

C311 – New Type Inference Study Guide

Corky Cartwright

Produced: November 26, 2007

1 Synopsis of Implicitly Polymorphic Jam

The syntax of (Implicitly) Polymorphic Jam is a restriction of the syntax of untyped Jam. Every legal Polymorphic Jam program is also a legal untyped Jam Program. But the converse is false, because there may not be a valid typing for a given untyped Jam program.

1.1 Abstract Syntax

The following grammar describes the abstract syntax of Polymorphic Jam. Each clause in the grammar corresponds directly to a node in the abstract syntax tree. The **let** construction has been limited to a single binding for the sake of notational simplicity. It is straightforward to generalize the rule to multiple bindings (with mutual recursion). Note that **let** is *recursive*.

```
M ::= M (M...M) | P (M...M) | if M then M else M | let x := M in M
      | V
V ::= map x...x to M | x | n | true | false | null
n ::= 1 | 2 | ...
P ::= cons | first | rest | null? | cons? | + | - | / | * | = | < |
<= | <- | + | - | ~ | ref | !
x ::= variable names
```

In the preceding grammar, unary and binary operators are treated exactly like primitive functions.

Monomorphic types in the language are defined by τ , below. Polymorphic types are defined by σ . The \rightarrow corresponds to a function type, whose inputs are to the left of the arrow and whose output is to the right of the arrow.

```
 $\sigma ::= \forall \alpha_1 \dots \alpha_n. \tau$ 
 $\tau ::= \text{int} | \text{bool} | \text{unit} | \tau_1 \times \dots \times \tau_n \rightarrow \tau | \alpha | \text{list } \tau | \text{ref } \tau$ 
 $\alpha ::= \text{type variable names}$ 
```

1.2 Type Checking Rules

In the following rules, the notation $\Gamma[x_1 : \tau_1, \dots, x_n : \tau_n]$ means the $(\Gamma \setminus \{x_1, \dots, x_n\}) \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ and Γ' abbreviates $\Gamma[x_1 : \tau'_1, \dots, x_n : \tau'_n]$. Note that $\Gamma \setminus \{x_1, \dots, x_n\}$ means Γ less the type assertions (if any) for $\{x_1, \dots, x_n\}$.

$$\frac{\Gamma[x_1 : \tau_1, \dots, x_n : \tau_n] \vdash M : \tau}{\Gamma \vdash \text{map } x_1 \dots x_n \text{ to } M : \tau_1 \times \dots \times \tau_n \rightarrow \tau} [\text{abs}]$$

$$\frac{\Gamma \vdash M : \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Gamma \vdash M_1 : \tau_1 \quad \dots \quad \Gamma \vdash M_n : \tau_n}{\Gamma \vdash M (M_1 \dots M_n) : \tau} [\text{app}]$$

$$\frac{\Gamma \vdash M_1 : \text{bool} \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau} [\text{if}]$$

Note that there are two rules for **let** expressions. The **[letmono]** rule corresponds to the **let** rule of Typed Jam; it places no restriction on the form of the right-hand side M_1 of the **let** binding. The **[letpoly]** rule generalizes the free type variables (not occurring in the type environment Γ) in the type inferred for the right-hand-side of a **let** binding – provided that the right-hand-side M_1 is a *syntactic* value: a *constant* like **null** or **cons**, a **map** expression, or a variable. Syntactic values are expressions whose evaluation is trivial, excluding evaluations that allocate storage.

$$\frac{\Gamma[x : \tau] \vdash x : \tau \quad \frac{\Gamma' \vdash M_1 : \tau'_1 \quad \dots \quad \Gamma' \vdash M_n : \tau'_n \quad \Gamma' \vdash M : \tau}{\Gamma \vdash \text{let } x_1 := M_1; \dots; x_n := M_n; \text{in } M : \tau} [\text{letmono}]}{\Gamma \vdash M_1 : \tau'_1 \quad \dots \quad \Gamma' \vdash M_n : \tau'_n \quad \Gamma[x_1 : C_{M_1}(\tau'_1, \Gamma), \dots, x_n : C_{M_n}(\tau'_n, \Gamma)] \vdash M : \tau} [\text{letpoly}]$$

$$\Gamma[x : \forall \alpha_1, \dots, \alpha_n. \tau] \vdash x : O(\forall \alpha_1, \dots, \alpha_n. \tau, \tau_1, \dots, \tau_n)$$

