Exploring Languages with Interpreters and Functional Programming
Chapter 24

H. Conrad Cunningham

2 August 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of August 2018 is a recent version of Firefox from Mozilla.

24 Type Inference

24.1 Chapter Introduction

The goal of this chapter is to show how type inference works. It presents the topic using an equational reasoning technique.

This chapter depends upon the reader understanding Haskell polymorphic, higher-order function concepts (e.g. from studying Chapters 13-17), but it is otherwise independent of other chapters. No subsequent chapter depends explicitly upon this content.

24.2 Motivation

How can we deduce the type of a Haskell expression?

To get the general idea, let's look at a few examples.

Note: The discussion here is correct for monomorphic functions, but it is a bit simplistic for polymorphic functions. However, it should be of assistance in understanding how types are assigned to Haskell expressions.

24.3 Example: Functional Composition

Expressed in prefix form, functional composition can be defined with the equation:

```haskell
(.) f g x = f (g x)
```

We begin the process of type inference by assigning types to the parameter names and to the function's defining expression (i.e. its result). We introduce new type names `t1`, `t2`, `t3` and `t4` for the components of `(.)` as follows:

```haskell
      f :: t1    -- parameter 1 of (.)
      g :: t2    -- parameter 2 of (.)
      x :: t3    -- parameter 3 of (.)
f (g x) :: t4    -- defining expression for (.)
```

The type of `(.)` is therefore given by:

```haskell
(.) :: t1 -> t2 -> t3 -> t4
```

We are not finished because there are certain relationships among the new types that must be taken into account. To see what these relationships are, we use the following inference rules.

- **Application rule:** If `f x :: t`, then we can deduce `x :: t'` and `f :: t' -> t` for some new type `t'`.

- **Equality rule:** If both `x :: t` and `x :: t'` for some variable `x`, then we can deduce `t = t'`.
- **Function rule:** If `(t -> u) = (t' -> u')`, then we can deduce `t = t'` and `u = u'`.

Using the *application rule* on `f (g x) :: t4`, we introduce a new type `t5` such that:

```
g x :: t5
  f :: t5 -> t4
```

Using the *application rule* for `g x :: t5`, we introduce another new type `t6` such that:

```
x :: t6
g :: t6 -> t5
```

Using the *equality rule* on the two types deduced for each of `f`, `g`, and `x`, respectively, we get the following identities:

```
t1 = (t5 -> t4)     -- f
t2 = (t6 -> t5)     -- g
t3 = t6             -- x
```

For function `(.)`, we thus deduce the type signature:

```
(.) :: (t5 -> t4) -> (t6 -> t5) -> t6 -> t4
```

If we replace the type names by Haskell generic type variables that follow the usual naming convention, we get:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

24.4 Example: Multiple Use of Polymorphic Function (`fst`)

Now let's consider the function definition:

```
f x y = fst x + fst y
```

Note that the names `(+)` and `fst` occur on the right side of the definition, but do not occur on the left.

From the Haskell Prelude, we can see that:

```
(+) :: Num a => a -> a -> a
fst :: (a, b) -> a
```

The `Num a` context contrains the polymorphism on type variable `a`.

We must be careful. The two occurrences of the polymorphic function `fst` in the definition for `f` need not bind the type variables `a` and `b` to the same concrete types. For example, consider the expression:

```
fst (2, True) + fst (1, "hello")
```

This expression is well-typed despite the fact that the first occurrence of `fst` has the type

```
Num a => (a,Bool) -> a
```

and the second occurrence has type

```
Num a => (a, [Char]) -> a
```

Furthermore, the two occurrences of the type variable `a` are not, in general, required to bind to the same type. (However, as we will see, they do in this expression because of the addition operation.)

To handle the situation with the multiple applications of `fst`, we use the following rule.

- **Polymorphic use rule:** If a polymorphic function is applied multiple times in an expression, then the type of each occurrence is determined independently, with each assigned new type variables.

Following the *polymorphic use* rule, we rewrite the definition of `f` in the form

```
f x y = fst1 x + fst2 y
```

and assume two different instantiations of the generic type of `fst`:

```
fst1 :: (u1, u2) -> u1
fst2 :: (v1, v2) -> v1
```

After making the above transformation, we proceed by assigning types to the parameters and definition of `f`, introducing three new types:

```
              x :: t1    -- parameter 1 of f
              y :: t2    -- parameter 2 of f
fst1 x + fst2 y :: t3    -- defining expression for f
```

Thus we have the following type for `f`:

```
f :: t1 -> t2 -> t3
```

Now we can rewrite the defining expression for `f` fully in prefix form to get:

```
(+) (fst1 x) (fst2 y)
```

Then, using the *application rule* on the above expression, we deduce:

```
    (fst2 y) :: t4
(+) (fst1 x) :: t4 -> t3
```

Using the *application rule* on `(fst2 y) :: t4`, we get:

```
    y :: t5
fst2 :: t5 -> t4
```

