Type Inference: Constraint Generation

sk and dbtucker

2002-11-11

1 Inferring Types

We've seen the value of having explicit polymorphism in our language—it lets us write programs that work on may different types of values. Even mainstream languages like C++ have recognized the value of this form of polymorphism (called *parametric polymorphism*), and they have noticed that it complements and is not subsumed by the polymorphism common to object-oriented languages (called *subtype polymorphism*).

That said, it's pretty painful to write programs such as this:

```
(define length
  ⟨Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?⟨τ⟩ l) 0]
        [(Cons?⟨τ⟩ l) (add1 (length⟨τ⟩ (Rest⟨τ⟩ l)))]))⟩)
```

when we could instead write

```
(define length
  (lambda (l)
    (cond
      [(empty? l) 0]
      [(cons? l) (add1 (length (rest l)))])))
```

As computer scientists, we should ask: Is it possible for a programming environment to convert the latter into the former? That is, can the environment *automatically fill in* the types necessary for the former? This would be the best of both worlds, because the programmer would avoid the trouble of all this typing (in both senses), while getting the benefits of type checking in a polymorphic manner (which has the advantage of needing to write fewer almost-identical procedures).

While this would be nice, it also seems nearly magical. It's hard enough for a human to figure out all the types that are necessary—that's why we cause type errors when we program!—that it seems well nigh impossible for a program to figure out our intent and fill in the types. Still, we should not despair too much. After all, modern programming environments do perform several convenient tasks (such as syntax highlighting) automatically. Maybe inserting type annotations could be one of those tasks.

Because this seems like an insurmountable challenge, let's try to make the problem easier. Let's ignore the polymorphism, and just focus on generating types for *monomorphic* programs (i.e., those that don't employ polymorphism). In fact, just to make life really simple, let's just consider a program that operates over numbers, such as factorial.

2 Example: Factorial

Suppose we're given the following program:

```
(define fact
  (lambda (n)
    (cond
```

[(*zero? n*) 1]
[true (∗ *n* (*fact* (*sub1 n*)))]))))

We've purposely written true instead of **else** for reasons we'll soon see. It should be clear that using true doesn't affect the meaning of the program (in general, **else** is just a more readable way of writing true).

If we were asked to determine the type of this function and had never seen it before, our reasoning might proceed roughly along these lines. First, we would annotate each expression:

(**define** *fact*
 $\boxed{1}$(**lambda** (*n*)
  $\boxed{2}$(**cond**
   [$\boxed{3}$(*zero? n*) $\boxed{4}$1]
   [$\boxed{5}$true $\boxed{6}$(∗ *n* $\boxed{7}$(*fact* $\boxed{8}$(*sub1 n*)))])))

We would now reason as follows. We'll use the notation $[\![\cdot]\!]$ to mean the type of the expression within the brackets.

- The type $\boxed{1}$[1] is clearly a function type (since the expression is a **lambda**). The function's argument type is that of *n*, and it computes a value with the type $\boxed{2}$. In other words,
$$[\![\boxed{1}]\!] = [\![n]\!] \rightarrow [\![\boxed{2}]\!]$$
- Because $\boxed{2}$ is a conditional, we want to ensure the following:
  - The first and second conditional expressions evaluate to boolean values. That is, we would like the following to hold:
$$[\![\boxed{3}]\!] = \mathsf{boolean}$$
$$[\![\boxed{5}]\!] = \mathsf{boolean}$$
  - We would like both branches of the conditional to evaluate to a value of the same type, so we can assign a meaningful type to the entire conditional expression:
$$[\![\boxed{2}]\!] = [\![\boxed{4}]\!] = [\![\boxed{6}]\!]$$
- What is the type of $\boxed{3}$? We have a constraint on what it can be:
$$[\![zero?]\!] = [\![n]\!] \rightarrow [\![\boxed{3}]\!]$$
  Because we know the type of *zero?*, we know that the right-hand-side of the above equality must be:
$$[\![n]\!] \rightarrow [\![\boxed{3}]\!] = \mathsf{number} \rightarrow \mathsf{boolean}$$
  which immediately tells us that $[\![n]\!] = \mathsf{number}$.