The functions $O(\cdot, \cdot)$ and $C(\cdot, \cdot)$ are the keys to polymorphism. Here is how $C(\cdot, \cdot)$ is defined:

$$C_V(\tau, \Gamma) := \forall \{ \text{FTV}(\tau) - \text{FTV}(\Gamma) \}. \tau$$

$$C_N(\tau, \Gamma) := \tau$$

where V is a syntactic value, N is an expression that is not a syntactic value, and $\text{FTV}(\alpha)$ means the “free type variables in the expression (or type environment) α ”.

When closing over a type, you must find all of the free variables in τ that are not free in any of the types in the environment Γ . Then, build a polymorphic type by quantifying τ over all of those type variables.

To open a polymorphic type

$$\forall \alpha_1, \dots, \alpha_n. \tau,$$

substitute *any* type terms τ_1, \dots, τ_n for the quantified type variables $\alpha_1, \dots, \alpha_n$:

$$O(\forall \alpha_1, \dots, \alpha_n. \tau, \tau_1, \dots, \tau_n) = \tau_{[\alpha_1 := \tau_1, \dots, \alpha_n := \tau_n]}$$

which creates a monomorphic type from a polymorphic type. For example,

$$O(\forall \alpha. \alpha \rightarrow \alpha, \tau) = \tau \rightarrow \tau$$

1.3 Types of Primitives

The following table gives types for all of the primitive constants, functions, and operators. The symbol n stands for any integer constant. Programs are type checked starting with a primitive type environment, denoted Γ_0 consisting of this table.

<code>true</code>	<code>bool</code>	<code>+</code>	<code>int × int → int</code>
<code>false</code>	<code>bool</code>	<code>-</code>	<code>int × int → int</code>
<code>n</code>	<code>int</code>	<code>*</code>	<code>int × int → int</code>
<code>null</code>	$\forall \alpha. \text{list } \alpha$	<code>/</code>	<code>int × int → int</code>
<code>cons</code>	$\forall \alpha. \alpha \times \text{list } \alpha \rightarrow \text{list } \alpha$	<code><</code>	<code>int × int → bool</code>
<code>first</code>	$\forall \alpha. \text{list } \alpha \rightarrow \alpha$	<code>></code>	<code>int × int → bool</code>
<code>rest</code>	$\forall \alpha. \text{list } \alpha \rightarrow \text{list } \alpha$	<code><=</code>	<code>int × int → bool</code>
<code>cons?</code>	$\forall \alpha. \text{list } \alpha \rightarrow \text{bool}$	<code>>=</code>	<code>int × int → bool</code>
<code>null?</code>	$\forall \alpha. \text{list } \alpha \rightarrow \text{bool}$	(unary) <code>-</code>	<code>int → int</code>
<code>=</code>	$\forall \alpha. \alpha \times \alpha \rightarrow \text{bool}$	(unary) <code>+</code>	<code>int → int</code>
<code>!=</code>	$\forall \alpha. \alpha \times \alpha \rightarrow \text{bool}$	(unary) <code>~</code>	<code>bool → bool</code>
		<code><-</code>	$\forall \alpha. \text{ref } \alpha \times \alpha \rightarrow \text{unit}$
		<code>ref</code>	$\forall \alpha. \alpha \rightarrow \text{ref } \alpha$
		<code>!</code>	$\forall \alpha. \text{ref } \alpha \rightarrow \alpha$

1.4 Typed Jam

The Typed Jam language used in Assignment 5 (absent the explicit type information embedded in program text) can be formalized as a subset of Polymorphic Jam. For the purposes of these exercises, Typed Jam is simply Polymorphic Jam less the **letpoly** inference rule which prevents it from inferring polymorphic types for program-defined functions.

