Type Inference and Type Habitation (5/10/2004)

Daniel P. Friedman, William E. Byrd, David W. Mack

Computer Science Department, Indiana University
Bloomington, IN 47405, USA

Oleg Kiselyov

Fleet Numerical Meteorology and Oceanography Center,
Monterey, CA 93943, USA

An interesting application of logic programming is the type-inference problem. We start by considering integers and booleans. Next, we include some familiar primitives. When we are comfortable with those features we include conditionals, followed by lexical variables, then `lambda`, application, and `fix` expressions. Finally, we include polymorphic `let`. Type inference allows for the system to determine a unique type if the expression has one. Similarly, using the same program we can determine an expression that has a given type. This last application is called *type habitation*.

The language for which we infer a type is basically a lambda-calculus variant of Scheme. We have chosen, however, to parse this variant into a language where every expression has a tag as in `parse` below.

```
(define parse
  (lambda (e)
    (cond
      [(symbol? e) `(var ,e)]
      [(number? e) `(intc ,e)]
      [(boolean? e) `(boolc ,e)]
      [else
        (case (car e)
          [(zero?) `(zero? ,(parse (cadr e)))]
          [(sub1) `(sub1 ,(parse (cadr e)))]
          [(+) `(+ ,(parse (cadr e)) ,(parse (caddr e)))]
          [(*) `(* ,(parse (cadr e)) ,(parse (caddr e)))]
          [(if) `(if ,(parse (cadr e)) ,(parse (caddr e)) ,(parse (cadddr e)))]
          [(fix) `(fix ,(parse (cadr e)))]
          [(lambda) `(lambda ,(cadr e) ,(parse (caddr e)))]
          [(let) `(let ([,(car (car (cadr e))) ,(parse (cadr (car (cadr e))))])
                    ,(parse (caddr e)))]
          [else `(app ,(parse (car e)) ,(parse (cadr e)))])])))
```

While you are reading the definitions for the type system, `!-`, it is important to keep in mind that although we are presenting a type inferencing algorithm, it is just a relatively simple logic program.

`!-` corresponds to the mathematical symbol $\vdash$ (turnstile) and reads "we can infer." That is, from looking at the relation `intc-rel` and the definition of `!-` below, we can read it as, "From `g` *we can infer* that `x` is of type `int` provided that `x` is an `intc`." For now, we leave `g` unspecified.

In the expressions below, we use `int`, `bool`, and `-->` for our type constructors.

```
(define intc-rel
  (fact (g x) g `(intc ,x) 'int))

(define !- intc-rel)
```

```
> (answer (?)
    (eigen (g)
      (!- g (parse 17) ?)))
? :: int
```

```
> (answer ()
    (eigen (g)
      (!- g (parse #t) 'int)))
#f
```

In the first example, we verify that `17` is of type `int`. In this case, `?` is instantiated to `int`. The second one does not display the `trace-vars`, so this one does not hold: The type of `#t` is not `int`. Even more interesting, is to try

```
> (answer (?)
    (eigen (g)
      (!- g (parse #t) ?)))
#f
```

which also displays nothing, so `#t` does not have a type according to the current definition of `!-`. Why? Because we have not yet extended the definition of `!-` to include a rule for boolean constants. So, we include `boolc-rel` below in `!-`.

```
(define boolc-rel
  (fact (g x) g `(boolc ,x) 'bool))

(define !- (extend-relation (a1 a2 a3) !- boolc-rel))
```

```
> (answer (?)
    (eigen (g)
      (!- g (parse #t) ?)))
? :: bool
```

Before we include relations for the arithmetic primitives, we observe that we need to use `all!!`. We do this because our type inferencer is deterministic. There cannot be any backtracking. That is, once a decision is made, it cannot be reconsidered!

