CS 510 Programming Languages
Assignment 4

Out: Wednesday October 16, 2002
Due: 4:00 Friday October 25, 2002

1 Introduction

In this assignment, you will explore the implications of combining static and dynamic typing in the same language. The written part will consist of designing a translation from a dynamically typed variant of MinML to a statically typed one, and the programming part will consit of implementing that translation.

You may hand this assignment in in class on Wednesday Oct. 23th. Alternatively, you may hand the assignment in to Tina (CS Bldg 410) before 4:00 on Friday Oct. 25 (or to me in my office if you can find me there).

Availability of Code

You can find a directory full of code off of the course web site.

2 Extended Dynamic MinML

The language Dynamic MinML, in which safety is guaranteed by run-time tag-checking rather than static type-checking, was introduced in lecture. It was argued in the lecture that Dynamic MinML could be translated fairly simply into a variant of Static MinML with a special type of tagged values: intuitively, every expression in Dynamic MinML could be translated into a MinML expression of type `Tagged`. You may find it helpful to take a moment to think about how this would be done before beginning this assignment.

One problem with the most obvious such translation is that it performs a lot of "unnecessary" tagging and tag-checking. For example, if we attempt to translate the (untyped) expression

$$\texttt{+}(\texttt{*}(\texttt{number}(2),\texttt{number}(3)),\texttt{number}(4))$$

into statically typed MinML using a translation that turned *every* subexpression into an expression of type `Tagged`, we would produce something like the following:

$$\begin{array}{l}\texttt{tag\_int}(\texttt{+}(\texttt{as\_int}(\texttt{tag\_int}(\texttt{*}(\texttt{as\_int}(\texttt{tag\_int}(\texttt{number}(2))),\texttt{as\_int}(\texttt{tag\_int}(\texttt{number}(3)))))),\\ \qquad\texttt{as\_int}(\texttt{tag\_int}(\texttt{number}(4)))))\end{array}$$

This is clearly ridiculous; in fact, any realistic implementation of Dynamic MinML would probably notice that the literals $\texttt{number}(2)$, $\texttt{number}(3)$ and $\texttt{number}(4)$ must be integers, and that the result of the multiplication operation must be an integer, and so the following more efficient code would produce the same behavior:

$$\texttt{tag\_int}(\texttt{+}(\texttt{*}(\texttt{number}(2),\texttt{number}(3)),\texttt{number}(4)))$$

Unfortunately, it is not always so easy for a compiler to notice such opportunities for optimization of tags. For example, in the untyped factorial function

$$\texttt{fun}\, f(x)\, \texttt{is}\, \texttt{if}\, \texttt{=}(x,0)\, \texttt{then}\, 1\, \texttt{else}\, \texttt{*}(x,\texttt{apply}(f,\texttt{-}(x,1)))\, \texttt{fi}\, \texttt{end}$$

a compiler cannot avoid tagging the argument and return value unless it can "guess" that the intended type of the function is `Int` $\to$ `Int`. Guessing the type of a recursive function is difficult (though, as we will see later in the course, not always impossible), and so it is unlikely that this function could be translated without tagging and tag-checking.

One idea to alleviate this problem is to allow the programmer to insert typing annotations in programs — effectively giving "hints" to the compiler as to what the types of values are expected to be. In this assignment, you will explore this approach. We will define a language we will call Extended Dynamic MinML, which will be essentially the same as the Dynamic MinML given in lecture except that it will contain two new constructs: first, Extended Dynamic MinML will contain typed `fun`-expressions as well as untyped ones, so that programmers can make clear what types they intend functions to have; and second, it will contain a `check` expression that forces a value to have a certain type. Your job will be to define and implement a translation from Extended Dynamic MinML to the variant of MinML with a type of tagged values that attempts to take advantage of typing annotations to remove some tagging and tag-checking operations.

