Interpreting with Type Inference

Introduction

Type inference, as opposed to type-checking aims at inferring the type of variables and functions from the restrictions imposed on them by the code. In order to achieve that it implements a so-called "unification" algorithm, that we saw earlier in the course and will now revisit.

Let us start by reviewing the language that we will consider:

```
(struct varC (s) #:transparent)
(struct numC (n) #:transparent)
(struct boolC (b) #:transparent)
(struct arithC (op e1 e2) #:transparent)
(struct ifC (tst thn els) #:transparent)
(struct pairC (e1 e2) #:transparent)
(struct carC (e) #:transparent)
(struct cdrC (e) #:transparent)
(struct unitC () #:transparent)
(struct letC (s e1 e2) #:transparent)
(struct funC (fname arg ty-arg body ty-body) #:transparent)
(struct callC (e1 e2) #:transparent)
```

The goal of the type-checker and type-inferer is to associate a type with each expression. We therefore need to decide what types we would have. We did that already when we looked at how to implement a type-checker. What is now different is that we need to implement a couple of new types.

- One is a "variable" type. This is a type that not known yet, and needs to be determined, or a type that is hidden behind a polymorphic type, that we'll discuss in a moment. Variable types will simply be indexed 1, 2, 3, . . . , so they'll contain an integer in them. We will have a function `(new-type)` that increments the variable type counter and returns a new variable type, like our store location worked.
- The other is polymorphic types. We will elaborate on these later.

```
(struct numT () #:transparent)
(struct boolT () #:transparent)
(struct unitT () #:transparent)
(struct pairT (t1 t2) #:transparent)
(struct funT (t1 t2) #:transparent)    ;; t1 -> t2
(struct varT (i) #:transparent)     ;; X1, X2, X3
(struct polyT (tvar texpr) #:transparent)
```

The main typecheck method is split in two parts:

- Constraint generation: This takes as input an expression and a static environment, and it returns a pair of a type and a list of constraints. The type is the computed type of the expression. It's a concrete type if it can be determined right away, or it's some sort of variable type otherwise. The constraints are a list of pairs of types, the understanding being that those pairs MUST be equal for the typechecking to succeed. As the constraint generator recursively calls other parts, it collects these constraints together to a total set of constraints. We will use a struct t*c for the return values of the constraint generator.

- Unification: The unification process goes through the list of constraints, and tries to solve them, via the process known as "unification". It basically goes through each pair and either finds a contradiction (a numT that's supposed to also be a boolT) or it learns a new correspondence (this varT 5 is supposed to be a boolT) and then uses that substitution in the rest of the constraints.

So our main typechecker would look something like this:

```
(struct t*c (t c))    ;; The constraint generator return type

(define (typecheck env e)
  (let* ([constr (gen-con env e)]
         [unif (unify (t*c-c constr))])
    (subst-type (t*c-t constr) unif)))
```

Constraint Generation

We will focus first on the constraint generation part. For type of each expression it needs to create a pair of the type of the expression and any constraints. So it will be our usual big cond:

```
(define (gen-con env e)
  (cond
    [(numC? e)
     (t*c (numT) null)]
    [(varC? e)
     (t*c (lookup (varC-s e) env)
          null)]
    [(boolC? e)
     (t*c (boolT) null)]
    [(unitC? e)
     (t*c (unitT) null)]
    ...
    ))
```

The standard values don't take much. Pairs are going to be just a tad harder.

```
    ;; (struct pairC (e1 e2))
    [(pairC? e)
     (let ([r1 (gen-con env (pairC-e1 e))]
           [r2 (gen-con env (pairC-e2 e))])
       (t*c (pairT (t*c-t r1) (t*c-t r2))
            (append (t*c-c r1) (t*c-c r2))))]
```

The conditional is equally straightforward. Remember that in normal typechecking we required that the conditional is a boolean and that the two branches have the same type. Instead of throwing an error if they don't, we now add these statements as constraints:

```
;; (struct ifC (tst thn els))
[(ifC? e)
 (let ([r1 (gen-con env (ifC-tst e))]
       [r2 (gen-con env (ifC-thn e))]
       [r3 (gen-con env (ifC-els e))])
   (t*c (t*c-t r2)
        (list* (cons (t*c-t r1) (boolT))
               (cons (t*c-t r2) (t*c-t r3))
               (append (t*c-c r1) (t*c-c r2) (t*c-c r3)))))]
```

