexts (for pairs, sums, etc).

We have modified the type checker of the Gofer interpreter to perform improvement with a bias to binding-time analysis. This modified version infers the type of the expression

```
runC (fromCode2 identity)
```

to be `Code`. It discovers that `f` in the predicate `Function f a a` is instantiated to `CodeGen2` and therefore instantiates `a` to `CodeGen Code` (as specified in the instance declaration for `Function`).

We have yet to perform experiments on larger programs to assess the effectiveness of the technique in more realistic settings.

4 Related Work

Writing program generator generators directly has proven a viable approach to partial evaluation [1, 5, 10, 15]. All of these works describe binding-time-monovariant PGGs. Also, the library-based proposals [5, 15] both depend on the dynamic typing discipline provided by the implementation language Scheme. Nelson, Sheard, and Taha [13, 14] propose programming languages specifically tailored to and program generation. The programmer must explicitly provide binding-time annotations; there is no automatic binding-time analysis. In these languages, generated programs are always type-correct. All the above approaches deal exclusively with strict languages. In fact, we are not aware of any specializer for a lazy language which our approach provides for free.

Polymorphic binding-time analysis [4] results in annotated programs with symbolic annotations. A corresponding program generator propagates actual binding-time values at generation time [3]. It cannot benefit from using the natural representation of static values (one important advantage of the PGG approach for typed languages) since the type of such a generator, when written using just ML-style parametric polymorphism, cannot express the dependency of the type on the actual binding-time values (if the binding-time parameter has value "dynamic" then the type is code, otherwise if the binding-time parameter has value "static" then the type is integer, for example). The polymorphic binding-time analysis [4] also uses qualified types in order to express binding-time constraints. However, the information inherent in these qualified types is not carried over to the specialization phase.

5 Conclusion

The PGG approach relying on denotational implementations of constructs of an annotated language seamlessly carries over to polymorphically typed languages. The overloading mechanism provided by Gofer with its multi-parameter constructor classes is essential to construct the denotational implementations. This fact provides further evidence that multi-parameter constructor classes should find their way into the Haskell standard.

Initial experience with the implementation shows that this approach to polymorphic and polyvariant program generation is feasible. We have yet to perform experiments with larger programs to assess the scalability.

References

1. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '94)*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.
2. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Symposium on Principles of Programming Languages '93*, pages 493–501, Charleston, January 1993. ACM.
3. Dirk Dussart, Rogardt Heldal, and John Hughes. Module-sensitive program specialisation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, USA, June 1997. ACM Press.
4. Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag. LNCS 983.
5. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Language Implementation and Logic Programming 1995*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.
6. Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proc. 5th European Symposium on Programming*, pages 287–301, Edinburgh, UK, April 1994. Springer-Verlag. LNCS 788.
7. Mark P. Jones. Partial evaluation for dictionary-free overloading. In Peter Sestoft and Harald Søndergaard, editors, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '94*, pages 107–118, Orlando, Fla., June 1994. ACM.
8. Mark P. Jones. Simplifying and improving qualified types. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 160–169, La Jolla, CA, June 1995. ACM Press, New York.
9. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
10. John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.
11. Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.
12. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. Conference on Programming Language Design and Implementation '88*, pages 199–208, Atlanta, July 1988. ACM.
13. Tim Sheard and Neal Nelson. Type safe abstractions using program generators. Technical Report 95-013, Oregon Graduate Institute of Science and Technology, PO Box 91000, Portland, OR 97291-1000 USA, July 1995.
14. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Charles Consel, editor, *Proc. Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, Amsterdam, The Netherlands, June 1997. ACM Press.
15. Peter Thiemann. Cogen in six lines. In *International Conference on Functional Programming '96*, pages 180–189, Philadelphia, May 1996. ACM.