Now we can extend `!-` below with the relations `zero?-rel` `sub1-rel`, `+-rel`, and `*-rel`, which correspond to the primitves `zero?`, `sub1`, `+`, and `*`, respectively.

```
(define zero?-rel
  (relation (g x)
    (to-show g `(zero? ,x) 'bool)
    (all!! (!- g x 'int))))

(define sub1-rel
  (relation (g x)
    (to-show g `(sub1 ,x) 'int)
    (all!! (!- g x 'int))))

(define +-rel
  (relation (g x y)
    (to-show g `(+ ,x ,y) 'int)
    (all!! (!- g x 'int) (!- g y 'int))))

(define *-rel
  (relation (g x y)
    (to-show g `(* ,x ,y) 'int)
    (all!! (!- g x 'int) (!- g y 'int))))

(define !- (extend-relation (a1 a2 a3) !- zero?-rel sub1-rel +-rel *-rel))

> (answer (?)
    (eigen (g)
      (!- g (parse '(zero? 24)) ?)))
? :: bool

> (answer (?)
    (eigen (g)
      (!- g (parse '(zero? (+ 24 50))) ?)))
? :: bool
```

The type system can infer that `(zero? 24)` is of type `bool`, because it can infer that `24` is of type `int`. It can infer that `(+ 24 50)` is of type `int`, so the answer in the second example must be of type `bool`. We can, of course, make more complicated examples using `zero?`, `sub1`, `+`, and `*`, but if they have a type, it is `int` or `bool`. For example,

```
> (answer (?)
    (eigen (g)
      (!- g (parse '(zero? (sub1 (+ 18 (+ 24 50))))) ?)))
? :: bool
```

Although our parser expects a larger language, at each stage of defining `!-`, we are writing a type inferencer for a larger and larger language. When we have defined `!-` for `let`, then we have a type inferencer for the full language. To reiterate, the language starts out very small! It only contains integers. Then, as we progress, it gets larger. But, the beauty of type inferencing, is that these little relations grow

naturally. Of course, we cannot test the relation for `zero?` until we have a relation for numbers and booleans, so there is a natural ordering to some extent.

In this type system, we must preserve the property that *Every well-typed expression has* ***one*** *type*. So, what do we do about conditionals? Easy. We require that not only must the test be of type `bool`, but the true branch and the false branch must have the *same* type. In a language without variables, `lambda` expressions, applications, `fix` expressions, and `let` expressions, that means that they must both be of type `int` or they must both be of type `bool`. By extending `!-`, we can now handle `if` expressions.

```
(define if-rel
  (relation (g t test conseq alt)
    (to-show g `(if ,test ,conseq ,alt) t)
    (all!! (!- g test 'bool) (!- g conseq t) (!- g alt t))))

(define !- (extend-relation (a1 a2 a3) !- if-rel))

> (answer (?)
    (eigen (g)
      (!- g (parse '(if (zero? 24) 3 4)) ?)))
? :: int
```

And we discover the test has type `bool` and the entire expression has type `int`.

Next, we include lexical variables, which are represented using symbols. What is the type of `(zero? a)`? If the type of `a` is `int`, then we know that the type of the entire expression is `bool`, but if the type of `a` is `bool`, then the expression does not have a type. How do we determine the type of `a`? We look in `g`, which is a type environment that associates lexical (both generic and non-generic) variables with types. So far we have ignored `g`, but now we consider its content using `non-generic` (a tag) variables. We extend `!-` to include a relation for variables.

```
(define var-rel
  (relation (g v t)
    (to-show g `(var ,v) t)
    (all!! (env g v t))))

(define !- (extend-relation (a1 a2 a3) !- var-rel))
```

Now we must define the `env` relation.

```
(define non-generic-base-env
  (fact (g v t) `(non-generic ,v ,t ,g) v t))

(define non-generic-recursive-env
  (relation (g  v t)
    (to-show `(non-generic ,_ ,_ ,g) v t)
    (all!! (instantiated g) (env g v t))))
```

```
(define env (extend-relation (a1 a2 a3)
              non-generic-base-env
              non-generic-recursive-env))

> (answer (?)
    (eigen (g)
      (env `(non-generic b int (non-generic a bool ,g)) 'a ?)))
? :: bool

> (answer (?)
    (eigen (g)
      (!- `(non-generic a int ,g) (parse '(zero? a)) ?)))
? :: bool

> (answer (?)
    (eigen (g)
      (!- `(non-generic b bool (non-generic a int ,g))  (parse '(zero? a)) ?)))
? :: bool
```

The first example tests `env`. The environment starts out with `int` bound to `b` and `bool` bound to `a`. The `non-generic-recursive-env` relation succeeds, since we are looking up `a`, and then `non-generic-base-env` succeeds, since we find `a`. In the second answer, we have one item in the type environment and the `non-generic-base-env` relation is followed. In the third example, we have two items in the type environment, so we take `non-generic-recursive-env`, then `non-generic-base-env` succeeds, since we have stripped off `b` and `bool`, leaving `a` and its associated type.