2 Exercises

Task 1: Prove the following type judgements for Typed Jam or explain why they are not provable:

1. $\Gamma_0 \vdash (\text{map } x \text{ to } x(10))(\text{map } x \text{ to } x) : \text{int}$
2. $\Gamma_0 \vdash \text{let fact} := \text{map } n \text{ to if } n=0 \text{ then } 1 \text{ else } n*(\text{fact}(n-1));$
 $\text{in fact}(10)+\text{fact}(0) : \text{int}$
3. $\Gamma_0 \vdash (\text{map } x \text{ to } 1 + (1/x))(0) : \text{int}$
4. $\Gamma_0 \vdash (\text{map } x \text{ to } x) (\text{map } y \text{ to } y) : (\text{int} \rightarrow \text{int})$
5. $\Gamma_0 \vdash \text{let id} := \text{map } x \text{ to } x; \text{ in id}(\text{id}) : (\text{int} \rightarrow \text{int})$

Task 2: Are the following Polymorphic Jam programs typable? Justify your answer either by giving a proof tree (constructed using the inference rules for PolyJam) or by showing a conflict in the type constraints generated by matching the inference rules against the program text.

1.

```
let listMap := map f,l to
    if null?(l) then null
    else cons(f(first(l)), listMap(f, rest(l)))
in listMap(first,null);
```
2.

```
let length := map l to if null?(l) then 0
    else 1 + length(rest(l));
    l := cons(cons(1,null),cons(cons(2,cons(3,null)),null));
in length(l)+length(first(l))
```

Task 3: Give a simple example of an untyped Jam expression that is not typable in Typed Jam but is typable in Polymorphic Jam.

3 Solutions to Selected Exercises

Task 1 : The first four expressions are typable in Typed Jam, but the fifth is not.

1. *Tree 1:*

$$\frac{\frac{\Gamma_0[f:\text{int} \rightarrow \text{int}] \vdash 10:\text{int} \quad \Gamma_0[f:\text{int} \rightarrow \text{int}] \vdash f:\text{int} \rightarrow \text{int}}{\Gamma_0[f:\text{int} \rightarrow \text{int}] \vdash f(10):\text{int}}[\text{app}]}{\Gamma_0 \vdash \text{map } f \text{ to } f(10):(\text{int} \rightarrow \text{int}) \rightarrow \text{int}}[\text{abs}]$$

- Tree 2:*

$$\text{Tree 1} \quad \frac{\Gamma_0[x:\text{int}] \vdash x:\text{int}}{\Gamma_0 \vdash \text{map } x \text{ to } x:\text{int} \rightarrow \text{int}}[\text{abs}]$$

$$\frac{\Gamma_0 \vdash \text{map } x \text{ to } x:\text{int} \rightarrow \text{int}}{\Gamma_0 \vdash (\text{map } f \text{ to } f(10))(\text{map } x \text{ to } x):\text{int}}[\text{app}]$$

2. Type Inference Proof Omitted.

3. *Tree 1:*

$$\frac{\Gamma_0[x:\text{int}] \vdash /:\text{int} \times \text{int} \rightarrow \text{int} \quad \Gamma_0[x:\text{int}] \vdash 1:\text{int} \quad \Gamma_0[x:\text{int}] \vdash x:\text{int}}{\Gamma_0[x:\text{int}] \vdash 1/x:\text{int}}[\text{app}]$$

Tree 2:

$$\frac{\Gamma_0[x:\text{int}] \vdash +:\text{int} \times \text{int} \rightarrow \text{int} \quad \Gamma_0[x:\text{int}] \vdash 1:\text{int} \quad \text{Tree 1}[\text{app}]}{\Gamma_0[x:\text{int}] \vdash (1 + (1/x)):\text{int}}[\text{app}]$$

$$\frac{\Gamma_0[x:\text{int}] \vdash (1 + (1/x)):\text{int}}{\Gamma_0 \vdash (\text{map } x \text{ to } 1 + (1/x)):\text{int} \rightarrow \text{int}}[\text{abs}]$$

