Problem 1 Polymorphic Types

Below are some programs written in FLARE. Translate them into FLEX/SP, being as faithful as possible in maintaining the spirit of the original computation.

a. (flare ()
    (def positive? (abs (n) (prim > n 0)))
    (def compose (abs (f g) (abs (x) (f (g x)))))
    ((compose not? positive?) 3))

Solution:

(flexsp ()
    (def positive? (-> (int) bool)
        (abs ((n int)) (prim > n 0)))
    (def compose (forall (in mid out)
        (-> ((-> (mid) out) (-> (in) mid))
            (-> (in) out)))
        (pabs (in mid out)
            (abs ((f (-> (mid) out))
                (g (-> (in) mid)))
            (abs ((x in))
                (f (g x))))))))
    (((pcall compose int bool bool)
        not?
        positive?)
        3))

b. (flare ()
    (def sum-integers
        (abs (int-list)
            (if (null? int-list)
                0
                (if (prim int? (prim fst int-list))
                    (prim + (prim fst int-list)
                        (sum-integers (prim snd int-list)))
                    (sum-integers (prim snd int-list)))))
        (sum-integers (list 1 (sym a) 2 #t)))

Solution: Let \( T_\Sigma \) be an abbreviation for \( \text{(oneof (int int) (sym sym) (bool bool))} \). Then the translation is:
(flexsp ()
   (def sum-integers (-> ((listof TΣ)) int)
      (abs ((lst (listof TΣ)))
         (if (null? lst)
             0
             (tagcase ((pcall car TΣ) lst)
               (int i (prim + i (sum-integers ((pcall cdr TΣ) lst))))
               (else (sum-integers ((pcall cdr TΣ) lst)))))))

(sum-integers
   ((pcall cons TΣ) (one TΣ 1)
      ((pcall cons TΣ) (one TΣ (sym a))
         ((pcall cons TΣ) (one TΣ #t)
            ((pcall null TΣ)())))))

Notes:

- FLEX/SP does not support type predicates such as `int?`. Explicit dynamic type checks in FL code must be simulated by ones and `tagcase`s in the corresponding FLEX/SP code. This is not as powerful as the original FLARE code because our `tagcase`s can only handle types we know about in advance.

- Our list operations from FLARE have become explicitly polymorphic global functions in FLEX/SP, which we must `pcall` onto the list type before using. These polymorphic functions are as powerful as our FLARE list operations for homogenous lists. As mentioned above, we simulate heterogeneous lists using ones and `tagcase`s to do our dynamic type-checking.

- We have to explicitly cons up our list because we don’t have a way to write functions with a variable number of arguments in FLEX/SP.

c. (flare ()
   (def inc (abs (n) (prim + n 1)))
   (def twice (abs (f) (abs (x) (f (f x))))))
   (((twice twice) inc) 3))

Solution:

(flexsp ()
   (def inc
      (-> (int) int)
      (abs ((n int)) (prim + n 1)))
   (def twice
      (forall (t) (-> ((-> (t) t)) (-> (t) t)))
      (pabs (t)
         (abs ((f (-> (t) t)))
            ((x t))
            (f (f x))))))
   (((pcall twice (-> (int) int)) (pcall twice int)) inc) 3))
Since the polymorphic \texttt{twice} is used in two different ways, the two different occurrences of \texttt{twice} must be projected onto two different types.

In the past, some people have used \texttt{(((pcall twice int) ((pcall twice int) inc)) 3)} instead of the solution above. This corresponds to the FLARE expression \texttt{((twice (twice inc)) 3)}, and therefore does not constitute a solution “maintaining the spirit of the original.”

**Problem 2 Exception Types**

Alyssa P. Hacker decides to extend FLEX with the dynamic exception handling primitives \texttt{catch} and \texttt{throw}:

\[
E ::= (\text{throw } E_v) \mid (\text{catch } E_h E_b) \mid \ldots
\]

The informal operational semantics of these constructs is as follows:

- \texttt{(throw } E_v\text{)} signals a dynamic exception whose value \(V\) is the value of \(E_v\). When an exception is signalled via \texttt{throw}, normal evaluation is aborted, and the first dynamically enclosing \texttt{catch} is found to handle the exception. If there is no enclosing \texttt{catch}, the behavior of \texttt{throw} is undefined.