Now that we can deal with lexical (non-generic) variables, we can consider the relation for `lambda` expressions by extending `!-`.

```
(define lambda-rel
  (relation (g v t body type-v)
    (to-show g `(lambda (,v) ,body) `(--> ,type-v ,t))
    (all!! (!- `(non-generic ,v ,type-v ,g) body t))))

(define !- (extend-relation (a1 a2 a3) !- lambda-rel))

> (answer (?)
    (eigen (g)
      (!- `(non-generic b bool (non-generic a int ,g))
          (parse '(lambda (x) (+ x 5)))
          ?)))
> (answer (?)
    (eigen (g)
      (!- `(non-generic b bool (non-generic a int ,g))
          (parse '(lambda (x) (+ x 5)))
          ?)))
? :: (--> int int)
```

```
> (answer (?)
    (eigen (g)
      (!- `(non-generic b bool (non-generic a int ,g))
          (parse '(lambda (x) (+ x a)))
          ?)))
? :: (--> int int)

> (answer (?)
    (eigen (g)
      (!- g (parse '(lambda (a) (lambda (x) (+ x a)))) ?)))
? :: (--> int (--> int int))
```

In the first answer, we see that we have an arrow (`-->`) type. The left argument of the arrow type is the type of argument coming into the function, and the right argument of the arrow type is the type of the result going out of the function. So, the inferred type is a function whose argument is an integer and whose result is an integer. The second answer states that the argument is an integer, but consults the type environment to make sure that the argument going out is an integer. In the third example, we forget about the first environment, because there are no free variables in the expression. We see that the argument coming in is an integer, but the result is an arrow type, which takes in an integer and returns an integer. Close inspection of the type (directly aligned below the item it is typing) shows that for each `lambda` there is an arrow and for each formal parameter there is a type. Also, there is a type for the body of each `lambda`. If we think about the type from the inside out, we see that `(+ x a)` is an integer only if `x` and `a` are integers. That determines the type of the inner `lambda` and then the type of the outer `lambda`. It is important that we *can* infer the type of `lambda` expressions, even though we do not yet have application in our language. This should be a bit of a surprise.

We come next to application.

```
(define app-rel
  (relation (g rator rand t)
    (to-show g `(app ,rator ,rand) t)
    (exists (t-rand)
      (all!!
        (!- g rator `(--> ,t-rand ,t))
        (!- g rand t-rand)))))

(define !- (extend-relation (a1 a2 a3) !- app-rel))
```

In determining the type of an application, we know that the operator in an application should be some arrow type. Furthermore, once we know that type, we know that the type going out of that type is the same as the type of the entire application and we know that the operand of the application must be the type going into that type.

```
> (answer (?)
    (eigen (g)
      (!- g  (parse '(lambda (f) (lambda (x) ((f x) x)))) ?)))
? :: (--> (--> t.0 (--> t.0 t.1)) (--> t.0 t.1))
```

Here, the type of `f` is `(--> t.0 (--> t.0 t.1))`, so the type of `x` must be `t.0`, and the type of `(lambda (x) ((f x) x))` must be `(--> t.0 t.1)`. As should be evident, once we add a relation for application, things start to get a bit tricky. We can no longer rely on aligning the `lambda`s with the arrows. Here we have two `lambda`s and four arrows. Yet another surprise. Our inferencer is starting to be clever.

We may be tempted to use our language to write (and test) recursive functions. To test the expression, we use the call-by-value `fix` primitive:

```
(define fix-rel
  (relation (g rand t)
    (to-show g `(fix ,rand) t)
    (all!! (!- g rand `(--> ,t ,t)))))

(define !- (extend-relation (a1 a2 a3) !- fix-rel))
```

```
> (answer (?)
    (eigen (g)
      (!- g
          (parse
            '((fix (lambda (sum)
                     (lambda (n)
                       (if (zero? n)
                         0
                         (+ n (sum (sub1 n)))))))
              10))
          ?)))
? :: int
```

In Scheme, we define `fix` below. Although `fix` can be defined using simple `lambda` terms as in the `Y` combinator, *our* type system cannot determine `Y`'s type. Thus, `fix` must be primitive and the associated primitive call's type can be inferred as above.

```
(define fix
  (lambda (e)
    (e (lambda (z) ((fix e) z)))))
```

