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 \texttt{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 \texttt{let} is recursive.

\[
M ::= M (M \cdots M) | P (M \cdots M) | \text{if } M \text{ then } M \text{ else } M | \text{let } x := M \text{ in } M \\
V ::= \text{map } x \cdots x \text{ to } M | x | n | \text{true} | \text{false} | \text{null} \\
n ::= 1 | 2 | \ldots \\
P ::= \text{cons} | \text{first} | \text{rest} | \text{null?} | \text{cons?} | + | - | / | * | = | < | <= | <-> | + | - | ~ | \text{ref} | ! \\
x ::= \text{variable names}
\]

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

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

\[
\sigma ::= \forall \alpha_1 \cdots \alpha_n. \tau \\
\tau ::= \text{int} | \text{bool} | \text{unit} | \tau_1 \times \cdots \times \tau_n \to \tau | \alpha | \text{list } \tau | \text{ref } \tau \\
\alpha ::= \text{type variable names}
\]
1.2 Type Checking Rules

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

\[
\begin{align*}
\frac{\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash M : \tau}{\Gamma \vdash \text{map } x_1 \ldots x_n \text{ to } M : \tau_1 \times \cdots \times \tau_n \rightarrow \tau} \quad \text{[abs]} \\
\frac{\Gamma \vdash M : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad \Gamma \vdash M_1 : \tau_1 \quad \cdots \quad \Gamma \vdash M_n : \tau_n}{\Gamma \vdash M (M_1 \cdots M_n) : \tau} \quad \text{[app]} \\
\frac{\Gamma \vdash M_1 : \text{bool} \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau} \quad \text{[if]}
\end{align*}
\]

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 $\Gamma$) 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.

\[
\begin{align*}
\Gamma[x : \tau] & \vdash x : \tau \\
\frac{\Gamma' \vdash M_1 : \tau'_1 \quad \cdots \quad \Gamma' \vdash M_n : \tau'_n \quad \Gamma' \vdash M : \tau}{\Gamma \vdash \text{let } x_1 := M_1; \ldots; x_n := M_n; \text{ in } M : \tau} \quad \text{[letmono]} \\
\frac{\Gamma' \vdash M_1 : \tau'_1 \quad \cdots \quad \Gamma' \vdash M_n : \tau'_n \quad \Gamma[x_1 : C_{M_1}(\tau'_1, \Gamma), \ldots, x_n : C_{M_n}(\tau'_n, \Gamma)] \vdash M : \tau}{\Gamma \vdash \text{let } x_1 := M_1; \ldots; x_n := M_n; \text{ in } M : \tau} \quad \text{[letpoly]} \\
\Gamma[x : \forall \alpha_1, \ldots, \alpha_n \vdash \tau] & \vdash x : O(\forall \alpha_1, \ldots, \alpha_n \vdash \tau, \tau_1, \ldots, \tau_n)
\end{align*}
\]

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 FTV($\alpha$) means the “free type variables in the expression (or type environment) $\alpha$”.

When closing over a type, you must find all of the free variables in $\tau$ that are not free in any of the types in the environment $\Gamma$. Then, build a polymorphic type by quantifying $\tau$ 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,
\[ O(\forall \alpha. \alpha \rightarrow \tau) = \tau \rightarrow \tau \]

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 consisting of this table.

<table>
<thead>
<tr>
<th>true</th>
<th>bool</th>
<th>+</th>
<th>int \times int \rightarrow int</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>bool</td>
<td>-</td>
<td>int \times int \rightarrow int</td>
</tr>
<tr>
<td>n</td>
<td>int</td>
<td>*</td>
<td>int \times int \rightarrow int</td>
</tr>
<tr>
<td>null</td>
<td>\forall \alpha. \text{list} \alpha</td>
<td>/</td>
<td>int \times int \rightarrow int</td>
</tr>
<tr>
<td>cons</td>
<td>\forall \alpha. \alpha \times \text{list} \alpha \rightarrow \text{list} \alpha</td>
<td>&lt;</td>
<td>int \times int \rightarrow \text{bool}</td>
</tr>
<tr>
<td>first</td>
<td>\forall \alpha. \text{list} \alpha \rightarrow \alpha</td>
<td>&gt;</td>
<td>int \times int \rightarrow \text{bool}</td>
</tr>
<tr>
<td>rest</td>
<td>\forall \alpha. \text{list} \alpha \rightarrow \text{list} \alpha</td>
<td>&lt;=</td>
<td>int \times int \rightarrow \text{bool}</td>
</tr>
<tr>
<td>cons?</td>
<td>\forall \alpha. \text{list} \alpha \rightarrow \text{bool}</td>
<td>&gt;=</td>
<td>int \times int \rightarrow \text{bool}</td>
</tr>
<tr>
<td>null?</td>
<td>\forall \alpha. \text{list} \alpha \rightarrow \text{bool}</td>
<td>(unary) \rightarrow</td>
<td>int \rightarrow int</td>
</tr>
<tr>
<td>=</td>
<td>\forall \alpha. \alpha \times \alpha \rightarrow \text{bool}</td>
<td>(unary) +</td>
<td>int \rightarrow int</td>
</tr>
<tr>
<td>!=</td>
<td>\forall \alpha. \alpha \times \alpha \rightarrow \text{bool}</td>
<td>(unary) \rightarrow</td>
<td>\text{bool} \rightarrow \text{bool}</td>
</tr>
<tr>
<td>ref</td>
<td>\forall \alpha. \alpha \rightarrow \text{ref} \alpha</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>\forall \alpha. \text{ref} \alpha \rightarrow \alpha</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. $\Gamma_0 \vdash (\text{map } x \to x(10))(\text{map } x \to x) : \text{int}$