2.1 Syntax

The abstract syntax of Extended Dynamic MinML is given below.

$$
\begin{array}{llll}
\textit{Expressions} & e & ::= & \texttt{var}(x) \mid \texttt{number}(n) \mid \texttt{true} \mid \texttt{false} \\
 & & \mid & o(e_1,\ldots,e_n) \mid \texttt{fun}\, f(x)\, \texttt{is}\, e\, \texttt{end} \mid \texttt{apply}(e_1,e_2) \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi} \\
 & & \mid & \texttt{fun}\, f(x{:}\tau_1) : \tau_2\, \texttt{is}\, e\, \texttt{end} \mid \texttt{check}(e,\tau) \\
\textit{Types} & \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau_1 \to \tau_2 \\
\textit{Operators} & o & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<} \\
\textit{Programs} & P & ::= & e
\end{array}
$$

There are two binding constructs in this language: typed and untyped `fun`-expressions. In both cases, the two variables (the function name and the parameter) are bound in the body expression.

2.2 Static Semantics

The static semantics for Extended Dynamic MinML are a little different from those of ordinary MinML. Rather than define a single judgment of the form "$\Gamma \vdash e : \tau$", meaning "e has type τ in context Γ," we will define *two* judgments simultaneously:

1. $\Delta \vdash_{\mathsf{d}} e : \tau$, meaning that e has type τ in context Δ, and
2. $\Delta \vdash_{\mathsf{d}} e\ \mathsf{wf}$, meaning that in context Δ, e is a *well-formed* program that *may or may not have a type.*

(The subscript d serves to distinguish the static semantic judgments of Extended Dynamic MinML from those of ordinary MinML.) The contexts (written Δ) used in these rules are themselves a bit different from their statically-typed MinML counterparts. Rather than simply assigning types to variables as ordinary MinML contexts do, Extended Dynamic MinML contexts assign types to some variables but also contain bindings that do not give any type information. These contexts are given by the following grammar:

$$\Delta ::= \epsilon \mid \Delta, x\ \mathsf{val} \mid \Delta, x{:}\tau$$

The context item x val should be read "x is valid", and it means that the variable x has been bound somewhere but its type is not known.

The above judgments are defined inductively by the following inference rules.

$$\frac{(x\ \mathsf{val}) \in \Delta}{\Delta \vdash_{\mathsf{d}} \mathtt{var}(x)\ \mathsf{wf}} \qquad \frac{(x{:}\tau) \in \Delta}{\Delta \vdash_{\mathsf{d}} \mathtt{var}(x) : \tau}$$

$$\frac{}{\Delta \vdash_{\mathsf{d}} \mathtt{number}(n) : \mathtt{Int}} \qquad \frac{}{\Delta \vdash_{\mathsf{d}} \mathtt{true} : \mathtt{Bool}} \qquad \frac{}{\Delta \vdash_{\mathsf{d}} \mathtt{false} : \mathtt{Bool}}$$

$$\frac{o \in \{\mathtt{+},\mathtt{-},\mathtt{*}\} \quad \Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_2\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} o(e_1, e_2) : \mathtt{Int}} \qquad \frac{o \in \{\mathtt{=},\mathtt{<}\} \quad \Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_2\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} o(e_1, e_2) : \mathtt{Bool}}$$

$$\frac{\Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_2\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_3\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi}\ \mathsf{wf}} \qquad \frac{\Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_2 : \tau \quad \Delta \vdash_{\mathsf{d}} e_3 : \tau}{\Delta \vdash_{\mathsf{d}} \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi} : \tau}$$

$$\frac{\Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad v \vdash_{\mathsf{d}} e_2\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \mathtt{apply}(e_1, e_2)\ \mathsf{wf}} \qquad \frac{\Delta \vdash_{\mathsf{d}} e_1 : \tau_1 \to \tau_2 \quad \Delta \vdash_{\mathsf{d}} e_2\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \mathtt{apply}(e_1, e_2) : \tau_2}$$

$$\frac{\Delta, f\ \mathsf{val}, x\ \mathsf{val} \vdash_{\mathsf{d}} e\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \mathtt{fun}\, f(x)\, \mathtt{is}\, e\, \mathtt{end}\ \mathsf{wf}}\ (f, x \notin \Delta) \qquad \frac{\Delta, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash_{\mathsf{d}} e : \tau_2}{\Delta \vdash_{\mathsf{d}} \mathtt{fun}\, f(x{:}\tau_1) : \tau_2\, \mathtt{is}\, e\, \mathtt{end} : \tau_1 \to \tau_2}\ (f, x \notin \Delta)$$

$$\frac{\Delta \vdash_{\mathsf{d}} e\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \mathtt{check}(e, \tau) : \tau} \qquad \frac{\Delta \vdash_{\mathsf{d}} e : \tau}{\Delta \vdash_{\mathsf{d}} e\ \mathsf{wf}}$$

Notice that we have tried to make these rules as "permissive" as possible: any well-typed expression of any type is always well-formed (make sure you see why), and in most cases expressions are not strictly required to have any particular type. This means that we are giving as much "freedom" to the programmer as possible.