Tree 3:

$$\frac{\text{Tree 2} \quad \Gamma_0 \vdash 0:\text{int}}{\Gamma_0 \vdash (\text{map } x \text{ to } 1 + (1/x))(0):\text{int}}[\text{app}]$$

4. *Tree 1:*

$$\frac{\Gamma_0[x:\text{int} \rightarrow \text{int}] \vdash x:\text{int} \rightarrow \text{int}}{\Gamma_0 \vdash (\text{map } x \text{ to } x):(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})}[\text{abs}]$$

Tree 2:

$$\frac{\Gamma_0[y:\text{int}] \vdash y:\text{int}}{\Gamma_0 \vdash (\text{map } y \text{ to } y):\text{int} \rightarrow \text{int}}[\text{abs}]$$

Tree 3:

$$\frac{\text{Tree 1} \quad \text{Tree 2}}{\Gamma_0 \vdash (\text{map } x \text{ to } x)(\text{map } y \text{ to } y):\text{int} \rightarrow \text{int}}[\text{app}]$$

5. This example is almost identical to the previous one, but the identity function `id` is defined only once in a `let` binding and then applied to itself. Since Typed Jam does not support polymorphism, we can only assign one typing to `id`. But we needed two different typings for the identity in the preceding example, so we cannot type this program.

Task 2: Both programs are typable in Polymorphic Jam. In fact the first program is typable in Typed Jam because the `length` function is only applied to one type of list. Hence the `letmono` rule can be used to type the `let` expression in this program instead of the more general `letpoly` rule.

1. Type Inference Proof Omitted.

2. Let Γ_1 abbreviate $\Gamma_0[\text{length} : \text{list } \alpha \rightarrow \text{int}, 1 : \text{list list int}]$;
 let Γ_2 abbreviate $\Gamma_0[\text{length} : \forall \alpha. (\text{list } \alpha \rightarrow \text{int}), 1 : \text{list list int}]$;
 and let Γ_3 abbreviate $\Gamma_1[1 : \text{list } \alpha]$.

Tree 1:

$$\frac{\Gamma_3 \vdash \text{rest} : \text{list } \alpha \rightarrow \text{list } \alpha \quad \Gamma_3 \vdash 1 : \text{list } \alpha}{\Gamma_3 \vdash \text{rest}(1) : \text{list } \alpha}[\text{app}] \quad \Gamma_3 \vdash \text{length} : \text{list } \alpha \rightarrow \text{int}$$

$$\frac{\Gamma_3 \vdash \text{rest}(1) : \text{list } \alpha}{\Gamma_3 \vdash \text{length}(\text{rest}(1)) : \text{int}}[\text{app}]$$

Tree 2:

$$\frac{\Gamma_3 \vdash + : \text{int} \times \text{int} \rightarrow \text{int} \quad \Gamma_3 \vdash 1 : \text{int} \quad \text{Tree 1}}{\Gamma_3 \vdash 1 + \text{length}(\text{rest}(1)) : \text{int}} [\text{app}]$$

Tree 3:

$$\frac{\frac{\Gamma_3 \vdash \text{null?} : \text{list } \alpha \rightarrow \text{bool} \quad \Gamma_3 \vdash 1 : \text{list } \alpha}{\Gamma_3 \vdash \text{null?}(1) : \text{bool}} [\text{app}] \quad \Gamma_3 \vdash 0 : \text{int} \quad \text{Tree 2}}{\Gamma_3 \vdash \text{if } \text{null?}(1) \text{ then } 0 \text{ else } 1 + \text{length}(\text{rest}(1)) : \text{int}} [\text{if}]$$

$$\frac{\Gamma_1 \vdash \text{map } 1 \text{ to if } \text{null?}(1) \text{ then } 0 \text{ else } 1 + \text{length}(\text{rest}(1)) : \text{list } \alpha \rightarrow \text{int}}{\Gamma_1 \vdash \text{map } 1 \text{ to if } \text{null?}(1) \text{ then } 0 \text{ else } 1 + \text{length}(\text{rest}(1)) : \text{list } \alpha \rightarrow \text{int}} [\text{abs}]$$