2. $\Gamma_0 \vdash \text{let } \text{fact} := \text{map } n \to \text{if } n=0 \text{ then } 1 \text{ else } n*(\text{fact}(n-1));$
   $\text{in } \text{fact}(10)+\text{fact}(0) : \text{int}$

3. $\Gamma_0 \vdash (\text{map } x \to 1 + (1/x))(0) : \text{int}$

4. $\Gamma_0 \vdash (\text{map } x \to x) (\text{map } y \to y) : (\text{int } \to \text{int})$

5. $\Gamma_0 \vdash \text{let } \text{id} := \text{map } x \to x; \text{ in } \text{id}(\text{id}) : (\text{int } \to \text{int})$

**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 := \text{map } f,l \to
   \text{if null}(l) \text{ then } \text{null}
   \text{ else cons}(f(\text{first}(l)), \text{listMap}(f, \text{rest}(l)))
   \text{ in listMap}(\text{first}, \text{null});

2. let length := \text{map } l \to \text{if null}(l) \text{ then } 0
   \text{ else } 1 + \text{length}(\text{rest}(l));
   \text{ l := cons(cons(1,\text{null}),cons(cons(2,cons(3,\text{null})),\text{null}));
   \text{ 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{
   \Gamma_0[f:\text{int } \to \text{int}] \vdash 10:\text{int}
   \quad \Gamma_0[f:\text{int } \to \text{int}] \vdash f:\text{int } \to \text{int}}{
   \Gamma_0[f:\text{int } \to \text{int}] \vdash f(10):\text{int}} \quad \text{[app]}

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

2. **Tree 2:**

   $\dfrac{
   \Gamma_0[x:\text{int}] \vdash x:\text{int}}{
   \Gamma_0 \vdash \text{map } x \to x:\text{int } \to \text{int}} \quad \text{[abs]}

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

2. Type Inference Proof Omitted.
3. **Tree 1:**

$$
\Gamma_0[x:int] \vdash /:int \times int \to int \quad \Gamma_0[x:int] \vdash 1:int \quad \Gamma_0[x:int] \vdash x:int
$$

$$
\Gamma_0[x:int] \vdash 1/x:int
$$

**Tree 2:**

$$
\Gamma_0[x:int] \vdash +:int \times int \to int \quad \Gamma_0[x:int] \vdash 1:int
$$

$$
\Gamma_0 \vdash (map x to 1 + (1/x)):int \to int
$$

**Tree 3:**

$$
\Gamma_0 \vdash 0:int
$$

$$
\Gamma_0 \vdash (map x to 1 + (1/x))(0):int
$$

4. **Tree 1:**

$$
\Gamma_0[x:int] \vdash x:int \to int
$$

$$
\Gamma_0 \vdash (map x to x):(int \to int) \to (int \to int)
$$

**Tree 2:**

$$
\Gamma_0[y:int] \vdash y:int
$$

$$
\Gamma_0 \vdash (map y to y):int \to int
$$

**Tree 3:**

$$
\Gamma_0 \vdash (map x to x)(map y to y):int \to int
$$

5. This example is almost identical to the previous one, but the identity function \(\text{id}\) 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 \(\text{id}\). 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 \(\text{length}\) function is only applied to one type of list. Hence the \(\text{letmono}\) rule can be used to type the \(\text{let}\) expression in this program instead of the more general \(\text{letpoly}\) rule.