2.3 Dynamic Semantics

The dynamic semantics of Extended Dynamic MinML is shown here for your reference. You will not have to implement an interpreter for this language, but when you write your translation from Extended Dynamic MinML to MinML with tagged values, you will have to be sure that the behavior of programs is not changed by the translation. You may find it helpful to verify for yourself that Preservation and Progress are satisfied by this semantics.

Instructions

$$\frac{(n = n_1 + n_2)}{\mathtt{+}(\mathtt{number}(n_1), \mathtt{number}(n_2)) \mapsto \mathtt{number}(n)} \qquad \text{(and similarly for } \mathtt{-},\mathtt{*}\text{)}$$

$$\frac{(b = (n_1 = n_2))}{\mathtt{=}(\mathtt{number}(n_1), \mathtt{number}(n_2)) \mapsto \mathtt{Bool}(b)} \qquad \text{(and similarly for } \mathtt{<}\text{)}$$

$$\frac{}{\mathtt{if}\, \mathtt{true}\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi} \mapsto e_2} \qquad \frac{}{\mathtt{if}\, \mathtt{false}\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi} \mapsto e_3}$$

$$\frac{(v_1 = \mathtt{fun}\, f(x)\, \mathtt{is}\, e\, \mathtt{end}) \quad v_2\ \mathsf{value}}{\mathtt{apply}(v_1, v_2) \mapsto \{v_1, v_2/f, x\}e} \qquad \frac{v_1 = \mathtt{fun}\, f(x{:}\tau_1) : \tau_2\, \mathtt{is}\, e\, \mathtt{end} \quad v_2\ \mathsf{value}}{\mathtt{apply}(v_1, v_2) \mapsto \mathtt{apply}(\mathtt{fun}\, f(x)\, \mathtt{is}\, e\, \mathtt{end}, \mathtt{check}(v_2, \tau_1))}$$

$$\frac{}{\mathtt{check}(\mathtt{number}(n), \mathtt{Int}) \mapsto \mathtt{number}(n)} \qquad \frac{b \in \{\mathtt{true}, \mathtt{false}\}}{\mathtt{check}(b, \mathtt{Bool}) \mapsto b}$$

$$\frac{v \ \mathsf{value}}{\mathtt{check}(v, \tau_1 \to \tau_2) \mapsto \mathtt{fun}\, f(x{:}\tau_1) : \tau_2\ \mathtt{is}\ \mathtt{check}(\mathtt{apply}(v, \mathtt{var}(x)), \tau_2)\ \mathtt{end}}$$

Search Rules

$$\frac{e_1 \mapsto e_1'}{o(e_1, e_2) \mapsto o(e_1', e_2)} \qquad \frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{o(v_1, e_2) \mapsto o(v_1, e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{apply}(e_1, e_2) \mapsto \mathtt{apply}(e_1', e_2)} \qquad \frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathtt{apply}(v_1, e_2) \mapsto \mathtt{apply}(v_1, e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi} \mapsto \mathtt{if}\, e_1'\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi}} \qquad \frac{e \mapsto e'}{\mathtt{check}(e, \tau) \mapsto \mathtt{check}(e', \tau)}$$

Error Transitions

$$\frac{v_1 \ \mathsf{isnt\_int} \quad v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value}}{o(v_1, v_2) \mapsto \mathtt{error}} \qquad \frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value} \quad v_2 \ \mathsf{isnt\_int}}{o(v_1, v_2) \mapsto \mathtt{error}}$$

