Generating Code with Polymorphic let

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Abstract

We practically solve the long-standing vexing problem of generating code with polymorphic let. For polymorphic functions, we rely on a conjecture whose proof will require a re-investigation of the seemingly long-settled value restriction.

1. Summary

Quotation is a time-honored way of embedding languages and generating code. The simplest way to support it is by the source-to-source translation of quoted expressions to code-generating combinators. For example, MetaOCaml could be implemented as a pre-processor to the ordinary OCaml. However simple, the approach is surprising productive and extensible, as Lightweight Modular Staging (LMS) in Scala has demonstrated. However, quotation of polymorphic let is the show-stopper, see §3. In the common translation, a quoted binding becomes a metalanguage lambda-binding, which is monomorphic. Polymorphic lambda-bindings, however we emulate them, require type annotations, which precludes the source-to-source translation.

We present a new translation that maps a quoted let-binding to a metalanguage let-binding, which is generalizable. It works with the current OCaml, demonstrating the benefits of the relaxed value restriction for the other ML languages. It solves the problem in practice, and almost in theory: It relies on a conjecture, whose proof is a research program, promising to re-open the old value-restriction wounds and heal them.

2. Introduction

One of the elegant ways to write programs to generate programs is quotation, made popular by Lisp. For example, MetaOCaml lets us quote any OCaml expression, by enclosing it within .< and >. brackets:

```
let c = .<1 + 2>.
⇝ val c : int code = .<1 + 2>.
```

The quoted expression, representing the generated code, is a value of the type αcode, and can bound, passed around, printed – as well as saved to file and then compiled, or evaluated. An expression that evaluates to a code value can be sliced-in (or, unquoted, in Lisp terminology) into a bigger quotation:

```
let cb = .<fun x → .~c + x>.
⇝ val cb : (int → int) code = .<fun x_1 → (1 + 2) + x_1>.
```

The splice is marked with .~, which is called an escape. When generating the typed language code, it is also natural to require that the produced code is type-correct by construction.

The merits of code generation with the typed and hygienic quotation have been already well-explained [7]. This paper is about implementing such quotation.

The simplest approach to adding quotation to an existing functional language is to write a pre-processor that translates quoted expressions into ordinary expressions, which use pre-defined functions that build and combine code values, so-called code combinators.

```
module type Code = sig
  type +α cod
  val int : int → int cod
  val add: int cod → int cod → int cod
  val mul: int cod → int cod → int cod
  val pair : α cod → β cod → (α * β) cod
  val lam: (α cod → β cod) → (α→β) cod
  val app: (α→β) cod →(α cod → β cod)
  val nil : α list cod
  val cons: α cod → α list cod → α list cod
end
```

The sample Code specifies a collection of typed combinators to generate code for a simple functional language: int 1 builds the literal 1, add combines two pieces of code into the addition expression, etc. The combinator lam builds the code of a function; its argument is an OCaml function that returns the code for the body upon receiving the code for the bound variable. A quoted expression like

```
fun x → .<fun y → .~x * y + 1>.
```

is then pre-processed into the expression that uses the code combinators:

```
fun x → lam (fun y → (add (mul x y) (int 1)))
```

This is plain OCaml, to be compiled as usual. The rules of the translation should be clear from the above example, and are elided to save space (see [1, §3] for formal presentation).

This implementation scheme for quotation by pre-processing them away has many benefits. It is a source-to-source translation, which can be done by a macro-processor such as camlp4 or a stand-alone pre-processor. The rest of the language system (type-checking, code-generation, standard and user-defined libraries) is used as it is. We notice from the translation of our example that the binding y in the quoted code becomes the lambda-binding in the translated code. Coupled with the appropriate implementation of the lam combinator, this property makes it easy to ensure hygiene. Finally, the translation is type-preserving: a well-typed quoted expression is translated into a well-typed OCaml expression. If we also ensure that individual combinators produce well-typed code (see below), any typing errors in the quoted code manifest themselves as OCaml type errors emitted when type-checking the translated expression. Absent such errors, the quoted expression, and hence the generated code, are type-correct.

This simple approach works surprisingly well: Scala's Lightweight Modular Staging (LMS) is based on similar ideas [6]. Scheme's implementation of quasi-quote is also quite similar; only it pays no attention to quoted bindings and is hence non-hygienic.