1. Type Inference Proof Omitted.

2. Let \(\Gamma_1\) abbreviate \(\Gamma_0[\text{length}:\text{list}\alpha \to \text{int}, 1:\text{list}\text{list}\text{int}]\);

let \(\Gamma_2\) abbreviate \(\Gamma_0[\text{length} : \forall \alpha. (\text{list}\alpha \to \text{int}), 1 : \text{list}\text{list}\text{int}]\);

and let \(\Gamma_3\) abbreviate \(\Gamma_1[1: \text{list}\alpha]\).

**Tree 1:**

$$
\Gamma_3 \vdash \text{rest}:\text{list}\alpha \to \text{list}\alpha \quad \Gamma_3[1: \text{list}\alpha] \quad \Gamma_3 \vdash \text{length}:\text{list}\alpha \to \text{int}
$$

$$
\Gamma_3 \vdash \text{rest}(1):\text{list}\alpha
$$

$$
\Gamma_3 \vdash \text{length} \text{rest}(1):\text{int}
$$
Tree 2:
\[ \Gamma_3 \vdash + : \text{int} \times \text{int} \rightarrow \text{int} \]
\[ \Gamma_3 \vdash 1 : \text{int} \]
\[ \Gamma_3 \vdash 1 + \text{length(rest(l))} : \text{int} \]

Tree 3:
\[ \Gamma_3 \vdash \text{null?} : \text{list} \alpha \rightarrow \text{bool} \]
\[ \Gamma_1 \vdash 1 : \text{int} \]
\[ \Gamma_3 \vdash 0 : \text{int} \]
\[ \Gamma_3 \vdash \text{if null?}(l) \text{ then } 0 \text{ else } 1 + \text{length}(\text{rest}(l)) : \text{int} \]
\[ \Gamma_1 \vdash \text{map l to if null?}(l) \text{ then } 0 \text{ else } 1 + \text{length}(\text{rest}(l)) : \text{int} \]

Tree 4:
\[ \Gamma_1 \vdash \text{cons} : \text{int} \times \text{list int} \rightarrow \text{list int} \]
\[ \Gamma_1 \vdash 1 : \text{int} \]
\[ \Gamma_1 \vdash \text{null} : \text{list int} \]
\[ \Gamma_1 \vdash \text{cons}(1, \text{null}) : \text{list int} \]

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

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

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

Tree 8:
\[ \Gamma_1 \vdash \text{length} : \text{list int} \rightarrow \text{int} \]
\[ \Gamma_2 \vdash \text{first} : \text{list list int} \rightarrow \text{list int} \]
\[ \Gamma_1 \vdash 1 : \text{list int} \]
\[ \Gamma_2 \vdash \text{first}(l) : \text{list int} \]
\[ \Gamma_2 \vdash \text{length}(\text{first}(l)) : \text{int} \]

Tree 9:
\[ \Gamma_1 \vdash \text{length} : \text{list int} \rightarrow \text{int} \]
\[ \Gamma_2 \vdash \text{length} : \text{list int} \rightarrow \text{int} \]
\[ \Gamma_2 \vdash 1 : \text{list int} \]
\[ \Gamma_2 \vdash \text{length}(l) : \text{int} \]
\[ \Gamma_2 \vdash \text{length}(l) + \text{length}(\text{first}(l)) : \text{int} \]

Tree 10:
\[ \Gamma_2 \vdash + : \text{int} \times \text{int} \rightarrow \text{int} \]
\[ \Gamma_2 \vdash \text{length} : \text{list int} \rightarrow \text{int} \]
\[ \Gamma_2 \vdash 1 : \text{list int} \]
\[ \Gamma_2 \vdash \text{length}(l) : \text{int} \]
\[ \Gamma_2 \vdash \text{length}(l) + \text{length}(\text{first}(l)) : \text{int} \]

Tree 11:
\[ \Gamma_0 \vdash \text{let length} := \text{map l to if null?}(l) \text{ then } 0 \text{ else } 1 + \text{length}(\text{rest}(l)) \]
\[ 1 := \text{cons}(\text{cons}(1, \text{null}), \text{cons}(\text{cons}(2, \text{cons}(3, \text{null})), \text{null})) \]
\[ \text{in length}(l) + \text{length}(\text{first}(l)) : \text{int} \]

Tree 3

Tree 8

Tree 10

Tree 4

Tree 7

Tree 6

Tree 5

Tree 2

Tree 1
Task 3: The second program in the preceding section is an example. The following is a shorter (but not necessarily simpler) example:

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

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