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value} \quad v_2 \ \mathsf{isnt\_fun}}{\mathtt{apply}(v_1, v_2) \mapsto \mathtt{error}} \qquad \frac{v_1 \ \mathsf{value} \quad v_1 \ \mathsf{isnt\_bool}}{\mathtt{if}\, v_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_2\, \mathtt{fi} \mapsto \mathtt{error}}$$

$$\frac{v \ \mathsf{value} \quad v \ \mathsf{isnt\_int}}{\mathtt{check}(v, \mathtt{Int}) \mapsto \mathtt{error}} \qquad \frac{v \ \mathsf{value} \quad v \ \mathsf{isnt\_bool}}{\mathtt{check}(v, \mathtt{Bool}) \mapsto \mathtt{error}}$$

Error Propagation

$$\frac{}{o(\mathtt{error}, e) \mapsto \mathtt{error}} \qquad \frac{v \ \mathsf{value}}{o(v, \mathtt{error}) \mapsto \mathtt{error}}$$

$$\frac{}{\mathtt{apply}(\mathtt{error}, e) \mapsto \mathtt{error}} \qquad \frac{v \ \mathsf{value}}{\mathtt{apply}(v, \mathtt{error}) \mapsto \mathtt{error}}$$

$$\frac{}{\mathtt{if}\, \mathtt{error}\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi} \mapsto \mathtt{error}} \qquad \frac{}{\mathtt{check}(\mathtt{error}, \tau) \mapsto \mathtt{error}}$$

3 MinML with Tagged Values

In this short section, we will review the syntax and semantics of the variant of MinML that includes a special type `Tagged` of tagged values. There is no work for you to do in this section; it is included for reference purposes only.

The syntax of this variant is a simple extension of ordinary MinML:

Expressions	e	$::=$	$\cdots \mid \mathtt{tag\_int}(e) \mid \mathtt{tag\_bool}(e) \mid \mathtt{tag\_fun}(e)$
		$\mid$	$\mathtt{as\_int}(e) \mid \mathtt{as\_bool}(e) \mid \mathtt{as\_fun}(e)$
Types	τ	$::=$	$\cdots \mid \mathtt{Tagged}$

The static semantics is extended with the following rules:

$$\frac{\Gamma \vdash e : \mathtt{Int}}{\Gamma \vdash \mathtt{tag\_int}(e) : \mathtt{Tagged}} \qquad \frac{\Gamma \vdash e : \mathtt{Bool}}{\Gamma \vdash \mathtt{tag\_bool}(e) : \mathtt{Tagged}} \qquad \frac{\Gamma \vdash e : \mathtt{Tagged} \to \mathtt{Tagged}}{\Gamma \vdash \mathtt{tag\_fun}(e) : \mathtt{Tagged}}$$

$$\frac{\Gamma \vdash e : \mathtt{Tagged}}{\Gamma \vdash \mathtt{as\_int}(e) : \mathtt{Int}} \qquad \frac{\Gamma \vdash e : \mathtt{Tagged}}{\Gamma \vdash \mathtt{as\_bool}(e) : \mathtt{Bool}} \qquad \frac{\Gamma \vdash e : \mathtt{Tagged}}{\Gamma \vdash \mathtt{as\_fun}(e) : \mathtt{Tagged} \to \mathtt{Tagged}}$$

The dynamic semantics is extended with the following rules (plus rules for propagation of `error`):

$$\frac{v\ \mathsf{value}}{\mathtt{as\_int}(\mathtt{tag\_int}(v)) \mapsto v} \qquad \frac{v\ \mathsf{value}}{\mathtt{as\_int}(\mathtt{tag\_bool}(v)) \mapsto \mathtt{error}} \qquad \frac{v\ \mathsf{value}}{\mathtt{as\_int}(\mathtt{tag\_fun}(v)) \mapsto \mathtt{error}}$$

$$\frac{e \mapsto e'}{\mathtt{as\_int}(e) \mapsto \mathtt{as\_int}(e')} \qquad \frac{e \mapsto e'}{\mathtt{tag\_int}(e) \mapsto \mathtt{tag\_int}(e')}$$

(and similarly for the other tag operations).