Let's consider the following expression

```
> ((fix (lambda (sum)
          (lambda (n)
            (+ n (sum (sub1 n))))))
   10)
```

It fails to terminate. But, can we infer its type? Yes, *an expression may have a type, even if evaluating it would lead to nontermination.* This is a confusing aspect of type inference. We know from the "Halting Problem" that we cannot tell in advance whether evaluating an arbitrary expression will terminate, but *this* type inferencing system is guaranteed to terminate. Thus, we can infer the type before we run it. As a result, information that the run-time system can learn from the type (or the process of inferring the type) can be put to good use. Here is its type.

```
> (answer (?)
    (eigen (g)
      (!- g
          (parse
            '((fix (lambda (sum)
                     (lambda (n)
                       (+ n (sum (sub1 n))))))
              10))
          ?)))
? :: int
```

The `let`-expression is a bit more subtle. Let's take a look at an expression that should type check, but won't in the absence of `let`.

```
> (answer (?)
    (eigen (g)
      (!- g
          (parse
            '((lambda (f)
                (if (f (zero? 5))
                  (+ (f 4) 8)
                  (+ (f 3) 7)))
              (lambda (x) x)))
          ?)))
#f
```

Because `f` becomes the non-generic identity, once a type for `f` is determined, it must stay the same. Obviously, we would expect that the evaluation of "`((lambda (f) ...) ...)` to be 10, but it has no type. If, however, we change the expression to use `let`

```
(let ([f (lambda (x) x)])
  (if (f (zero? 5))
    (+ (f 4) 8)
    (+ (f 3) 7)))
```

and think about β-substituting for `f` throughout,

```
(if ((lambda (x) x) (zero? 5))
  (+ ((lambda (x) x) 4) 8)
  (+ ((lambda (x) x) 3) 7))
```

then we can see that this expression should have a type, `int`. Instead of doing the substitution, we mark certain variables *generic* as they are placed in the environment.

Below is the *polymorphic* `let`, since the variable is tagged with `generic` in the environment. If the variable were tagged with `non-generic`, then this would be the familiar `let`. We have only to determine what happens in environment lookup when a variable with a `generic` tag is an arrow type. (Those who wish to include a more general `let` expression, one whose binding variable is bound to a `non-generic`, feel free to do so, but for our purposes, we assume that all right-hand sides of `let` expressions are `lambda` expressions.)

```
(define polylet-rel
  (relation (g v rand body t)
    (to-show g `(let ([,v ,rand]) ,body) t)
    (exists (t-rand)
      (all!!
        (!- g rand t-rand)
        (!- `(generic ,v ,t-rand ,g) body t)))))

(define !- (extend-relation (a1 a2 a3) !- polylet-rel))
```

The relation `logical-copy-term` takes a term with variables within it and creates a new term with new, but corresponding variables in it. Thus if the same variable occurs in several positions in the input term, then a new variable will occur in the same positions in the output term.

In `logical-copy-term` below, we pass `input-term`, and `output-term`, along with an empty table and a table `varmap`, with no bindings, which associates each input term variable with an output term variable.

```
(define logical-copy-term
  (relation (input-term output-term)
    (to-show input-term output-term)
    (exists (varmap)
      (logical-copy-term-raw input-term output-term '() varmap))))
```

In `logical-copy-term-raw`, if the `ef/only` test succeeds, then `car` is bound to the `car` of the input term and `cdr` is bound to the `cdr` of the input term. The

two recursive calls feed the variable map generated from the first call to the seocnd call. When the second call binds varmap, it contains an association list which binds each input variable to each output variable. of the first call as the input list of the second call. Then `output-term` is constructed from `new-car` and `new-cdr`. If the input is not instantiated, it is a free variable, and we lookup its value in the variable map. So, for the invocation of `logical-assq`, `input-term` is a logic variable and `output-term` is a logic variable.

```
(define logical-copy-term-raw
  (relation (input-term output-term lst varmap)
    (to-show input-term output-term lst varmap)
    (exists (car cdr)
      (ef/only (all!! (instantiated input-term) (== input-term `(,car . ,cdr)))
        (exists (new-car new-cdr varmap-car)
          (all!!
            (logical-copy-term-raw car new-car lst varmap-car)
            (logical-copy-term-raw cdr new-cdr varmap-car varmap)
            (== output-term `(,new-car . ,new-cdr))))
        (logical-assq input-term lst output-term varmap)))))
```

