CS 510 Programming Languages
Assignment 5

1 Introduction

In this assignment, we will explore the semantics of safe, dynamically typed languages such as Scheme or Python. We'll also explore how one can translate such a safe *untyped* language in to a lower-level, safe *typed* language. One of the points here is to show that typed languages are expressive and efficient: Whereas safe, untyped languages require dynamic checks surrounding every critical operation, typed languages can be proven safe, despite the absence of such dynamic checks.

To Hand In

You will hand it two things

- A file theory.txt or theory.pdf containing answers to questions A1, A2, A3 and A4 in this document below.
- A collection of Haskell files that implement well-formedness checking of the untyped language, type checking of the typed language, and the translation from the untyped to typed language.

Availability of Code

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

2 Dynamic ML and Tagged ML

Dynamic ML (DML) is a safe, untyped language that contains first-class, recursive functions, booleans and integers. It's a lot like (a very cut down version of) Scheme or Racket. It's safety is guaranteed primarily by run-time checking rather than static type-checking. Hence, a programmer can write a program that uses the constant "true" as if it is a function, or that uses "3" as if it is a boolean. Such errors are not caught at compile time – they are only caught at run time when the offending code is encountered.

Tagged ML (TML), on the other hand, is a strongly, statically typed language. It also contains recursive functions, booleans and integers and its type system is similar to the type system of the simply typed lambda calculus that we have studied in class. Hence, using the constant "true" as if it were a function is a compile-time error.

However, in addition to these basic types, TML contains the type "`Tagged`." This type is often called the "universal type" in the PL literature because the information content of any typed data structure can be injected in to a data structure with this one type. This `Tagged` type is really a fancy sum type. Intuitively, `Tagged` is a (recursive) sum of either:

- an Int, or
- a Bool, or

- a function from `Tagged` to `Tagged`

The `Tagged` type has constructors that can build values of type `Tagged`. For example, `tag_int`(e) has type `Tagged` if e has type Int. The `Tagged` type also has destructors that can take apart values of type `Tagged`.[1] For example, if v_1 is `tag_int`(v_0), then `as_int`(v_1) strips the tag from v_1 and returns just v_0 (i.e., `as_int`(`tag_int`(v_0)) steps to v_0). On the other hand, if v_1 has a different form, then `as_int`(v_1) raises the `error` exception.

Interesting, even though the type system of TML makes it appear very inflexible when compared with DML, every DML expression can be translated into a TML expression of type `Tagged`. However, the most obvious such translation performs *a lot* of unnecessary tagging and tag-checking. For example, if we attempt to translate the (untyped) expression

$$+(*(2,3),4)$$

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

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

This is clearly ridiculous; in fact, any realistic implementation of Dynamic ML would probably notice that the literals 2, 3 and 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:

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

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

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

a compiler cannot avoid tagging the argument and return value unless it can "guess" that the intended type of the function is `Int` $\rightarrow$ `Int`. Of course, guessing the type of a recursive function may be difficult (though we know that ML and Haskell implement type inference, so it is not always impossible). 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, even though the DML language is officially untyped. In this assignment, you will explore this approach.

Hence, DML, in addition to having dynamic types, will also contain two additional constructs. First, DML will contain *typed* `fun`-expressions as well as *untyped* ones, so that programmers can make clear what types they intend functions to have. Second, DML 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 DML to TML that attempts to take advantage of typing annotations to remove some tagging and tag-checking operations.

3 Formalizing DML

In this section, we walk through the formal syntax, dynamic semantics and static semantics of DML.

3.1 Syntax

The abstract syntax of DML is given below.

[1] We might have chosen to add pattern matching facilities to TML. But we didn't. The translation code winds up being more compact if we just have these special destructors in TML.

Values	v	::=	$n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\, f(x) = e\, \texttt{end} \mid \texttt{fun}\, f(x{:}\tau_1) : \tau_2 = e\, \texttt{end}$
Expressions	e	::=	$x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi}$
		$\mid$	$o(e_1, \ldots, e_n) \mid \texttt{fun}\, f(x) = e\, \texttt{end} \mid \texttt{fun}\, f(x{:}\tau_1) : \tau_2 = e\, \texttt{end}$
		$\mid$	$e_1(e_2) \mid \texttt{check}(e, \tau)$
Types	τ	::=	$\texttt{Int} \mid \texttt{Bool} \mid \tau_1 \to \tau_2$
Operators	o	::=	$\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<}$
Programs	P	::=	e

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.