4 Translation

In this section, you will design a translation from Extended Dynamic MinML to the static version with a type of tagged values. The translation will be similar in some ways to the one for the simple Dynamic MinML discussed in lecture, but will take advantage of the type annotations, where possible, to avoid the inefficiency of constructing tagged values and of checking tags where this is unnecessary. To make this happen, you should design your translation so that expressions are translated into expressions *of the most precise type possible*. To do this, we will define the translation inductively as the judgment $\Delta \vdash e \Rightarrow e' : \tau$, meaning "The expression e, which is well-formed in the context Δ, translates to the expression e' of type τ," and maintain the following "invariants" throughout the translation:

1. If $\Delta \vdash_{\mathsf{d}} e_d : \tau$, then for some e_s, $\Delta \vdash e_d \Rightarrow e_s : \tau$.
2. If $\Delta \vdash_{\mathsf{d}} e_d\ \mathsf{wf}$ but there is no τ such that $\Delta \vdash_{\mathsf{d}} e_d : \tau$, then for some e_s, $\Delta \vdash e_d \Rightarrow e_s : \mathtt{Tagged}$.
3. If $\Delta \vdash e_d \Rightarrow e_s : \tau$ (where τ may be `Tagged`), and $|\Delta|$ is the MinML context obtained from Δ by replacing bindings of the form $x\ \mathsf{val}$ with ones of the form $x{:}\mathtt{Tagged}$, then $|\Delta| \vdash e_s : \tau$.

There are a lot of cases that the translation must cover, and some are more difficult than others. To help keep you from getting lost, the cases are separated into 4 separate categories: "well-typed" cases, pure tagged cases, boundary cases, and "odd" cases. You will probably find that the well-typed and pure tagged cases are the easiest, and that the boundary cases are the hardest.

Translation 1: "Well-Typed" Cases

To get you started with the translation, here are the rules for all the so-called "well-typed" cases. Note that because of the invariant we are maintaining, all of the expressions in the premises of these rules have translated to expressions of types other than `Tagged`, and their types "match up" in the usual way for well-typed expressions. As a result, these cases of the translation do not require any tagging or tag-checking at all

$$\frac{(x : \tau) \in \Delta}{\Delta \vdash x \Rightarrow x : \tau} \qquad \frac{}{\Delta \vdash \mathtt{number}(n) \Rightarrow \mathtt{number}(n) : \mathtt{Int}} \qquad \frac{}{\Delta \vdash \mathtt{true} \Rightarrow \mathtt{true} : \mathtt{Bool}} \qquad \text{(and similarly for } \mathtt{false}\text{)}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \mathtt{Int} \quad \Delta \vdash e_2 \Rightarrow e_2' : \mathtt{Int} \quad (op \in \{\mathtt{+}, \mathtt{-}, \mathtt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow op(e_1', e_2') : \mathtt{Int}}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Int} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Int} \quad (op \in \{\texttt{=},\texttt{<}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow op(e_1', e_2') : \texttt{Bool}}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Bool} \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau \quad \Delta \vdash e_3 \Rightarrow e_3' : \tau}{\Delta \vdash \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi} \Rightarrow \texttt{if}\, e_1' \,\texttt{then}\, e_2' \,\texttt{else}\, e_3' \,\texttt{fi} : \tau}$$

$$\frac{\Delta, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e \Rightarrow e' : \tau 2 \quad (f, x \notin \mathrm{dom}(\Delta))}{\Delta \vdash \texttt{fun}\, f(x{:}\tau_1) : \tau_2 \,\texttt{is}\, e \,\texttt{end} \Rightarrow \texttt{fun}\, f(x{:}\tau_1) : \tau_2 \,\texttt{is}\, e' \,\texttt{end} : \tau_1 \to \tau_2} \qquad \frac{\Delta \vdash e_1 \Rightarrow e_1' : \tau_1 \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau_2}{\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \texttt{apply}(e_1', e_2') : \tau_2}$$

Question: Translation 2: Pure Tagged Cases

For this question, fill in the blanks in the inference rules below. These rules cover the next set of cases in the translation: the ones in which some tag-checking is required. (We have also included the "untyped" variable rule in this group.)

$$\frac{(x \ \mathsf{val}) \in \Delta}{\Delta \vdash x \Rightarrow x : \texttt{Tagged}} \qquad \frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged} \quad (op \in \{\texttt{+},\texttt{-},\texttt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Int} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged} \quad (op \in \{\texttt{+},\texttt{-},\texttt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Int} \quad (op \in \{\texttt{+},\texttt{-},\texttt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}}$$

(Similar rules to the above apply for $op \in \{\texttt{=},\texttt{<}\}$.)