The first response in the **cond** tells us that $[\![\boxed{4}]\!] = \mathsf{number}$, which immediately resolves the type of $\boxed{2}$ and determines the type of $\boxed{1}$ in atomic terms. Therefore, the type of *fact* must be number → number. However, it's worthwhile to continue with this process as an illustration:

- We have a constraint on the type of $\boxed{6}$: it must be the same as the result type of multiplication. Concretely,
$$[\![n]\!] \times [\![\boxed{7}]\!] \rightarrow [\![\boxed{6}]\!] = \mathsf{number} \times \mathsf{number} \rightarrow \mathsf{number}$$
- The type of $\boxed{7}$ must be whatever *fact* returns, while $\boxed{8}$ must be the type that *fact* consumes:
$$[\![\boxed{1}]\!] = [\![\boxed{8}]\!] \rightarrow [\![\boxed{7}]\!]$$
- Finally, the type of $\boxed{8}$ must be the return type of *sub1*:
$$[\![sub1]\!] = [\![n]\!] \rightarrow [\![\boxed{8}]\!] = \mathsf{number} \rightarrow \mathsf{number}$$

[1] Whenever we say "the type $\boxed{n}$", we mean the type of the expression labeled by $\boxed{n}$; similarly for other abuses of language.

3 Example: Numeric-List Length

Now let's look at a second example:

```
(define nlength
  (lambda (l)
    (cond
      [(nempty? l) 0]
      [(ncons? l)  (add1 (nlength (nrest l)))])))
```

First, we annotate it:

```
(define nlength
  [1](lambda (l)
    [2](cond
      [[3](nempty? l) [4]0]
      [[5](ncons? l)  [6](add1 [7](nlength [8](nrest l)))])))
```

We can begin by deriving the following constraints:

$$[\![\boxed{1}]\!] = [\![l]\!] \rightarrow [\![\boxed{2}]\!]$$

$$[\![\boxed{2}]\!] = [\![\boxed{4}]\!] = [\![\boxed{6}]\!]$$

$$[\![\boxed{3}]\!] = [\![\boxed{5}]\!] = \mathsf{boolean}$$

Because $\boxed{3}$ and $\boxed{5}$ are each applications, we derive some constraints from them:

$$[\![nempty?]\!] = [\![l]\!] \rightarrow [\![\boxed{3}]\!] = \mathsf{nlist} \rightarrow \mathsf{boolean}$$

$$[\![ncons?]\!] = [\![l]\!] \rightarrow [\![\boxed{5}]\!] = \mathsf{nlist} \rightarrow \mathsf{boolean}$$

The first conditional's response is not very interesting:

$$[\![\boxed{4}]\!] = [\![0]\!]^2 = \mathsf{number}$$

Finally, we get to the second conditional's response, which yields several constraints:

$$[\![add1]\!] = [\![\boxed{7}]\!] \rightarrow [\![\boxed{6}]\!] = \mathsf{number} \rightarrow \mathsf{number}$$

$$[\![\boxed{1}]\!] = [\![\boxed{8}]\!] \rightarrow [\![\boxed{7}]\!]$$

$$[\![nrest]\!] = [\![l]\!] \rightarrow [\![\boxed{8}]\!] = \mathsf{nlist} \rightarrow \mathsf{nlist}$$

Notice that in the first and third set of constraints above, because the program applies a primitive, we can generate an extra constraint which is the type of the primitive itself. In the second set, because the function is user-defined, we cannot generate any other meaningful constraint just by looking at that one expression.

Solving all these constraints, it's easy to see both that the constraints are compatible with one another, and that each expression receives a monomorphic type. In particular, the type of $\boxed{1}$ is $\mathsf{nlist} \rightarrow \mathsf{number}$, which is therefore the type of *nlength* also (and proves to be compatible with the use of *nlength* in expression $\boxed{7}$).

[2] Note that the 0 inside the $[\![\cdot]\!]$ is an expression itself, not a number labeling an expression.