We will next deal with arithmetic operators. They are similar:

```
;; (struct arithC (op e1 e2))
[(arithC? e)
 (let ([r1 (gen-con env (arithC-e1 e))]
       [r2 (gen-con env (arithC-e2 e))])
   (t*c (numT)
        (list* (cons (t*c-t r1) (numT))
               (cons (t*c-t r2) (numT))
               (append (t*c-c r1) (t*c-c r2)))))]
```

Next we will consider carC. This one is somewhat tricky. Normally we would compute the type of its component, then require that type to be a pairT. We can try to do the same, but the type doesn't have to literally be a pairT. It could be for example some variable type varT 6 that only after a lot of future work ends up being pairT. We can't wait for that, we need to impose our constraints now.

We really don't have many available options: We must create a new pairT type, with two newly created variable types as its components, then add the appropriate constraints. We will add an extra optimization. In the event where the type of the subexpression is indeed a pairT type, we don't need to create new types.

```
;; (struct carC (e))
[(carC? e)
 (let* ([r (gen-con env (carC-e e))]
        [ty-e (if (pairT? (t*c-t r))
                  (t*c-t r)
                  (pairT (new-type) (new-type)))])
   (t*c (pairT-t1 ty-e)
        (list* (cons (t*c-t r) ty-e)
               (t*c-c r))))]
```

Of course, cdrC is similar. Let statements are straightforward, for now:

```
;; (struct letC (s e1 e2))
[(letC? e)
 (let* ([r1 (gen-con env (letC-e1 e))]
        [r2 (gen-con (bind (letC-s e) (t*c-t r1) env)
                     (letC-e2 e))])
   (t*c (t*c-t r2)
        (append (t*c-c r1) (t*c-c r2))))]
```

Calls are a bit like carC. They expect the first argument to be a function type. If it is not, they need to build a new function type to use in its place.

```
;; (struct callC (e1 e2))
[(callC? e)
```

```
     (let* ([r1 (gen-con env (callC-e1 e))]
            [r2 (gen-con env (callC-e2 e))]
            [ty-f (if (funT? (t*c-t r1))
                      (t*c-t r1)
                      (funT (new-type) (new-type)))])
       (t*c (funT-t2 ty-f)
            (list* (cons (funT-t1 ty-f) (t*c-t r2))
                   (cons ty-f (t*c-t r1))
                   (append (t*c-c r1) (t*c-c r2)))))]
```

Lastly, our function definitions. If they have provided us with a type for the argument, we must use that, otherwise create a new type variable. Same with the return type (ty-body).

```
    ;; (struct funC (fname arg ty-arg body ty-body))
    [(funC? e)
     (let* ([targ (or (funC-ty-arg e) (new-type))]
            [tbody (or (funC-ty-body e) (new-type))]
            [env1 (bind (funC-arg e) targ env)]
            [env2 (if (funC-fname e)
                      (bind (funC-fname e) (funT targ tbody) env1)
                      env1)]
            [r (gen-con env2 (funC-body e))])
       (t*c (funT targ tbody)
            (list* (cons tbody (t*c-t r))
                   (t*c-c r))))]
```

Phew. Let's look at an example. Consider for instance the following program:

```
;; ex1 is function: f(x) = x + 2 called on 10
(define ex1
  (callC (funC #f 'x #f
               (arithC + (varC 'x) (numC 2)) #f)
         (numC 10)))
```

It defines a simple non-recursive function that adds 2 to its parameter, then calls it on 10. Let us look at the constraints generated:

```
(list
  (cons (varT 1) (numT))    ;; callC-e1 arg type must match callC-e2 type
  (cons (varT 2) (numT))    ;; funC-ty-body must match body's type of numT
  (cons (varT 1) (numT))    ;; arithC-e1 must be numT
  (cons (numT) (numT)))     ;; arithC-e2 must be numT
```

This concludes our first pass on constraint generation. We will need to revisit some ideas when we discuss polymorphic types.