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau \quad \Delta \vdash e_3 \Rightarrow e_3' : \tau}{\Delta \vdash \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \,\texttt{fi} \Rightarrow \boxed{??} : \tau}$$

$$\frac{\Delta, f \ \mathsf{val}, x \ \mathsf{val} \vdash e \Rightarrow e' : \texttt{Tagged} \quad (f, x \notin \mathrm{dom}(\Delta))}{\Delta \vdash \texttt{fun}\, f(x) \,\texttt{is}\, e \,\texttt{end} \Rightarrow \boxed{??} : \texttt{Tagged}} \qquad \frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged}}{\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Tagged}}$$

Question: Translation 3: Boundary Cases

The trickiest part of the translation is dealing with what we may call the "boundary cases" — *i.e.*, those that arise when values must "cross the boundary" between typed and untyped parts of a program. You will not have to come up with the rules for these cases on your own — we will give you some big hints.

The first kind of boundary case is the kind that comes up when we "want" a tagged value, but "get" a value of some more specific type τ instead. For example, consider the case covered by the following partial rule:

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau \quad (\tau \neq \texttt{Tagged})}{\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \texttt{apply}(\texttt{as\_fun}(e_1'), \boxed{??}) : \texttt{Tagged}}$$

In this case, we are translating an application where the function translates as type `Tagged`. As you saw in the previous section, the obvious thing to do here is to try to extract a function from that tagged value and apply it to the argument — but in order for that to work we need an argument of type `Tagged`, whereas the translation of the argument e_2 has some type τ that is different from `Tagged`. Fortunately, `Tagged` is supposed to be the type of "untyped" values, and so we should be able to turn any value of any type into one of type `Tagged`. In fact, for any type τ we can (and you will, below) write a function:

$$\texttt{tagify}_\tau : \tau \to \texttt{Tagged}$$

Once we have defined such a function, we can replace the $\boxed{??}$ in the above rule so that the conclusion is the following:

$$\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \texttt{apply}(\texttt{as\_fun}(e_1'), \texttt{apply}(\texttt{tagify}_\tau, e_2')) : \texttt{Tagged}$$

The other kind of boundary case is the kind where a value of a particular type τ is "required", but the most precise type at which the expression may be translated is `Tagged`. For example, consider the following partial rule:

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \tau_1 \to \tau_2 \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged}}{\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \texttt{apply}(e_1', \boxed{??}) : \tau_2}$$

This rule applies when we are attempting to translate an application where the function expects a particular type τ_1, but we are unable to determine a more precise type for the argument expression than `Tagged`. What does this mean? It means that the argument *may turn out to have the wrong type*, and so the function application *may fail at run-time*. In order to perform the application in the statically typed MinML language, we must therefore *check* that the argument value is appropriate for the function (and, in the process, turn it into a value of type τ_1). In other words, for every type τ we need (and you will write, below) a function

$$\texttt{cast}_\tau : \texttt{Tagged} \to \tau$$

that takes a tagged value and converts it to one of type τ, taking a transition to `error` if this is impossible. Once we have such a function, we can replace the $\boxed{??}$ in the above rule so that the conclusion reads:

$$\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \texttt{apply}(e_1', \texttt{apply}(\texttt{cast}_{\tau_1}, e_2')) : \tau_2$$

For the first part of this question, you will show how to come up with functions $\texttt{tagify}_\tau$ and $\texttt{cast}_\tau$ that allow the boundary cases above to be translated. You will define these simultaneously, because they will be mutually referential. The first three cases of each will be very simple; the cases where τ is a function type will be the difficult ones. To do these, keep the following two observations in mind:

1. When you are *tagifying* a function, you are doing so because it is about to be used in a context where it may be applied to inappropriate values. You must therefore convert it to a function that *checks* that its argument is appropriate.

2. When you are *casting* a function, you are doing so because although it expects arguments of type `Tagged`, it is appearing in a context where it will be applied to values of a particular type τ_1 and expected to return values of a particular type τ_2. You must therefore convert it to a function that meets this specification.

Now, define $\texttt{tagify}_\tau$ and $\texttt{cast}_\tau$ by induction on the structure of the type τ.

