Typed Assembly Language

1. In class we saw an example program in TAL-0 for computing the product of integers stored in register $r_1$ and $r_2$. Suppose that the heap type $\Psi$ maps label $\text{loop}$ to type

$$\text{code}(\{r_1, r_2, r_3: \text{int}, r_4: \forall \alpha. \text{code}(r_1, r_2, r_3: \text{int}, r_4: \alpha)\}).$$

Define types for labels $\text{done}$ and $\text{prod}$ and show their typing derivations.

2. Suppose that we change the $\text{done}$ instruction sequence from $\text{jump } r_4$ to $\text{jump } r_1$. Argue that there is no way to prove the code is well-typed.

Abstract Interpretation

1. In class we gave a definition that $\alpha$ and $\gamma$ form a Galois connection if

   - $\alpha$ and $\gamma$ are monotonic
   - $S \leq \gamma(\alpha(S))$ for any concrete element $S$.
   - $\alpha(\gamma(A)) \leq A$ for any abstract element $A$.

   (Actually, in class the last inequality was an equality, which it usually is in practice; in that case, $\alpha$ and $\gamma$ would form a Galois insertion.)

   Similarly, we say that two (arbitrary) functions $\alpha$ and $\gamma$ form an adjunction if

   $$S \leq \gamma(A) \text{ iff } \alpha(S) \leq A \text{ for all concrete } S, \text{ abstract } A$$

   Show that $\alpha$ and $\gamma$ form a Galois connection if and only if they form an adjunction.

   (next page)
2. Consider the factorial function:

\[
\text{fun fact } n = \text{if } (n == 0) \text{ then } 1 \text{ else } n \ast (\text{fact } (n-1))
\]

We wish to use abstract interpretation to prove that, given a non-negative integer as input, this function always produces a positive integer as output.

(a) Suppose we use as our abstract domain the rule of signs, so that numbers are represented as one of \( A = \{+,-,0,\perp,\top\} \) (ordered as shown in the lecture notes). We want to compute a function \( \text{fact}' : A \to A \) in our abstract domain that soundly models \( \text{fact} \). Ultimately, we would like to show that \( \text{fact}'(+) = \text{fact}'(0) = + \). Since \( \text{fact} \) is a recursive function, we need to do this by computing a fixpoint. We will compute a series \( \text{fact}_0, \text{fact}_1, \text{fact}_2, \ldots \) of successively more accurate models of \( \text{fact} \) until we reach a fixpoint. Let \( \text{fact}_0(x) = \perp \) for all \( x \in A \). Written out in a tabular form, we have

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \perp )</th>
<th>( \top )</th>
<th>( + )</th>
<th>( - )</th>
<th>( 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fact}_0(x) )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
</tr>
</tbody>
</table>

This is our first (rather poor) approximation. To compute the next approximation, \( \text{fact}_1 \), for each possible input value of \( x \), we will use abstract interpretation to evaluate \( \text{fact} \). When we see the recursive call \( \text{fact}(n-1) \), we use our previous approximation \( \text{fact}_0 \). Formally, we compute a series with

\[
\text{fact}_{i+1}(x) = \text{AEval}(\text{if } (x == 0) \text{ then } 1 \text{ else } x \ast (\text{fact}_i (x - 1)))
\]

For example, using this formula to compute \( \text{fact}_1(x) \), we get

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \perp )</th>
<th>( \top )</th>
<th>( + )</th>
<th>( - )</th>
<th>( 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fact}_1(x) )</td>
<td>( \perp )</td>
<td>( + )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( + )</td>
</tr>
</tbody>
</table>

Remember that all operations, including the test at the \( \text{if} \), are strict in \( \perp \); e.g., \( x \ast \perp = \perp \). The definition of the * operation is given in the class notes; you need to construct semantics for the subtraction operation, and for test and if statement. For the latter, recall we are starting with \( \perp \) for this problem, and we will use join for approximations.

Continue computing successive approximations of \( \text{fact} \), writing each \( \text{fact}_i \) in tabular form, until you reach a fixpoint \( \text{fact}' \). Is it the case that \( \text{fact}'(+) = + \)? What went wrong?

(b) Construct a new abstract domain \( A' \) for which we can show that \( \text{fact} \) is positive given non-negative input. Draw a picture showing the lattice structure for \( A' \), and describe (briefly) operations in your new domain (if they are obvious, you can just say that you use the obvious abstract operations). Finally, compute a fixpoint approximation (i.e., give a sequence of tables, as above) for \( \text{fact} \) in your new abstract domain showing that \( \text{fact} \) is positive given non-negative input. \textbf{Note:} To get an answer to this problem, \( \text{AEval()} \) must treat \( \text{if} \) as precisely as possible, i.e., it must know that the test \( 0 == 0 \) is always true, and it must know that \( x \) is zero on the true branch of the test \( x == 0 \) and non-zero on the false branch.