3.2 Static Semantics

The static semantics for DML are a little different from those of ordinary simply typed lambda calculus. Rather than define a judgment of the form "$\Gamma \vdash e : \tau$", meaning "e has type τ in context Γ," we will instead define a looser judgment:

- $\Delta \vdash_{\mathsf{d}} e\ \mathsf{wf}$, 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 DML from those of ordinary ML.) The contexts (written Δ) used in these rules are themselves a bit different from their statically-typed ML counterparts. Rather than assigning types to variables as ordinary ML contexts do, DML contexts simply keep track of untyped variables. These contexts are given by the following grammar:

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

The context item $x\ \mathsf{val}$ should be read "x is valid", and it means that the variable x has been bound somewhere but its type is not known. The context item $x{:}\tau$ is the standard typing hypothesis. The latter item won't be used in the static semantics for DML, but it will be used later in the translation from DML to TML.

The above judgment is defined inductively by the following inference rules.

$$\frac{(x\ \mathsf{val}) \in \Delta}{\Delta \vdash_{\mathsf{d}} x\ \mathsf{wf}}\ (\text{T0})$$

$$\frac{}{\Delta \vdash_{\mathsf{d}} n :\ \mathsf{wf}}\ (\text{T1}) \qquad \frac{}{\Delta \vdash_{\mathsf{d}} \texttt{true} :\ \mathsf{wf}}\ (\text{T2}) \qquad \frac{}{\Delta \vdash_{\mathsf{d}} \texttt{false} :\ \mathsf{wf}}\ (\text{T3})$$

$$\frac{\Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_2\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} o(e_1, e_2) :\ \mathsf{wf}}\ (\text{T4}) \qquad \frac{\Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_2\ \mathsf{wf} \quad \Delta \vdash_{\mathsf{d}} e_3\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi}\ \mathsf{wf}}\ (\text{T5})$$

$$\frac{\Delta \vdash_{\mathsf{d}} e_1\ \mathsf{wf} \quad v \vdash_{\mathsf{d}} e_2\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} e_1(e_2)\ \mathsf{wf}}\ (\text{T6}) \qquad \frac{\Delta, f\ \mathsf{val}, x\ \mathsf{val} \vdash_{\mathsf{d}} e\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \texttt{fun}\, f(x) = e\, \texttt{end}\ \mathsf{wf}}\ (f, x \notin \Delta)\ (\text{T7})$$

$$\frac{\Delta \vdash_{\mathsf{d}} \texttt{fun}\, f(x) = e\, \texttt{end}\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \texttt{fun}\, f(x{:}\tau_1) : \tau_2 = e\, \texttt{end} :\ \mathsf{wf}}\ (f, x \notin \Delta)\ (\text{T8}) \qquad \frac{\Delta \vdash_{\mathsf{d}} e\ \mathsf{wf}}{\Delta \vdash_{\mathsf{d}} \texttt{check}(e, \tau) :\ \mathsf{wf}}\ (\text{T9})$$

Notice that the well-formedness judgment is much more "permissive" than the typing judgment of the simply typed lambda calculus. As you will see in the next section, the dynamic semantics handles cases that the STLC type system would rule out, such as adding an integer and a boolean.

3.3 Dynamic Semantics

The dynamic semantics of DML is shown here for your reference. You will not have to implement an interpreter for this language, but when you write your translation from DML to TML, you will have to be sure that the behavior of programs is not changed by the translation.

In order to specify the operational semantics, we need one additional expression form, "`error`," which we use to flag a dynamic type-checking failure.

$$e ::= \cdots | \texttt{error}$$

The form of the operational judgement is $e \mapsto e$.

Instructions

$$\frac{(n = n_1 + n_2)}{\texttt{+}(n_1, n_2) \mapsto n} \text{ (O1)} \qquad \text{(and similarly for -,*)}$$

$$\frac{(b = (n_1 = n_2))}{\texttt{=}(n_1, n_2) \mapsto \texttt{Bool}(b)} \text{ (O2)} \qquad \text{(and similarly for <)}$$

$$\frac{}{\texttt{if true then } e_2 \texttt{ else } e_3 \texttt{ fi} \mapsto e_2} \text{ (O3)} \qquad \frac{}{\texttt{if false then } e_2 \texttt{ else } e_3 \texttt{ fi} \mapsto e_3} \text{ (O4)}$$

$$\frac{(v_1 = \texttt{fun } f(x) \ = \ e \texttt{ end})}{v_1(v_2) \mapsto \{v_1, v_2/f, x\}e} \text{ (O5)} \qquad \frac{v_1 = \texttt{fun } f(x{:}\tau_1) : \tau_2 \ = \ e \texttt{ end}}{v_1(v_2) \mapsto \texttt{fun } f(x) \ = \ e \texttt{ end}(\texttt{check}(v_2, \tau_1))} \text{ (O6)}$$

$$\frac{}{\texttt{check}(n, \texttt{Int}) \mapsto n} \text{ (O7)} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\texttt{check}(b, \texttt{Bool}) \mapsto b} \text{ (O8)}$$

$$\frac{}{\texttt{check}(v, \tau_1 \to \tau_2) \mapsto \texttt{fun } f(x{:}\tau_1) : \tau_2 \ = \ \texttt{check}(v(x), \tau_2) \texttt{ end}} \text{ (O9)}$$