Unification

During unification we must look at the list of constraints and determine what they tell us about our types. The work roughly consists of the following parts:

- If we have two primitive types, they better be the same.

- If we have two non-primitive types (pair, function), they better be the same kind and their parts better unify.
- If we have a variable type on one side, then:
  - If we have the same type on the other side, then we don't learn anything, just move on.
  - If we have a "safe" type on the other side (not containing the first variable type) then we learned a new "substitution": Add it to our return dictionary and substitute in the rest of the clauses.
  - If we have a type on the other side that somehow contains the first variable type, this is an error.
- Everything else is an error.

Let us start building this function:

```
(define (unify constrs)
  (if (null? constrs)
      null
      (let* ([nxt (car constrs)]
             [t1 (car nxt)]
             [t2 (cdr nxt)]
             [rst (cdr constrs)])
        (cond
          [(equal? t1 t2) (unify rst)]
          [(and (boolT? t1) (boolT? t2))
           (unify rst)]
          [(and (numT? t1) (numT? t2))
           (unify rst)]
          [(and (unitT? t1) (unitT? t2))]
          ...
          [else (error "Types don't match")]
          ))))
```

If there are no constraints left then we can return an empty list of substitutions. Otherwise we must do something with the next constraint. The atomic types must match.

Next up we have pairT. It must instead expect that the corresponding components match, so we add those in the list of constraints.

```
          [(and (pairT? t1) (pairT? t2))
           (unify (list* (cons (pairT-t1 t1) (pairT-t1 t2))
                         (cons (pairT-t2 t1) (pairT-t2 t2))
                         rst))]
```

Notice that this seemingly increases the number of constraints by 1. But it also *simplifies* those constraints. Functions are similar. Lastly, we have the case of variables.

We now are left with the cases where one of the two types is a variable type. If the other one is also a variable type, then either they are the same and we can ignore the constraint, or they are different, in which case we learn a new substitution, of say the first type for the second. We then add that to our return value, and continue unifying the remaining constraints, substituting the first type for the second throughout:

```
          [(and (varT? t1) (varT? t2))
           (if (equal? (varT-i t1) (varT-i t2))
               (unify rst)
               (list* (cons t1 t2)
                      (unify (subst-all t1 t2 rst))))]
```

This requires that we implement the subst-all function. It takes a type and another type, and a list of constraints, and it substitutes the first type with the second throughout. What is left is a set of costraints that do not contain the first type. We will take advantage of Racket's flexibility here and implement a function that handles any combination of cells containing types. The big conditional has a couple of cases to handle the recursion, then deals with the various recursive types. Every atomic type just stays itself.

```
(define (subst-all t1 t2 els)
  (cond [(null? els) null]
        [(pair? els) (cons (subst-all t1 t2 (car els))
                           (subst-all t1 t2 (cdr els)))]
        [(equal? t1 els) t2]
        [(pairT? els) (pairT (subst-all t1 t2 (pairT-t1 els))
                             (subst-all t1 t2 (pairT-t2 els)))]
        [(funT? els) (funT (subst-all t1 t2 (funT-t1 els))
                           (subst-all t1 t2 (funT-t2 els)))]
        [else els]))
```

Back to our constraint unification, if the second type is a variable type and the first isn't, then we simply flip the sides and call ourselves:

```
          [(varT? t2)
           (unify (list* (cons t2 t1)
                         rst))]
```

All that remains will then be the case where the first type is a variable type and the second isn't. We need to check if the second type contains the first in it. If it does, this is an error in the constraints. Otherwise we've learned a new substitution:

```
          [(varT? t1)
           (if (contained t1 t2)
               (error "Type contained in itself")
               (list* (cons t1 t2)
                      (unify (subst-all t1 t2 rst))))]
```

The function contained is rather straightforward:

```
(define (contained t1 t2)
  (cond [(equal? t1 t2) #t]
        [(pairT? t2) (or (contained t1 (pairT-t1 t2))
                         (contained t1 (pairT-t2 t2)))]
        [(funT? t2) (or (contained t1 (funT-t1 t2))
                        (contained t1 (funT-t2 t2)))]
        [else #f]))
```

This is almost the end of our unification algorithm! What it does is take a list of constraints, and turn it into a list of "substitutions", where each substitution has a variable on the left and its "result" on the right.