- \texttt{(catch } E_h E_b\text{)} first evaluates \(E_b\). If an exception with value \(V\) is signalled during the evaluation of \(E_b\), the value of the \texttt{catch} expression is the value of \((\text{call } E_h V)\). If no exception is signalled during the evaluation of \(E_b\), the value of the \texttt{catch} expression is the value of \(E_b\). Note that the handler expression \(E_h\) is never evaluated if no exception is raised in the body \(E_b\).

For example:

\[
\begin{align*}
\text{let } ((\text{foo } (\text{abs } ((x \text{ int}))) \\
\quad (\text{if } (= x 0) \\
\quad \quad (\text{throw } 300) \\
\quad \quad 1000)))))) \\
\quad (\text{catch } (\text{abs } ((x \text{ int})) (+ x 1)) \\
\quad \quad (+ 20 (\text{foo } 7)))) \\
\Rightarrow 1020
\end{align*}
\]

\[
\begin{align*}
\text{let } ((\text{foo } (\text{abs } ((x \text{ int}))) \\
\quad (\text{if } (= x 0) \\
\quad \quad (\text{throw } 300) \\
\quad \quad 1000)))))) \\
\quad (\text{catch } (\text{abs } ((x \text{ int})) (+ x 1)) \\
\quad \quad (+ 20 (\text{foo } 0)))) \\
\Rightarrow 301
\end{align*}
\]

\[
\begin{align*}
\text{(catch } (\text{abs } ((x \text{ int})) (+ x 1)) \\
\quad (\text{throw } (\text{throw } 7))) \\
\Rightarrow 8
\end{align*}
\]

Alyssa also modifies the type system of FLEX. She assigns each expression two types, a normal type (denoted with : ) and an exception type (denoted with $ ). When she writes

\[
E : T_n$T_e
\]

3
she means that expression $E$ has normal type $T_n$ and exceptional type $T_e$. The exception type is the type of the argument to the `throw` that raised the exception. An expression that cannot produce an exceptional (normal) value will have `void` as its exceptional (normal) type. Here are some examples:

$$\begin{align*}
(\text{throw } 1): & \text{ void } \$ \text{ int} \\
1: & \text{ int } \$ \text{ void} \\
(\text{if } \#\text{t (throw (sym true)) } 1): & \text{ int } \$ \text{ sym} \\
(\text{throw } (\text{throw } 1)): & \text{ void } \$ \text{ int} \\
(\text{catch } (\text{abs } ((x \text{ int})) x) (\text{throw } 1)): & \text{ int } \$ \text{ void}
\end{align*}$$

Notice that an exception type is not needed for the arguments to a procedure. Also notice that the phrase $T_n$ $\$ T_e$ is not a type.

The FLEX typing operator $:$ is a relation on Expression $\times$ Type, while Alyssa’s new typing operator $:\$ is a relation on Expression $\times$ Type $\times$ Type. Type environments are the same for both relations: they map identifiers to types. In particular, in the case of $:\$, type environments do not map identifiers to pairs of types.

Alyssa also wants to add a limited form of subtyping to the language. (Recall that Alyssa is starting with FLEX and not FLEX/SP, so that there is no subtyping in the original language). She decides that `void` is a subtype of all types, and this is the only subtyping relationship that she allows in the language. She defines $\text{LUB}$, a least-upper-bound operator that takes a sequence of types and has the following functionality: if all of the types in the sequence are `void`, $\text{LUB}$ returns `void`; if all the non-`void` types are the same type, $\text{LUB}$ returns this type; otherwise, $\text{LUB}$ is undefined.