Tree 4:

$$\frac{\Gamma_1 \vdash \text{cons} : \text{int} \times \text{list int} \rightarrow \text{list int} \quad \Gamma_1 \vdash 1 : \text{int} \quad \Gamma_1 \vdash \text{null} : \text{list int}}{\Gamma_1 \vdash \text{cons}(1, \text{null}) : \text{list int}} [\text{app}]$$

Tree 5:

$$\frac{\Gamma_1 \vdash \text{cons} : \text{int} \times \text{list int} \rightarrow \text{list int} \quad \Gamma_1 \vdash 3 : \text{int} \quad \Gamma_1 \vdash \text{null} : \text{list int}}{\Gamma_1 \vdash \text{cons}(3, \text{null}) : \text{list int}} [\text{app}]$$

Tree 6:

$$\frac{\Gamma_1 \vdash \text{cons} : \text{int} \times \text{list int} \rightarrow \text{list int} \quad \Gamma_1 \vdash 2 : \text{int} \quad \text{Tree 5}}{\Gamma_1 \vdash \text{cons}(2, \text{cons}(3, \text{null})) : \text{list int}} [\text{app}]$$

Tree 7:

$$\frac{\Gamma_1 \vdash \text{cons} : \text{list int} \times \text{list list int} \rightarrow \text{list list int} \quad \text{Tree 6} \quad \Gamma_1 \vdash \text{null} : \text{list list int}}{\Gamma_1 \vdash \text{cons}(\text{cons}(2, \text{cons}(3, \text{null})), \text{null}) : \text{list list int}} [\text{app}]$$

Tree 8:

$$\frac{\Gamma_1 \vdash \text{cons} : \text{list int} \times \text{list list int} \rightarrow \text{list list int} \quad \text{Tree 4} \quad \text{Tree 7}}{\Gamma_1 \vdash \text{cons}(\text{cons}(1, \text{null}), \text{cons}(\text{cons}(2, \text{cons}(3, \text{null})), \text{null})) : \text{list list int}} [\text{app}]$$

Tree 9:

$$\frac{\Gamma_2 \vdash \text{length} : \text{list int} \rightarrow \text{int} \quad \frac{\Gamma_2 \vdash \text{first} : \text{list list int} \rightarrow \text{list int} \quad \Gamma_2 \vdash 1 : \text{list list int}}{\Gamma_2 \vdash \text{first}(1) : \text{list int}} [\text{app}]}{\Gamma_2 \vdash \text{length}(\text{first}(1)) : \text{int}} [\text{app}]$$

Tree 10

$$\frac{\Gamma_2 \vdash + : \text{int} \times \text{int} \rightarrow \text{int} \quad \frac{\Gamma_2 \vdash \text{length} : \text{list list int} \rightarrow \text{int} \quad \Gamma_2 \vdash 1 : \text{list list int}}{\Gamma_2 \vdash \text{length}(1) : \text{int}} [\text{app}] \quad \text{Tree 9}}{\Gamma_2 \vdash \text{length}(1) + \text{length}(\text{first}(1)) : \text{int}} [\text{app}]$$

Tree 11

$$\frac{\text{Tree 3} \quad \text{Tree 8} \quad \text{Tree 10}}{\Gamma_0 \vdash \text{let length} := \text{map } 1 \text{ to} \\ \text{if } \text{null?}(1) \text{ then } 0 \\ \text{else } 1 + \text{length}(\text{rest}(1)) \\ 1 := \text{cons}(\text{cons}(1, \text{null}), \text{cons}(\text{cons}(2, \text{cons}(3, \text{null})), \text{null})) \\ \text{in } \text{length}(1) + \text{length}(\text{first}(1)) : \text{int}} [\text{letpoly}]$$

Task 3: The second program in the preceding section is an example. The following is a shorter (but not necessarily simpler) example:

```
let id := map x to x;  
in (id(id))(0)
```

The program is not typable in Typed Jam because the function `id` is applied to an argument of type `int → int` and again (since `id(id)` is `id`) to the an argument of type `int`. Hence it must have type `(int → int) → (int → int)` and type `(int → int)` which cannot be unified.