What is missing is a back-substitution step. To explain the need for it, let us look at another example:

```
(define ex2
  (funC #f 'x #f
        (arithC + (carC (varC 'x)) (cdrC (varC 'x)))
        #f))
```

This function expects a pair as input, and simply adds its two components. Here's what the list of constraints looks like:

```
;; Return type is: (funT (varT 1) (varT 2))
(list
  (cons (varT 2) (numT))          ;; arithC / function body return is numT
  (cons (varT 3) (numT))          ;; arithC-e2 needs to be numT
  (cons (varT 6) (numT))          ;; arithC-e2 needs to be numT
  (cons (varT 1)
        (pairT (varT 3) (varT 4)))       ;; carC needs a pair
  (cons (varT 1)
        (pairT (varT 5) (varT 6)))))    ;; cdrC needs a pair
```

The unification process results in the following substitutions:

```
(list
  (cons (varT 2) (numT))
  (cons (varT 3) (numT))
  (cons (varT 6) (numT))
  (cons (varT 1)
        (pairT (numT) (varT 4)))
  (cons (varT 5) (numT))
  (cons (varT 4) (numT)))
```

Note that when we learned that varT 1 is a pair we only knew the value of the first part of that pair, the second part would only come later. But we've already committed to a substitution for varT 1 at that point. What we need to do now is back-substitution: Any variable that is defined later in the list may appear in the substitution of a variable somewhere earlier. The other way around is not possible as when we established a new substitution we also performed that substitution in all yet-unresolved constraints. We can do this via a foldr:

```
(define (back-sub subs)
  (foldr (lambda (sub rest)
           (list* (cons (car sub)
                        (subst-type (cdr sub) rest))
                  rest))
         '()
        subs))
```

where the subst-type function is a function that we need to write, which takes a type and a list of substitutions and performs those substitutions on the type.

```
(define (subst-type type subs)
  (cond [(pairT? type) (pairT (subst-type (pairT-t1 type) subs)
                              (subst-type (pairT-t2 type) subs))]
        [(funT? type) (funT (subst-type (funT-t1 type) subs)
                            (subst-type (funT-t2 type) subs))]
        [(varT? type)
```

```
         (let ([pr (assoc type subs)])
           (if pr (cdr pr) type))]
        [else type]))
```

Polymorphic Types

All of the above will work great for many cases. But there are instances where the types cannot be precisely determined. Consider for example the following function, that given an input returns a pair having that input in both parts:

```
(define f
  (funC #f 'x #f
        (pairC (varC 'x) (varC 'x))
        #f))
```

Then the constraint generation would give us:

```
;; return type: (funT (varT 1) (varT 2))
(list
  (cons (varT 2)
        (pairT (varT 1) (varT 1))))
```

and the unification part won't really add much to that. So we end up with a function type: `(funT (varT 1) (pairT (varT 1) (varT 1)))`. Which makes sense. There's absolutely no constraints on what kind of value we can feed the function. This is the classical case where in OCAML we used a polymorphic type. In effect we keep the type a variable, until we actually call the function. There is a problem however. Imagine the following, which for simplicity we write in OCAML:

```
let f = fun x -> (x, x)
in (f 2, f true)
```

This should produce the value `((2, 2), (true, true))`. In our system it will give an error. This is how the expression might look like in our system:

```
(define ex3
  (letC 'f f
    (pairC (callC (varC 'f) (numC 2))
           (callC (varC 'f) (boolC #t)))))
```

We will get a "type do not match" error. This is because each time we end up using the function, we get a constraint binding that variable type to the type of the passed value. If we call the function with different types of values, they will produce conflicting constraints on the variable.

A simple solution to this is as follows: Each time we use the function, we create a fresh variable to use, we essentially instantiate the variable type. The problem is that this may cause us to lose constraints that might have been in place in the function. We must not let future constraints determine the type of a function that was defined earlier.

