C411 – Type Inference Study Guide

Corky Cartwright

Produced: March 18, 2017

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*.

$M ::= M\ \texttt{(}M \cdots M\texttt{)} \mid P\ \texttt{(}M \cdots M\texttt{)} \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \mid \texttt{let}\ x\ \texttt{:=}\ M\ \texttt{in}\ M$
$\quad \mid \mathrm{V}$
$V ::= \texttt{map}\ x \cdots x\ \texttt{to}\ M \mid x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{null}$
$n ::= \texttt{1} \mid \texttt{2} \mid \ldots$
$P ::= \texttt{cons} \mid \texttt{first} \mid \texttt{rest} \mid \texttt{null?} \mid \texttt{cons?} \mid \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<} \mid$
$\texttt{<=} \mid \texttt{<-} \quad \mid \texttt{+} \mid \texttt{-} \mid \texttt{\~{}} \mid \texttt{ref} \mid \texttt{!}$
$x ::=$ 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.

$\sigma ::= \forall \alpha_1 \cdots \alpha_n . \tau$
$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \alpha \mid \mathsf{list}\,\tau \mid \mathsf{ref}\,\tau$
$\alpha ::=$ type variable names

1.2 Type Checking Rules

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

$$\frac{\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash M : \tau}{\Gamma \vdash \texttt{map}\ x_1 \ldots x_n\ \texttt{to}\ M : \tau_1 \times \cdots \times \tau_n \to \tau}[\textbf{abs}]$$

$$\frac{\Gamma \vdash M : \tau_1 \times \cdots \times \tau_n \to \tau \quad \Gamma \vdash M_1 : \tau_1 \quad \cdots \quad \Gamma \vdash M_n : \tau_n}{\Gamma \vdash M\ \ (M_1 \cdots M_n) : \tau}[\textbf{app}]$$

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

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.

$$\Gamma[x : \tau] \vdash x : \tau$$

$$\frac{\Gamma' \vdash M_1 : \tau'_1 \quad \ldots \quad \Gamma' \vdash M_n : \tau'_n \quad \Gamma' \vdash M : \tau}{\Gamma \vdash \texttt{let}\ x_1\ \texttt{:=}\ M_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{:=}\ M_n\texttt{;}\ \texttt{in}\ M : \tau}[\textbf{letmono}]$$

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

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

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

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

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

where V is a syntactic value, N is an expression that is not a syntactic value, and $\mathrm{FTV}(\alpha)$ 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

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

substitute *any* type terms $\tau_1, \ldots, \tau_n$ for the quantified type variables $\alpha_1, \ldots, \alpha_n$:

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

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

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

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 consisting of this table.

		`+`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`true`	bool	`-`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`false`	bool	`*`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
n	int	`/`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`null`	$\forall \alpha. \mathsf{list}\, \alpha$		
		`<`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`cons`	$\forall \alpha. \alpha \times \mathsf{list}\, \alpha \to \mathsf{list}\, \alpha$	`>`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`first`	$\forall \alpha. \mathsf{list}\, \alpha \to \alpha$	`<=`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`rest`	$\forall \alpha. \mathsf{list}\, \alpha \to \mathsf{list}\, \alpha$	`>=`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`cons?`	$\forall \alpha. \mathsf{list}\, \alpha \to \mathsf{bool}$		
`null?`	$\forall \alpha. \mathsf{list}\, \alpha \to \mathsf{bool}$	(unary) `-`	$\mathsf{int} \to \mathsf{int}$
		(unary) `+`	$\mathsf{int} \to \mathsf{int}$
`=`	$\forall \alpha. \alpha \times \alpha \to \mathsf{bool}$	(unary) `~`	$\mathsf{bool} \to \mathsf{bool}$
`!=`	$\forall \alpha. \alpha \times \alpha \to \mathsf{bool}$	`<-`	$\forall \alpha. \mathsf{ref}\, \alpha \times \alpha \to \mathsf{unit}$
		`ref`	$\forall \alpha. \alpha \to \mathsf{ref}\, \alpha$
		`!`	$\forall \alpha. \mathsf{ref}\, \alpha \to \alpha$

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. Γ_0 `|- (map x to x(10))(map x to x) :  int`

2. Γ_0 `|- let fact := map n to if n=0 then 1 else n*(fact(n-1));`
   `in fact(10)+fact(0) :  int`

3. Γ_0 `|- (map x to 1 + (1/x))(0) :  int`

4. Γ_0 `|- (map x to x) (map y to y) :  (int -> int)`

5. Γ_0 `|- let id := map x to x; in id(id) :  (int -> 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*:

$$\dfrac{\dfrac{\Gamma_0[\mathtt{f}:\mathsf{int}\to\mathsf{int}]\vdash \mathtt{10}:\mathsf{int} \qquad \Gamma_0[\mathtt{f}:\mathsf{int}\to\mathsf{int}]\vdash \mathtt{f}:\mathsf{int}\to\mathsf{int}}{\Gamma_0[\mathtt{f}:\mathsf{int}\to\mathsf{int}]\vdash \mathtt{f(10)}:\mathsf{int}}[\mathbf{app}]}{\Gamma_0\vdash \mathtt{map\ f\ to\ f(10)}:(\mathsf{int}\to\mathsf{int})\to\mathsf{int}}[\mathbf{abs}]$$

   Tree 2:

$$\dfrac{\mathit{Tree}\ 1 \qquad \dfrac{\Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{x}:\mathsf{int}}{\Gamma_0\vdash \mathtt{map\ x\ to\ x}:\mathsf{int}\to\mathsf{int}}[\mathbf{abs}]}{\Gamma_0\vdash \mathtt{(map\ f\ to\ f(10))(map\ x\ to\ x)}:\mathsf{int}}[\mathbf{app}]$$

2. Type Inference Proof Omitted.