In the first relation of `logical-assq`, we make no changes, since `input-var` is bound, and we are only replacing free variables. In the second relation, we bind `var` and `output-var` from the first element of `varmap` provided that `input-var` and `var` are the same variable. In the third relation, we do the natural recursion on the `cdr` of `varmap` looking for a match. In the final relation, which is a fact, if the varmap is empty, we add one association to it.

```
(define logical-assq
  (extend-relation (a1 a2 a3 a4)
    (relation (input-var varmap)
      (to-show input-var varmap input-var varmap)
      (instantiated input-var))
    (relation (input-var varmap output-var)
      (to-show input-var varmap output-var varmap)
      (exists (var)
        (all!!
          (== varmap `((,var . ,output-var) . ,_))
          (predicate (*equal? var input-var)))))
    (relation (input-var h varmap output-var varmap1)
      (to-show input-var `(,h . ,varmap) output-var `(,h . ,varmap1))
      (logical-assq input-var varmap output-var varmap1))
    (fact (input-var output-var) input-var '() output-var `((,input-var . ,output-var)))))
```

```
(define *equal?
  (lambda (x y)
    (cond
      [(and (var? x) (var? y)) (eq? x y)]
      [(var? x) #f]
      [(var? y) #f]
      [else (equal? x y)])))
```

We can now implement generic variables using the `logical-copy-term` relation.

```
(define generic-base-env
  (relation (g v targ tresult t)
    (to-show `(generic ,v (--> ,targ ,tresult) ,g) v t)
    (logical-copy-term `(--> ,targ ,tresult) t)))

(define generic-recursive-env
  (relation (g v t)
    (to-show `(generic ,_ ,_ ,g) v t)
    (all! (env g v t))))

(define generic-env
  (extend-relation (a1 a2 a3) generic-base-env generic-recursive-env))

(define env (extend-relation (a1 a2 a3) env generic-env))
```

Now that we have extended our environments to handle generic as well as non-generic variables, we can infer the right type.

```
> (answer (?)
    (eigen (g)
      (!- g
          (parse
            '(let ([f (lambda (x) x)])
               (if (f (zero? 5))
                 (+ (f 4) 8)
                 (+ (f 3) 7))))
          ?)))
? :: int
```

Finally we consider type habitation. We are going to do an experiment and in order to get the results we want, we need to respecify the order of the relations. Thus we redefine `!-`.

```
(define !-
  (extend-relation (a1 a2 a3)
    var-rel intc-rel boolc-rel zero?-rel sub1-rel +-rel
    if-rel lambda-rel app-rel fix-rel polylet-rel))
```

Here are three, perhaps unexpected, examples.

```
> (answer (g ?)
    (!- g ? '(--> int int)))
g :: (non-generic v.0 (--> int int) g.0)
? :: (var v.0)

> (answer (la f b)
    (eigen (g)
      (!- g `(,la (,f) ,b) '(--> int int))))
la :: lambda
f :: v.0
b :: (var v.0)

> (answer* 1 (h r q z y t)
    (eigen (g)
      (!- g `(,h ,r (,q ,z ,y)) t)))
h :: +
r :: (intc x.0)
q :: +
z :: (intc x.1)
y :: (intc x.2)
t :: int

h :: lambda
r :: (v.0)
q :: +
z :: (var v.0)
y :: (var v.0)
t :: (--> int int)

#f
```

The first example attempts to find an expression whose type is `(--> int int)`, but instead finds a type environment that binds that type to the variable `v.0`, and then the expression is trivially `v.0`. The second example produces an expression given the type. This is answering the question, "What expression *inhabits* that type?" In our case, the identity function inhabits that type. But, to make these first two examples work, we had to place `var-rel` first in the most recent definition of `!-`. The last example yields two answers. First, it infers that `t` must be of `int` type. Then since there is only one binary operation that returns an `int` (i.e., `+`), it determines `q` and `h`. Next, we can infer that `r`, `z`, and `y` must be of `int` type, and what is easier than making them all values of type integer. Second, it infers that `t` is of type `(-> int int)`. That forces `h` to be the symbol `lambda`. Then we need an expression whose value is of type `int`. Thus, `r` becomes `(v.0)`, and `q` becomes `+`, the first binary operator on integers. Finally, `z` and `y` become `(var v.0)`.

`Exercise 1:` Take the results of these four test programs and reconstruct what the terms are by substituting for each variable. ◇