4 Formalizing Constraint Generation

What we've done so far is extremely informal, and purely to generate some intuition for constraints. Now we formalize this process.

Constraints relate different portions of the program by determining how they should be compatible for the program to execute without error. Consequently, a single program point may result in multiple constraints. Each set of constraints represents a "wish list" about that particular point in the program. Consequently, a program may lead to contradictory constraints; hopefully we will be able to find these later. One slightly confusing aspect of constraints is that we write them to look like equations, but they reflect what we *hope* will be true, not what we *know* is true. Specifically, they represent what suffices for safe program execution.[3]

For each node n in the program's abstract syntax tree, we introduce a variable of the form $[\![n]\!]$. That is, if the program has the form (*foo* 1 2), we would want to introduce variables for 1, 2 and (*foo* 1 2). Because abstract syntax tree nodes are unwieldy to write down explicitly, we will introduce some ambiguity by associating the node with the expression at that node.[4] We will denote the variables by $[\![\cdot]\!]$, so the three variables corresponding to the above nodes would be $[\![1]\!]$, $[\![2]\!]$ and $[\![(foo\ 1\ 2)]\!]$. (Some of these variables obviously have quite funny names, but obviously, in practice, we just associate unique tokens with each node, as we did in the examples above.)

Each expression type generates different constraints. We present below a table that relates the type of expression at a node to the (set of) constraints that to generate at that node. Remember to always read $[\![\cdot]\!]$ as "the type of the expression" (within the brackets):

Expression at Node	Generated Constraints
n, where *n* is a numeral	$[\![n]\!] = \mathsf{number}$
true	$[\![\mathsf{true}]\!] = \mathsf{boolean}$
false	$[\![\mathsf{false}]\!] = \mathsf{boolean}$
(*add1 e*)	$[\![(add1\ e)]\!] = [\![e]\!] = \mathsf{number}$
(+ *e1 e2*)	$[\![(+\ e1\ e2)]\!]=\mathsf{number} \quad [\![e1]\!] = \mathsf{number} \quad [\![e2]\!] = \mathsf{number}$
(*zero? e*)	$[\![(zero?\ e)]\!] = \mathsf{boolean} \quad [\![e]\!] = \mathsf{number}$
(*ncons e1 e2*)	$[\![(ncons\ e1\ e2)]\!]=\mathsf{nlist} \quad [\![e1]\!] = \mathsf{number} \quad [\![e2]\!] = \mathsf{nlist}$
(*nfirst e*)	$[\![(nfirst\ e)]\!]=\mathsf{number} \quad [\![e]\!] = \mathsf{nlist}$
(*nrest e*)	$[\![(nrest\ e)]\!]=\mathsf{nlist} \quad [\![e]\!] = \mathsf{nlist}$
(*nempty? e*)	$[\![(nempty?\ e)]\!]=\mathsf{boolean} \quad [\![e]\!] = \mathsf{nlist}$
nempty	$[\![nempty]\!]=\mathsf{nlist}$
(**lambda** (*x*) *b*)	$[\![(\textbf{lambda}\ (x)\ b)]\!]= [\![x]\!]\rightarrow[\![b]\!]$
(*f a*)	$[\![f]\!] = [\![a]\!]\rightarrow[\![(f\ a)]\!]$

Notice that in the two prior examples, we did not create new node numbers for those expressions that consisted of just a program identifier; similarly, we have not given a rule for identifiers. We *could* have done this, for consistency, but it would have just created more unnecessary variables.

5 Errors

Here's an erroneous program:

```
(define nlsum
  (lambda (l)
    (cond
      [(nempty? l) 0]
      [(ncons? l)  (+ (nrest l)
                      (nlsum (nrest l)))])))
```

[3] We use the term "suffices" advisedly: these constraints are sufficient but not necessary. They may reject some programs that might have run without error had the type system not intervened. This is inherent in the desire to statically approximate dynamic behavior: the Halting Problem is an insurmountable obstacle. An important constraint on good type system design is to maximize the set of legal problems while still not permitting errors: *balancing programmer liberty with execution safety*.