Therefore at every function definition we need to:

- Unify the constraints so far
- Determine if we have any "free" variable types
- Somehow indicate that these are indeed free variable types

And when we do a function call, we must create a new variable type and "instantiate" our function with this new variable type. This won't actually happen on the function call, but instead during constraint unification.

In order to achieve this we create a new "polymorphic" type. It looks like this: (polyT t1 t2) where t1 is meant to be a variable type, the parameter, and t2 is the type that is supposed to depend on that variable. For example the polymorphic type 'a -> 'a may be thought of as (polyT (varT 1) (funT (varT 1) (varT 1))). We can also nest these polymorphic types. For instance 'a -> 'b might be thought of as (polyT (varT 1) (polyT (varT 2) (funT (varT 1) (varT 2)))).

We need to first of all change our funC constraint generation:

```
[(funC? e)
 (let* ([targ (or (funC-ty-arg e) (new-type))]
        [tbody (or (funC-ty-body e) (new-type))]
        [env1 (bind (funC-arg e) targ env)]
        [env2 (if (funC-fname e)
                  (bind (funC-fname e) (funT targ tbody) env1)
                  env1)]
        [r (gen-con env2 (funC-body e))]
        ;; New stuff here:
        [subs (unify (list* (cons tbody (t*c-t r))
                            (t*c-c r)))]
        [targ2 (subst-type targ subs)]
        [tbody2 (subst-type tbody subs)])
   (t*c (generalize (funT targ2 tbody2))
        subs))]
```

Here generalize is a function that takes as input a type, finds all the distinct variable types in it, and "generalizes" them via polymorphic types.

```
(define (generalize type)
  (foldr polyT type (uniq (get-vars type))))

(define (get-vars type)
  (cond [(varT? type) (list type)]
        [(pairT? type) (append (get-vars (pairT-t1 type))
                               (get-vars (pairT-t2 type)))]
        [(funT? type) (append (get-vars (funT-t1 type))
                              (get-vars (funT-t2 type)))]
        ;; Poly should not occur when we call this function
        ;; Other cases are atomic
        [else null]))

(define (uniq vars)
  (define (aux sofar vars)
    (if (null? vars)
        sofar
        (aux (if (member (car vars) sofar)
                 sofar
```

```
                (cons (car vars) sofar))
            (cdr vars))))
  (aux null vars))

(define (instantiate t)
  (subst-all (polyT-tvar t)
             (new-type)
             (polyT-texpr t)))
```

We need to amend our unification algorithm. When a polymorphic type is encountered, it must be instantiated with a fresh variable and then unified (these cases need to be added before the handling of variable types in unify).

```
[(polyT? t1)
 (unify (list* (cons (instantiate t1) t2)
               rst))]
[(polyT? t2)
 (unify (list* (cons t1 (instantiate t2))
               rst))]
```

In practice it might take a bit more work to get polymorphic types to work properly, but the above at least gives you an idea of what is involved.

Value restriction

There is an important problem related to polymorphic types that is worth discussing. It relates to mutation in combination with polymorphic typing. Here's an example (we can make a similar example using functions):

```
let r = ref None;;
r := Some "hi";;
1 + valOf (!r)
```

Let us think of what each line does.

- The first line creates a reference, and must give it a type. in this this case that type is 'a option ref.
- The second line puts a new value in the reference. The value's type is string option, which is compatible with the contained type 'a option.
- The third line recovers the value stored in the reference, a 'a option value, grabs the actual value in the option, of polymorphic type 'a, then adds 1 to it. This will also typecheck.

So we have three lines that according to our rules should typecheck. But at runtime we will get an error, as we are adding 1 to a string! Our type system is no longer sound!

There are two lessons to learn from this. The first is that soundness of a type system needs to be proven in some formal way, and a decent amount of programming language theory is devoted to that goal. The second is of course that we need to make some changes to our type system that don't allow programs like the one above.

The specific change needed is what is known as “value restriction”. It says that generalizing variables to form a polymorphic type will only happen if the expression on the right-hand-side of the assignment is a *syntactic value*. And `ref None` in the previous example is not a value, it is a function call. So it can’t have any generalized types.