1 Constraint Solving Example

In order to gain some intuition about constraint solving, let’s work through an example by hand before we formalize the constraint solving process. Consider the Rip code

\[(\text{proc } \{g\} \ (\text{proc } \{x\} \ (+1 \ \{g \ x\})))\]

We can use our rules for \([\bullet]\) from last time to generate type constraints for this program:

\[\begin{align*}
[[\{\text{proc} \ {g}\}]] &= [[g]] \rightarrow [[\{\text{proc} \ {x}\} \ (+1 \ \{g \ x\})]] \\
[[\{\text{proc} \ {x}\} \ (+1 \ \{g \ x\})]] &= [[x]] \rightarrow [[(+1 \ \{g \ x\})]] \\
[[+1 \ \{g \ x\}]] &= [[\{g \ x\}] = \text{num} \\
[[g]] &= [[x]] \rightarrow [[\{g \ x\}]]
\end{align*}\]

Solving constraints:

\[\begin{align*}
[[+1 \ \{g \ x\}]] &= \text{num} \\
[[\{\text{proc} \ {x}\} \ (+1 \ \{g \ x\})]] &= [[x]] \rightarrow \text{num} \\
[[g]] &= [[x]] \rightarrow \text{num} \\
[[\{\text{proc} \ {g}\} \ ...]] &= (([[x]] \rightarrow \text{num}) \rightarrow ([[x]] \rightarrow \text{num}))
\end{align*}\]

2 The Algorithm

We will use an algorithm called unification to solve constraints. Unification is what Prolog is based on. It is used all over the AI field. In short, it’s an all-around good thing to know.

1. Begin with an empty stack and an empty substitution (\(\text{subst}\)).
2. Place an equation $S = T$ on the stack.

3. Until $stack$ is empty:
   a. Pop the term $X = Y$ off the stack.
   b. If $X$ and $Y$ are identical constants, then do nothing.
   c. Else if $X$ and $Y$ are identical variables, then do nothing.
   d. Else if $X$ is a variable, then add substitution $X \mapsto Y$ to $subst$ and replace all occurrences of $X$ by $Y$ in both $stack$ and $subst$.
   e. Else if $Y$ is a variable, then add substitution $Y \mapsto X$ to $subst$ and replace all occurrences of $Y$ by $X$ in both $stack$ and $subst$.
   f. Else if $X = C(X_1, \ldots, X_n)$ and $Y = C(Y_1, \ldots, Y_n)$, then for $i = 1..n$, add each of $X_i \mapsto Y_i$ onto the stack.
   g. Else, $X$ and $Y$ do not unify so the original $S$ and $T$ do not unify. Hence, return a unification error.

Note that in rule $f$, the $C$'s are type constructors. For example, $\rightarrow$ is a 2-ary type constructor, $list$ is a 1-ary type constructor, and $num$ is a 0-ary type constructor.

### 3 Examples of the Algorithm

#### 3.1 Example

The Rip program

```plaintext
{{proc {x} x}
 7}
```

generates the following constraints. The superscripts below are a shorthand for introducing type variables for each expression:

- $[[\{proc\ {x} \ x}\]]^\alpha = ([[[7]]]^{\delta} \rightarrow [[[\{proc\ {x} \ x\} \ 7]]])$
- $[[\{proc\ {x} \ x\}]]^\alpha = ([[x]]^{\delta} \rightarrow [[x]])$
- $[[7]]^{\delta} = num$

When we run the algorithm, the stack and substitution are as follows:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Current Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta = \text{num})</td>
<td>[ ]</td>
</tr>
<tr>
<td>(\alpha = (\delta \rightarrow \delta))</td>
<td>(\beta \mapsto \text{num})</td>
</tr>
<tr>
<td>(\alpha = (\beta \rightarrow \gamma))</td>
<td>(\alpha \mapsto (\delta \rightarrow \delta), \beta \mapsto \text{num})</td>
</tr>
<tr>
<td>(\delta \rightarrow \delta = (\text{num} \rightarrow \gamma))</td>
<td>(\alpha \mapsto (\delta \rightarrow \delta), \beta \mapsto \text{num})</td>
</tr>
<tr>
<td>(\delta = \text{num})</td>
<td>(\delta \mapsto \text{num}, \alpha \mapsto (\text{num} \rightarrow \text{num}), \beta \mapsto \text{num})</td>
</tr>
<tr>
<td>(\text{num} = \gamma)</td>
<td>(\gamma \mapsto \text{num}, \delta \mapsto \text{num}, \alpha \mapsto (\text{num} \rightarrow \text{num}), \beta \mapsto \text{num})</td>
</tr>
<tr>
<td>({})</td>
<td>({})</td>
</tr>
</tbody>
</table>

### 3.2 Example

Consider the Rip code

```plaintext
(proc {x})
  {cons (+ 1 x) (car x))
```

This code generates the following constraints:

- \([[[\{\text{proc} \ldots\}]]^\alpha = [[[\{\text{cons} \{+ 1 x\} \{\text{car} x\}\}]]^\beta\rightarrow [[[\{\text{cons} \{+ 1 x\} \{\text{car} x\}\}]]^\gamma}}]
- \(\{[[\{\text{cons} \{+ 1 x\} \{\text{car} x\}\}]]^\gamma = \text{list}([[[[+ 1 x] ]]^\delta})]
- \(\{[[\{\text{car} x\}]]^\gamma = \text{list}([[[[+ 1 x] ]]^\delta])\)
- \(\{[[x] ]^\delta = \text{list}([[[[\{\text{car} x\}]]^\gamma])}
- \(\{[[+ 1 x] ]^\delta = \text{num}
- \(\{[[x] ]^\delta = \text{num}

Running the algorithm looks like this:
The next step of the algorithm fails because the 0-ary constructor `num` does not match the 1-ary constructor `list`.

### 3.3 Example

Assume we have the constraint

\[
\text{list(list(α), α) = list(list(bool), list(num))}
\]

The next step of the algorithm follows the \textit{else} clause, and thus fails.

### 3.4 Example

Assume we have the constraint

\[
\text{list(α) = list(list(α))}
\]
The substitution that unification produces here doesn’t even make sense as a type. We need to restrict our algorithm from producing answers like this. Hence we add an occurs check to parts d and e of the algorithm. That is, we only add the substitution $X \rightarrow Y$ if $X$ does not appear anywhere in $Y$. If it does, then we fail.