Finally, fill in the blank in these two rules for translating `check` expressions:

$$\frac{\Delta \vdash e \Rightarrow e' : \tau}{\Delta \vdash \texttt{check}(e, \tau) \Rightarrow \boxed{??} : \tau} \qquad \frac{\Delta \vdash e \Rightarrow e' : \texttt{Tagged}}{\Delta \vdash \texttt{check}(e, \tau) \Rightarrow \boxed{??} : \tau}$$

Question: Translation 4: Odd Cases

The last few cases correspond to programs that contain what might look like type errors in a statically-typed language. In Extended Dynamic MinML, however, these expressions are well-formed and must be handled sensibly by any translation.

Part 1: Fill in the blanks in these rules and answer the question below. (Note: There are analogous additional rules to cover some other combinations, but this should give you the idea.)

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \tau_1 \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau_2 \quad \tau_1 \neq \tau \to \tau',\ \tau_1 \neq \texttt{Tagged}}{\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Tagged}}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \tau_1 \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau_2 \quad \tau_1, \tau_2 \notin \{\texttt{Int}, \texttt{Tagged}\} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \tau \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau' \quad \Delta \vdash e_3 \Rightarrow e_3' : \tau' \quad \tau \notin \{\texttt{Bool}, \texttt{Tagged}\}}{\Delta \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi} \Rightarrow \boxed{??} : \tau'}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \tau_1 \to \tau_2 \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau' \quad \tau' \neq \tau_1,\ \tau' \neq \texttt{Tagged}}{\Delta \vdash \texttt{apply}(e_1, e_2) \Rightarrow \boxed{??} : \tau_2}$$

Part 2: Fill in the blank in the following rule. (Note: there is an additional analogous rule to cover the case where e_1 translate at type Tagged.)

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Bool} \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau_2 \quad \Delta \vdash e_3 \Rightarrow e_3' : \tau_3 \quad (\tau_2 \neq \tau_3)}{\Delta \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi} \Rightarrow \boxed{??} : \texttt{Tagged}}$$

5 Implementation

For the programming part of this assignment, you will implement the translation you have just defined. We will give you the following components:

- A parser for Extended Dynamic MinML
- A type checker for Extended Dynamic MinML
- A type checker for statically typed MinMLwith tagged values
- A top-level module providing an interactive loop and a function for processing files, to help you test your code.

You are to fill in the blanks in the structure `Translate` to implement the function `translateExp`, which should translate an Extended Dynamic MinML expression (an ast of type `EDMinML.exp`) to a statically typed MinML expression (an ast of type `TMinML.exp`). There are some helper functions whose bodies you will have to fill in as well.

To help you test your code, we have provided a "driver" in the structure `Top` (`top.sml`); that structure contains an interactive loop function and one that can read Extended Dynamic MinML expressions from a file. Both of these perform the same processing on expressions: they are type-checked to ensure that they are well-formed according to the static semantics presented earlier, then your code is called to translate them into statically typed MinML, and then the resulting expressions are type-checked before being printed on the standard output. The type checking after translation is intended to help you discover bugs in your translation code: if your implementation is correct, the output it produces for well-typed input should *always* be well-typed.

6 Abstract Machines

For this question, you'll have to look back at the definition of the record calculus given in assignment No. 3. A sample solution to assignment No. 3 will be posted on the Web. Here, we simply remind you of the syntax of the record calculus operations.

$$\text{expressions} \quad e \quad ::= \quad \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e_1 \mathbin{++} e_2 \mid e_1/l \mid \texttt{match}\, e_1\, \texttt{with}\, \{l = x\}\, \texttt{in}\, e_2\, \texttt{end}$$

Instead of using the keyword "empty," for the empty record, we will simply use $\{\,\}$. We also allow users to define a record with many fields all at the same time.

1. Extend the definition of C-machine stack frames.
2. Give C-machine rules for the record calculus operations.
3. Describe how to change your C-machine rules so that the record concatenation operator $(e_1 \mathbin{++} e_2)$ first evaluates e_2, then evaluates e_1 and then does the record concatenation.
4. Define E-machine values, frames and stacks for the record calculus operations.
5. Give E-machine rules for the operations in the record calculus.