Search Rules

$$\frac{e_1 \mapsto e_1'}{o(e_1, e_2) \mapsto o(e_1', e_2)} \text{ (O10)} \qquad \frac{e_2 \mapsto e_2'}{o(v_1, e_2) \mapsto o(v_1, e_2')} \text{ (O11)}$$

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \text{ (O12)} \qquad \frac{e_2 \mapsto e_2'}{v_1(e_2) \mapsto v_1(e_2')} \text{ (O13)}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} \mapsto \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi}} \text{ (O14)} \qquad \frac{e \mapsto e'}{\texttt{check}(e, \tau) \mapsto \texttt{check}(e', \tau)} \text{ (O15)}$$

Error Transitions

$$\frac{v_1 \text{ isnt\_int}}{o(v_1, v_2) \mapsto \texttt{error}} \text{ (O16)} \qquad \frac{v_2 \text{ isnt\_int}}{o(v_1, v_2) \mapsto \texttt{error}} \text{ (O17)}$$

$$\frac{v_2 \text{ isnt\_fun}}{v_1(v_2) \mapsto \texttt{error}} \text{ (O18)} \quad \frac{v_1 \text{ isnt\_bool}}{\texttt{if } v_1 \texttt{ then } e_2 \texttt{ else } e_2 \texttt{ fi} \mapsto \texttt{error}} \text{ (O19)}$$

$$\frac{v \text{ isnt\_int}}{\texttt{check}(v, \texttt{Int}) \mapsto \texttt{error}} \text{ (O20)} \quad \frac{v \text{ isnt\_bool}}{\texttt{check}(v, \texttt{Bool}) \mapsto \texttt{error}} \text{ (O21)}$$

Error Propagation

$$\frac{}{o(\texttt{error}, e) \mapsto \texttt{error}} \text{ (O22)} \quad \frac{}{o(v, \texttt{error}) \mapsto \texttt{error}} \text{ (O23)}$$

$$\frac{}{\texttt{error}(e) \mapsto \texttt{error}} \text{ (O24)} \quad \frac{}{v(\texttt{error}) \mapsto \texttt{error}} \text{ (O25)}$$

$$\frac{}{\texttt{if error then } e_2 \texttt{ else } e_3 \texttt{ fi} \mapsto \texttt{error}} \text{ (O26)} \quad \frac{}{\texttt{check}(\texttt{error}, \tau) \mapsto \texttt{error}} \text{ (O27)}$$

3.4 Progress and Preservation

Dynamic ML should satisfy variants of our progress and preservation lemmas.

Definition 1 (Stuck State)**.** *A state s is* stuck *if*

- *s is not a value, and*
- *s is not* `error`*, and*
- *s is not an expression e that can take an execution step.*

Lemma 1 (Progress)**.** *If $\vdash_{\mathsf{d}} e$ wf then e is not stuck.*

Lemma 2 (Preservation)**.** *If $\vdash_{\mathsf{d}} e$ wf and $e \mapsto e'$ then $\vdash_{\mathsf{d}} e'$ wf.*

Answer the following questions. You will probably want to do questions A3 and A4 before A2.

Question: A1

What kinds of DML programs fail to type check? Explain in a single sentence.

Question: A2

There is either a missing typing rule. What rule is missing? Please write out a rule with all the necessary conditions that will complete the semantics properly. Which theorem(s) (Progress and/or Preservation) fail due to the absence of this rule? Explain exactly which specific case of the proof of either Progress or Preservation can't be proven without the rule and why. (If multiple cases of the proof can't be proven, you only need to refer to one case of the proof in your explanation.)

Question: A3

Give the proof for the cases of the Progress Lemma that involve the typing rule T6. Do not show the proofs of the other cases. If you require an additional rule you specified in question A2, then use it. If you require auxiliary lemmas such as Exchange, Weakening, Canonical Forms, Inversion, Substitution or other similar lemmas then carefully write down the statement of the lemma that you require, but do not bother to write out the proof for it.

Question: A4

Give the proof for the cases of the Preservation Lemma that involve operational rules O5, O7, and O11. Do not show the proofs of the other cases. Be clear about what you must prove in each case. If you require the additional rule you specified in question A2, then use it. If you require auxiliary lemmas such as Exchange, Weakening, Canonical Forms, Inversion, Substitution or other similar lemmas then carefully write down the statement of the lemma that you require, but do not bother to write out the proof for it.