Similarly, using the *application rule* on `(+) (fst1 x) :: t4 -> t3`, we get:

```
(fst1 x) :: t6
     (+) :: t6 -> t4 -> t3
```

Going further and applying the *application rule* to `(fst1 x) :: t6`, we deduce:

```
   x :: t7
fst1 :: t7 -> t6
```

Now we have introduced types for all the symbols appearing in the definition of function `f`. We begin simplification by using the *equality rule* for `x`, `y`, `fst1`, `fst2`, and `(+)`, respectively. We thus deduce the type equations:

```
                       t1 = t7                 -- x
                       t2 = t5                 -- y
     ((u1, u2) -> u1)  =  (t7 -> t6)           -- fst1
     ((v1, v2) -> v1)  =  (t5 -> t4)           -- fst2
(Num a => a -> a -> a) = (t6 -> t4 -> t3)      -- (+)
```

Now, using the function rule on the last three equations above, we derive:

```
t7 = (u1, u2)
t6 = u1

t5 = (v1, v2)
t4 = v1

t3 = t4 = t6 = v1 = u1 = (Num a => a)
```

We had assigned type `f :: t1 -> t2 -> t3` originally. Substituting from the above, we deduce the following type:

```
f :: Num a => (a, u2) -> (a, v2) -> a
```

Finally, we can replace the type names `u2` and `v2` by Haskell generic type variables that follow the usual naming convention. We get the following inferred type for function `f`:

```
f :: Num a => (a, b) -> (a, c) -> a
```

24.5 Example: Fixpoint (`fix`)

For this example, consider the definition:

```
fix f = f (fix f)
```

To deduce a type for `fix`, we proceed as before and introduce types for the parameters and defining expression of `f`:

```
        f :: t1    -- parameter of fix
f (fix f) :: t2    -- defining expression for fix
```

Thus, `fix` has the type:

```
fix :: t1 -> t2
```

Using the *application rule* on the expression `f (fix f)`, we obtain:

```
(fix f) :: t3
      f :: t3 -> t2
```

Then using the *application rule* on the expression `fix f`, we get:

```
    f :: t4
  fix :: t4 -> t3
```

Using the *equality rule* on `f` and `fix`, we deduce:

```
t1 = t4 = (t3 -> t2)        -- f
(t1 -> t2) = (t4 -> t3)     -- fix
```

Then, using the *function rule* on the second equation, we obtain the identities:

```
t1 = t4
t2 = t3
```

Since `fix :: t1 -> t3`, we derive the type:

```
fix :: (t3 -> t3) -> t3
```

If we replace `t3` by a Haskell generic type variable that follows the usual naming convention, we get the following inferred type for `fix`:

```
fix :: (a -> a) -> a
```

24.6 Example: Incorrect Typing (selfapply)

Finally, let us consider an example in which the typing is wrong. Let us define `selfapply` as follows:

```
selfapply f = f f
```

Proceeding as in the previous examples, we introduce new types for the parameters and defining expression of `f`:

```
  f :: t1    -- parameter of selfapply
f f :: t2    -- defining expression for selfapply
```

Thus we have the type:

```
selfapply :: t1 -> t2
```

Using the *application rule* on `f f`, we get:

```
f :: t3
f :: t3 -> t2
```

But the *equality rule* for `f` tells us that:

```
t1 = t3 = (t3 -> t2)
```

or just

```
t1 = (t1 -> t2)
```

However, the equation `t1 = (t1 -> t2)` does not possess a solution for `t1` and the definition of `selfapply` is thus rejected by the type checker.

24.7 Other Aspects of Type Inference

Haskell function definitions must also conform to the following rules.

- **Guard rule:** Each guard must be an expression of type `Bool`.
- **Tuple rule:** The type of a tuple of elements is the tuple of their respective types.

24.8 What Next?

This chapter is largely independent of other chapters. No subsequent chapter depends explicitly upon this content.

24.9 Exercises

TODO

24.10 Acknowledgements

In Spring 2017, I adapted and revised this chapter from my previous HTML notes on this topic [Cunningham 2017]. (These were supplementary notes for a course based on [Cunningham 2014].) I based the previous notes on the presentations in:

- Section 2.8 of the book *Introduction to Functional Programming* (First Edition) by Richard Bird and Philip Wadler [Bird 1988].
- Chapter 9 of the book *Haskell: The Craft of Functional Programming* (First Edition) by Simon Thompson [Thompson 1996]

I thank MS student Hongmei Gao for helping me prepare the first version of the previous notes in Spring 2000.

In Summer 2018, I incorporated this work as new Chapter 24, Type Inference, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

24.11 References

[**Bird 1988**]**:** Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[**Cunningham 2017**]**:** H. Conrad Cunningham. *Type Inference*, 2000-17.

[**Thompson 1996**]**:** Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

24.12 Terms and Concepts

Type inference, function, polymorphism, type variable, function composition, fixpoint, application rule, equality rule, function rule, polymorphic use rule, guard rule, tuple rule.