Compiling a Polymorphic Language

Zach Sullivan and Gordon Lin

Prelude

Our class compiler handles a subset of Typed Racket with simple arithmetic, control flow, functions, and lambdas. Because our language is typed, we will have to rewrite similar functions for different types when the functions actually do the exact same thing with their arguments. A key example of this is the function `id` that just returns its argument. Currently, we would need a separate `id` function for both integers and booleans, which is redundant because their definition is exactly the same except for the types. With parametric polymorphism we can have one definition of `id` that will work with both of those types and any new types we add later on, saving space in the source file and without using up as many names for function and variables.

```
(define (idInteger [x : Integer]) : Integer x)
(define (idBoolean [x : Boolean]) : Boolean x)
```

The Grammar

Adding parametric polymorphism to our language will require our type-level language to be expanded to include a type-variable. The complete grammar is below.

$$\begin{aligned}
\mathit{type} &::= \text{Boolean} \mid \text{Integer} \mid (\text{Vector } \mathit{type}^{+}) \mid \text{Void} \mid (\mathit{type}^{*} \to \mathit{type}) \mid \mathit{typevar} \\
\mathit{expr} &::= \mathit{int} \mid \mathit{var} \mid \#f \mid \#t \mid (\mathit{read}) \mid (\mathit{void}) \mid (-\ \mathit{expr}) \mid (+\ \mathit{expr}\ \mathit{expr}) \\
&\quad \mid (\text{let } ([\mathit{var}\ \mathit{expr}])\ \mathit{expr}) \mid (\text{and } \mathit{expr}\ \mathit{expr}) \mid (\text{or } \mathit{expr}\ \mathit{expr}) \\
&\quad \mid (\text{vector } \mathit{expr}^{+}) \mid (\text{vector-ref } \mathit{expr}\ \mathit{expr}) \mid (\text{vector-set! } \mathit{expr}\ \mathit{expr}\ \mathit{expr}) \\
&\quad \mid (\mathit{expr}\ \mathit{expr}^{*}) \mid (\text{lambda } (\mathit{var}^{*})\ \mathit{expr}) \\
\mathit{defs} &::= (\text{define } (\mathit{var}\ [\mathit{var}\ :\ \tau]^{*})\ :\tau\ \mathit{expr}) \\
R_T &::= (\text{program } \mathit{defs}^{*}\ \mathit{expr})
\end{aligned}$$

With this new grammar, our definition of `id` can be the same for all of our types:

```
(define (id [x : a]) : a x)
```

Implementation

Our approach to implementing parametric polymorphism relies upon the typechecker deciding what correct type of a function where it is called. After the typechecker is complete there should be no more type variables in the program except those in function and lambda definitions.

Simply adding the correct type annotations could cause problems when passing arguments to functions, because arguments will need to have enough space either of the stack or in registers to hold their values. Luckily, our language *only* has values that are 64 bits, including vector pointers, function pointers, integers, and booleans. So in our particular case, we do not have to worry about making sure our data is the right size when we call it.

In order to have the correct types, we have built a new helper function `unify` that takes two types and returns the most general type between them. Here are some key examples:

```
(unify a b)
> error: cannot unify a with b

(unify (Vector a Integer) (Vector a b))
> (Vector a Integer)

(unify (Vector a Integer) (Vector a Boolean))
> error: cannot unify Integer with Boolean
```

Functions required special attention for unification, because the return type of a function depends on the unification of it's arguments. We first unify the function type minus it's return type with the types of the arguments it is applied to. We use this to create a mapping of type variables to their more specific types, a type-environment. After unifying the return types, we replace any type information we gleaned from the unifying the arguments.

```
(define (unify t1 t2)
  (cond
   [(and (monomorphic? t1) (monomorphic? t2)) (if (equal? t1 t2) t1 error)]
   [(and (vector? t1) (vector? t2))
    `(Vector ,@(map unify (vec-types t1) (vec-types t2)))]
   [(and (func? t1) (func? t2))
    (define unified-args (map unify (arg-types t1) (arg-types t2)))
    (define uni-assoc (map cons (arg-types t1) unified-args))
    (define unified-args^ (map (lambda (x) (lookup x uni-assoc x)) unified-args))
    (define urt (unify (return-type t1) (return-type t2)))
    `(,@unified-args^
      ->
      ,(lookup urt uni-assoc urt))]
   [(monomorphic? t2) t2]
   [(monomorphic? t1) t1]
   [(and (symbol? t1) (symbol? t2)) (if (equal? t1 t2) t1 error)]
   [(and (or (vector? t1) (func? t1)) (symbol? t2)) t1]
   [(and (or (vector? t2) (func? t2)) (symbol? t1)) t2]
   [else error]))
```

This unification function is used in our typechecker when we match on function application. After the typechecker, all of the uses of functions will know what types their arguments will be when its called. If a usage does not unify into a concrete type then there is a type error. All the correct programs will know what type a value is as it is passed through the program, as if we had actually defined functions for the specific type a polymorphic function was called on.

Evaluation

We were able to implement parametric polymorphism in our Typed Racket compiler. The first test case we wanted work was this simple example:

```
(define (id [x : a]) : a x)
(id (if (id #f)
        0
        42))
> 42
```

This is a key test case because it ensures that we can use the same function on two different types in the sample program, in this case, an integer and boolean. We also want to make sure that parametric polymorphism works over vectors and functions, which can contain several different types which all must be unified. In addition, vectors and functions were important because they can be nested, and we needed to be sure that our unification recurred down them correctly.

```
(define (swap [v : (Vector a b)]) : (Vector b a)
  (vector (vector-ref v 1) (vector-ref v 0)))
(vector-ref (swap (vector #t 42)) 0)
> 42

-------------------------------------------------------

(define (add1 [x : Integer]) : Integer (+ 1 x))
(define (fmapVec1 [f : (a -> b)]
                  [v : (Vector a)])
  : (Vector b)
  (vector (f (vector-ref v 0))))
(vector-ref (fmapVec1 add1 (vector 41)) 0)
 > 42
```

We have written more test cases that attempt to push the boundaries of our code, including tests that should not unify because the types conflict. Our goals when building test cases was to create ones in which we use a function parameterized with different types and creating enough that we have tested all of our language keywords. Without lots of users of our language writing strange and wonderful programs, we do not know whether or not we have our implementation completely correct. We our limited by our imagination with our test cases.

We have fully implemented parametric polymorphism, which was all we described in our proposal. We wished to take it a step further and implement existential quantifiers in our types, but that was left unfinished.