For example:

$$\begin{align*}
\text{LUB} [\text{void}, \text{void}] & = \text{void} \\
\text{LUB} [\text{int}, \text{int}] & = \text{int} \\
\text{LUB} [\text{void}, \text{int}] & = \text{int} \\
\text{LUB} [\text{bool}, \text{void}, \text{bool}] & = \text{bool} \\
\text{LUB} [\text{void}, \text{bool}, \text{int}] & = \text{undefined}
\end{align*}$$

Here is how $\text{LUB}$ is used in Alyssa’s rule for `if`:

$$\begin{align*}
A & \vdash E_1 : T_{n_1} \$ T_{e_1} \\
A & \vdash E_2 : T_{n_2} \$ T_{e_2} \\
A & \vdash E_3 : T_{n_3} \$ T_{e_3} \\
T_{n_1} & \sqsubseteq \text{bool} \quad \text{[if]} \\
T_n & = \text{LUB} [T_{n_2}, T_{n_3}] \\
T_e & = \text{LUB} [T_{e_1}, T_{e_2}, T_{e_3}] \\
A & \vdash (\text{if } E_1 E_2 E_3) : T_n \$ T_e
\end{align*}$$

Unfortunately, Alyssa was called away to help with 6.001 before she could complete her type checking rules and you are asked to help out.

a. Modify the grammar of FLEX types from Figure 11.28 on page 639 of the course notes to accommodate Alyssa’s new features.

\textbf{Solution:}

$$\begin{align*}
T & ::= \text{void} \quad \text{[new]} \\
& \mid (\rightarrow (T^*) T_n T_e) \quad \text{[modified]} \\
& \mid \ldots \quad \text{[as before]}
\end{align*}$$

As you can see, only two changes need to be made:

4
i. The base type \texttt{void} must be added.

ii. Arrow types must include a \textit{latent exception type} \( T_e \) in addition to the normal return type \( T_n \). This exception type indicates the type of the exception that can be signalled during the evaluation of the procedure body when the procedure is called. This information must be included in the procedure type so that the exception types of procedure calls can be properly determined. Similar latent information must be included in procedures in other kinds of static analysis, for example, analysis of side effects.

Note that it is \textit{not} necessary to include exception types for the formal parameters of a procedure in the type of the procedure. Since FLEX is call-by-value, a procedure in an application will never be called if one of the arguments signals an exception; thus, formal parameters should only have a normal type associated with them. If FLEX were call-by-name, then it would be necessary to include exception types for formal parameters.

The grammar for expressions remains unchanged with the possible exception of \texttt{catch}, which might be extended to declare an exception type in addition to a normal type.

b. Give the typing rules for each of the following forms:

i. \texttt{throw}

\textbf{Solution:}

\[
\frac{A \vdash E : T_n \land T_e}{T = \text{LUB} \ [T_n, T_e]} \quad [\text{throw}]
\]

\[
A \vdash \text{(throw} \ E) : \text{void} \land T
\]

The typing rule for \texttt{(throw} \ E)\texttt{)} must take the \textit{LUB} of the normal and exception types for \( E \) so that types in the following expressions are well-typed (suppose \( b \) has type \texttt{bool} below):

\[
\begin{align*}
(\text{throw} \ 1) & : \text{void} \land \text{int} \\
(\text{throw} \ (\text{throw} \ 20)) & : \text{void} \land \text{int} \\
(\text{throw} \ (\text{if} \ b \ 1 \ (\text{throw} \ 20))) & : \text{void} \land \text{int}
\end{align*}
\]

Note that the following expression is not well-typed because it can signal either \texttt{int} or \texttt{bool}, which are incomparable (again suppose \( b \) has type \texttt{bool}):

\[
(\text{throw} \ (\text{if} \ b \ 1 \ (\text{throw} \ #t)))
\]

ii. \texttt{catch}

\textbf{Solution:} This is the hardest case. Our solution uses two rules:

\[
\frac{A \vdash E_1 : (\rightarrow \ T_n, T_r) \land T_{e_1}}{A \vdash E_2 : T_{n_2} \land T_{e_2}}
\]

\[
T_{e_2} \subseteq T_n \quad [\text{catch-handled}]
\]

\[
T_n = \text{LUB} \ [T_{n_2}, T_r]
\]