The following figure shows two implementations of the Code signature. CodeString combinators generate ML code as text strings. CodeReal is a meta-circular interpreter, representing a code expression as an OCaml thunk. It is easy to see by inspection and simple induction on the code that the two implementations correspond: the behavior of the code produced by

CodeString is the same as the behavior of running the thunk of CodeReal. The OCaml type-checker ensures that any thunk built by CodeReal combinators is well-typed; therefore, it "will not go wrong" thanks to the soundness of OCaml. Hence the code generated by CodeString will also be well-typed and will not go wrong either. The existence of the CodeReal implementation is thus crucial to assuring the soundness of code generation[1].

```
module CodeString = struct
 type α cod = string
 let int         = string_of_int
 let add x y     = paren x ^ "␣+␣" ^ paren y
 let lam body = let var = gensym () in
     "fun␣" ^ var ^ "␣→␣" ^ paren (body var)
 ...
end

module CodeReal = struct
 type α cod = unit → α
 let int x       = fun () → x
 let add x y     = fun () → x () + y ()
 let lam body = fun () x → body (fun () → x) ()
 ...
end
```

3. Let-polymorphism problem

The implementation of quotation by translation into code combinators becomes more complex as we add to the target language more special forms such as loops, pattern matching, type annotations, etc. They pose problems, but they can and have been dealt with, e.g., in [6]. The show-stoppers, until now, were quotations that contain polymorphic let-bindings, such as

```
(1)  .<let l = [] in (1:: l, (1,2):: l)>.
(2)  .<let f = fun x → x in (f 1, f (2,3))>.
```

To generate let-statements we add to Code the combinator that combines app and lam:

```
val let_ : α cod → (α cod →β cod) → β cod
```

so that (1) translates to

```
let_ nil (fun l → pair (cons (int 1) l) ...))
```

Recall that the translation maps a binding in the quoted code to a metalanguage lambda-binding. This exactly is the problem: lambda bindings in ML, unlike let-bindings, are not generalizable. First-class polymorphism, if available, does not help since it requires type annotations; hence our translation to code combinators cannot be a source-to-source translation, done before type checking.

MetaOCaml does support polymorphic let in quotations, at the price of doing the translation to combinators after the type-checking, which requires painful modifications to the OCaml type-checker [5].

4. New translation of quoted let-expressions

We now present a new translation for quoted let-expressions, which works even with polymorphic let bindings. We introduce another combinator for generating let-expressions, along with an auxiliary new_scope:

```
type α scope
val new_scope: ('w scope → 'w cod) → 'w cod
val genlet : 'w scope → α cod → α cod
```

They are intended to be used as

```
new_scope (fun p →
  lam (fun x→ add x (genlet p (add (int 1) (int 2)))))
```

which results in the generated **let** t = 1 + 2 **in fun** x →x + t. In other words, genlet p e inserts, at the place marked by the corresponding new_scope, a **let** statement that binds e to a fresh variable, and returns the code that contains the name of that variable.

The combinators genlet and new_scope are introduced for the sake of translating quoted expressions rather than for end users. More formally, the translation of let-expressions has the form

```
⌈ let x = e1 in e2 ⌉ ↦
new_scope (fun p → let x = genlet p ⌈ e1 ⌉ in ⌈ e2 ⌉)
```

For instance, example (1) is translated as

```
new_scope (fun p →
  let l = genlet p nil in
  pair (cons (int 1) l) (cons (pair (int 1) (int 2)) l))
```

The quoted let-binding becomes the meta-language let-binding, which is generalizable. It is in fact generalized, even if l is bound to a non-value: the relaxed value restriction [2] in OCaml also generalizes expressions provided the quantified type variable occurs in the covariant position, which it does in the case of α list code.

The seemingly magical genlet is implementable: it is the let-insertion well-known in the partial evaluation community. We wrote it with the delimited control library delimcc, as described in [5], for both CodeString, and, more importantly for soundness, CodeReal code combinators. The latter is shown below. Although it is really short:

```
module CodeLetReal = struct
  include CodeReal
  open Delimcc
  type α scope = α cod prompt
  let new_scope body =
   let p = new_prompt () in
   push_prompt p (fun () → body p)
  let genlet p e =
   shift0 p (fun k → let t = e () in k (fun () → t))
end
```

its explanation is not. Here we only mention that the type α prompt and the delimited control operators push_prompt and shift0 are provided by the delimcc library; see [5] for details. In the end, we indeed generate the desired code.