[4] **Puzzle**: Why does this introduce ambiguity?

Can you spot the problem?

First, we'll annotate the sub-expressions:

```
(define nlsum
  [1](lambda (l)
    [2](cond
      [[3](nempty? l) [4]0]
      [[5](ncons? l) [6](+ [7](nrest l)
                          [8](nlsum [9](nrest l)))])))
```

Generating constraints as usual, we get the following (amongst others):

$$[\![\boxed{8}]\!] = \mathsf{number}$$

from the type of addition, and

$$[\![\boxed{9}]\!] = \mathsf{nlist}$$

from the type of *nrest*. Consequently, it appears we can infer that the value bound to *nlsum* has the type $\mathsf{nlist} \rightarrow \mathsf{number}$. Indeed, this is indeed the type we expect for this procedure.

It is clear, however, that we have not looked as closely as we can at the set of constraints: indeed, most constraints haven't been examined at all (even if they have been generated). Before we can conclude a type for any expressions, we must make sure that *all the constraints are consistent*! That isn't the case for this set of constraints. In particular, we have

$$[\![\boxed{7}]\!] = \mathsf{nlist}$$

from the type of *nrest*, while

$$[\![\boxed{7}]\!] = \mathsf{number}$$

from the type of $+$. Indeed, the latter is the type we want: the nlist only materializes because of the faulty use of *nrest*. Had the programmer used *nfirst* instead of *nrest* in the left-hand-side of the addition, the entire program would have checked correctly. Instead, the type checker prints an error indicating that there is a *type conflict*: the expression (*nrest l*) is being expected to have both the type number and the type nlist. Because these are not compatible types, the type "checker" halts with an error.[5]

6 Example: Using First-Class Functions

We will consider one final example of constraint generation, to show that the process scales in the presence of functions as arguments. Consider the following program:

```
(define nmap
  [1](lambda (f l)
    [2](cond
      [[3](nempty? l) [4]nempty]
      [[5](ncons? l) [6](ncons [7](f [8](nfirst l))
                               [9](nmap f [10](nrest l)))])))
```

This program generates the following constraints:

$$[\![\boxed{1}]\!] = [\![f]\!] \times [\![l]\!] \rightarrow [\![\boxed{2}]\!]$$

[5] We use quotes because the checker has, in some sense, disappeared. Instead of checking types annotated by the programmer, the type system now tries to fill in the programmer's annotations. If it succeeds, it can do so only by respecting the types of operations, so there is no checking left to be done; if it fails, the type inference engine halts with an error.

We get the usual constraints about boolean conditional tests and the type equality of the branches (both must be of type nlist due to the first response). From the second response, we derive

$$[\![ncons]\!] = [\![\boxed{7}]\!] \times [\![\boxed{9}]\!] \rightarrow [\![\boxed{6}]\!] = \mathsf{number} \times \mathsf{numlist} \rightarrow \mathsf{numlist}$$

The most interesting constraint is this one:

$$[\![f]\!] = [\![\boxed{8}]\!] \rightarrow [\![\boxed{7}]\!]$$

and with that, we have captured first-class functions! The only constraint imposed by this type system is that all functions passed as parameters meet this type reqirement.

Continuing, we obtain the following three constraints also:

$$[\![nfirst]\!] = [\![l]\!] \rightarrow [\![\boxed{8}]\!] = \mathsf{nlist} \rightarrow \mathsf{number}$$

$$[\![nmap]\!] = [\![f]\!] \times [\![\boxed{10}]\!] \rightarrow [\![\boxed{9}]\!]$$

$$[\![nrest]\!] = [\![l]\!] \rightarrow [\![\boxed{10}]\!] = \mathsf{nlist} \rightarrow \mathsf{nlist}$$

Since l is of type nlist, we can substitute and solve to learn that f has type $\mathsf{num} \rightarrow \mathsf{num}$. Conseqently, *nmap* has type

$$(\mathsf{number} \rightarrow \mathsf{number}) \times \mathsf{nlist} \rightarrow \mathsf{nlist}$$

which is the type we would desire and expect!