\[
T_e = \text{LUB} \ [T_{e_1}, T_l]
\]

\[
A \vdash \text{(catch} \ E_1 \ E_2) : T_n \land T_e
\]

\[
\begin{align*}
A \vdash E_1 : & \text{void} \land T_{e_1} \\
A \vdash E_2 : & T_{n_2} \land T_{e_2}
\end{align*} \quad [\text{catch-void}]
\]

\[
A \vdash \text{(catch} \ E_1 \ E_2) : T_{n_2} \land T_{e_1}
\]
If $E_1$ has type $\rightarrow (T_a \ T_r \ T_t)$, the catch-handled rule is used.

- A catch expression returns normally whenever either $E_2$ signals no exception or $E_2$ signals an exception that is handled (without error) by $E_1$. Thus, the normal type of the catch is $LUB [T_{e_2}, T_r]$.
- A catch expression itself signals an exception whenever either the evaluation of $E_1$ signals an exception or the application of the exception-handling procedure signals an exception. Thus, the exception type is $LUB [T_{e_1}, T_l]$. Note that $T_{e_2}$ plays no part in the exception type of the catch.
- The exception type of $E_2$ must be a subtype of the type of the formal parameter of the value of $E_1$. For example:

$$(\text{catch (abs ((x bool)) x) 3})$$

Here, the exception type of 3 is void, but the type of the argument to the exception handling procedure is bool. If we required $T_{e_2} = T_a$, this would not be well-typed.

The type of $E_1$ could also be void. For example:

$$(\text{catch (throw 4) (throw #t)})$$

Here $E_1$ has type void, and exception type int. By the catch-void rule above, we can give this expression normal type void and exception type int.

### iii. abs

**Solution:**

$A[I_1 : T_1, \ldots, I_n : T_n] \vdash E_b : T_{e_b}$

$$A \vdash (\text{abs } ((I_1 \ T_1) \ldots (I_n \ T_n)) \ E_b) : \rightarrow (T_1 \ldots T_n) \ T_{e_b} \ T_{e_a}$$

This rule is crucial to the handling of latent exception types. The evaluation of a abs expression can never signal an exception; therefore its exception type must be void. However, later calling the procedure created by the abs expression may signal an exception during the evaluation of the body $E_b$. The latent exception type in the modified arrow type is necessary to “remember” that evaluating the body can signal an exception of type $T_{e_b}$.

### iv. application

**Solution:** As with catch, there are two rules.

$A \vdash E_p : \rightarrow (T_1 \ldots T_n) \ T_r \ T_t \ T_{e_p}$

$$\forall i \in \{1, \ldots, n\}. A \vdash E_i : T_i' \ T_{e_i}$$

$$T_e = LUB [T_{e_p}, T_{e_1}, \ldots, T_{e_n}]$$

$$A \vdash (E_p \ E_1 \ldots E_n) : T_r \ T_{e_p}$$

This rule handles the case where the operator has procedure type.
The normal types of the actual arguments are subtypes of the formals. This allows combinations where arguments always signal an error (whose normal type is void). For example,

\[
((\text{abs } ((x \text{ bool}) \ x)) \ (\text{throw } 3))
\]

would not be well-typed without the subtyping rule.

An exception can be signalled by evaluating \(E_p\), evaluating any one of the \(E_i\), or by applying the procedural value of \(E_p\) (possibly signalling the latent exception type). Thus, all these types must be combined via \(LUB\).

The second rule handles cases like this:

\[
(((\text{throw } 4) \ (\text{throw } #t))
\]

c. Give a modified subtyping rule for procedure types.

\textbf{Solution:} As with the result type, the procedure type is monotonic in the latent exception type:

\[
\forall i \in \{1, \ldots, n\}. T_i \sqsubseteq S_i \\
S_b \sqsubseteq T_b \\
S_l \sqsubseteq T_l \\
\Rightarrow (\rightarrow (S_1 \ldots S_n) \ S_b \ S_l) \sqsubseteq (\rightarrow (T_1 \ldots T_n) \ T_b \ T_l)
\]