The genlet is so powerful that it easily moves bound variables

```
new_scope (fun p →
  lam (fun x→ add x (genlet p (add x (int 2)))))
```

resulting in the generated code **let** t = x + 2 **in fun** x →x + t with an unbound variable x. One may prevent such undesirable behavior either with a complex type system (whose glimpse can be caught in [4]) or with a dynamic test, as implemented in MetaOCaml [5]. In our case however genlet appears in the code solely as the result of the translation of a quoted expression. Fortunately, our translation of let-expressions puts new_scope "right above" genlet, never letting them be separated by a lam binding. Therefore, in this case using delimited control, which underlies genlet, is safe (for proofs, see [3]).

4.1 Value restriction at the whole new level

Alas, this approach stumbles for the most interesting and usual case of polymorphic function bindings, like (2). Its translation includes

```
let f = genlet p (lam "x" (fun x → x)) ...
```

Here, the genlet expression has the type $(\alpha \to \alpha)$ code, which is not covariant in α. Generalizing expressions of such types is unsound: otherwise, we will have to accept the following clearly undesirable code

```
(3) .<let f = let r = ref [] in
                fun x → rset r [x]
      in (f 1, f (1,2))>.
```

(where rset r x stores x in the reference cell r and returns its old contents). That is, the translation of (3)

[1] The shown implementation of lam in CodeReal assumes closed quotations, similar to those of F#. It is easier to discuss let-polymorphism problems in this simpler setting.

```
new_scope @@ fun p1 →
  let f = genlet p1
    (new_scope @@ fun p2 →
       let r = genlet p2 (ref_ nil) in
       lam "x" (fun x → rset r (cons x nil))) in
  pair (app f (int 1)) (app f (pair (int 1) (int 2)))
```

will type-check, if we allow generalization for the genlet p1 expression.

We need something like the value restriction for the quoted **let**. Our translation hence should recognize when the **let**-bound expression is syntactically a function, and use a different combinator. Therefore, (2) is to be translated as

```
new_scope (fun p → let f = genletfun p (fun x → x) in
  pair (app (unarr f) (int 1))
       (app (unarr f) (pair (int 2) (int 3)))
```

where

```
type (+α,+β) arr
val unarr: (α,β) arr cod → (α→β) cod
val genletfun : 'w scope → (α cod → β cod) → (α,β) arr cod
```

The return type of genletfun is (α,β) arr cod, where (α,β) arr is by fiat covariant. Therefore, with the relaxed value restriction, OCaml accepts the translation. We need a similar genletX for other polymorphic values of non-covariant types (which are rare).

In the problematic example (3), the f-bound expression is not syntactically a function, and so we have to use the general genlet rather than the specific genletfun translation. The $(\alpha\to\alpha)$ cod expressions produced by the general translation are not generalizable.

We can implement genletfun for CodeString (which disregards code types) – but not for CodeReal at the moment. The reason for that is that our genletfun is not type safe in one edge case: a cross-stage persistent reference cell – the same case where MetaOCaml is also unsafe[2]. Although the full explanation of this rare edge case is rather involved, intuitively cross-stage-persistent reference cells could be understood as quoted expressions whose translation is

```
let x = genletfun p (let r = ref [] in fun x → rset r [x]) in ...
```

(which is similar to the unsafe example (3), only without quotes.) The value restriction in the quoted code hence has to be strengthened. We conjecture that this edge case is the only problematic one. The implementation can easily exclude it.

Overall, we re-open the value restriction for further investigation.

Acknowledgments

I thank Jacques Garrigue and Atsushi Igarashi for many helpful discussions. Comments and suggestions by Yukiyoshi Kameyama and anonymous reviewers are gratefully appreciated. This work was partially supported by JSPS KAKENHI Grant Numbers 22300005, 25540001, 15H02681.

References

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

[2] J. Garrigue. Relaxing the value restriction. In *FLOPS*, number 2998 in LNCS, pages 196–213, Berlin, 2004. Springer-Verlag.

[3] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shifting the stage: Staging with delimited control. *Journal of Functional Programming*, 21(6): 617–662, 2011. .

[4] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Combinators for impure yet hygienic code generation. In *PEPM*, pages 3–14, New York, 2014. ACM Press. URL `http://dx.doi.org/10.1145/2543728.2543740`.

[5] O. Kiselyov. The design and implementation of BER MetaOCaml - system description. In *FLOPS*, number 8475 in LNCS, pages 86–102. Springer, 2014. .

[6] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, (Sept.), 2013. .

[7] W. Taha. A gentle introduction to multi-stage programming. In *DSPG 2003*, number 3016 in LNCS, pages 30–50, 2004.

[2] `http://okmij.org/ftp/meta-programming/calculi.html#staged-poly`