4 Formalizing TML

In this short section, we will present the syntax and semantics of the variant of ML 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 the simply typed lambda calculus with recursive functions, booleans, integers:

Values	v	::=	$n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\, f(x) = e\, \texttt{end} \mid \texttt{fun}\, f(x{:}\tau_1) : \tau_2 = e\, \texttt{end}$
		$\mid$	$\texttt{tag\_int}(v) \mid \texttt{tag\_bool}(v) \mid \texttt{tag\_fun}(v)$
Expressions	e	::=	$x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi}$
		$\mid$	$o(e_1, \ldots, e_n) \mid \texttt{fun}\, f(x{:}\tau_1) : \tau_2 = e\, \texttt{end} \mid e_1(e_2)$
		$\mid$	$\texttt{tag\_int}(e) \mid \texttt{tag\_bool}(e) \mid \texttt{tag\_fun}(e)$
		$\mid$	$\texttt{as\_int}(e) \mid \texttt{as\_bool}(e) \mid \texttt{as\_fun}(e)$
Types	τ	::=	$\texttt{Int} \mid \texttt{Bool} \mid \tau_1 \to \tau_2 \mid \texttt{Tagged}$
Operators	o	::=	$\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<}$

The static semantics of the language is the standard static semantics for the simply-typed lambda calculus with booleans, integers and recursive functions extended with the following rules:

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

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

The dynamic semantics of the simply-typed lambda calculus with recursive functions, integers and booleans is extended with the following rules (plus rules for propagation of `error`):

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

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

(and similarly for the other tag operations).

5 Translation

In this section, you will design a translation from DML to TML. Your goal is to 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* (using the rules of the simply typed lambda calculus). Only if no standard type (ie, Int, Bool, or $\tau_1 \to \tau_2$) can be given to an expression will it be translated to an expression with the type `Tagged`.

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." Your translation should maintain the following invariant:

Lemma 3. *If* $\Delta \vdash e_d \Rightarrow e_s : \tau$ *(where* τ *may be* `Tagged`*), and* $|\Delta|$ *is the ML context obtained from* Δ *by replacing bindings of the form* x `val` *with ones of the form* x:`Tagged`*, then* $|\Delta| \vdash e_s : \tau$.

You do not have to prove the lemma, but think about how you would do it. (By induction on???)

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. We recommend that you read through this description and fill in the holes in the description we have left for you before beginning to code up a solution.

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.

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

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

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

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

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

Translation 2: Pure Tagged Cases

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.) We have left some blanks in the Pure Tagging Cases for you to fill in.

$$\frac{(x\ \mathsf{val}) \in \Delta}{\Delta \vdash x \Rightarrow x : \texttt{Tagged}} \qquad \frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged} \quad (op \in \{\texttt{+}, \texttt{-}, \texttt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}} \text{ (B1)}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Int} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged} \quad (op \in \{\texttt{+}, \texttt{-}, \texttt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}} \text{ (B2)}$$

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Int} \quad (op \in \{\texttt{+}, \texttt{-}, \texttt{*}\})}{\Delta \vdash op(e_1, e_2) \Rightarrow \boxed{??} : \texttt{Int}} \text{ (B3)}$$

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

$$\frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \tau \quad \Delta \vdash e_3 \Rightarrow e_3' : \tau}{\Delta \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3\, \texttt{fi} \Rightarrow \boxed{??} : \tau} \text{ (B4)}$$

$$\frac{\Delta, f\ \mathsf{val}, x\ \mathsf{val} \vdash e \Rightarrow e' : \texttt{Tagged} \quad (f, x \notin \mathrm{dom}(\Delta))}{\Delta \vdash \texttt{fun}\, f(x)\ =\ e\, \texttt{end} \Rightarrow \boxed{??} : \texttt{Tagged}} \text{ (B5)} \qquad \frac{\Delta \vdash e_1 \Rightarrow e_1' : \texttt{Tagged} \quad \Delta \vdash e_2 \Rightarrow e_2' : \texttt{Tagged}}{\Delta \vdash e_1(e_2) \Rightarrow \boxed{??} : \texttt{Tagged}} \text{ (B6)}$$

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:

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

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:

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

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:

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

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 ML 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:

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

In order to implement the translation, 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:

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

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 DML, 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.)

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

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

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

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

Part 2: Figure out how to 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`.)

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

6 Implementation

For the programming part of this assignment, you will implement the translation you have just defined as well as some utility routines. You must implement the following functions and submit the following files:

File	Functions to implement	Description
`edtyping.hs`	`check_wf`	Well-formedness checking of the untyped language.
`ttyping.hs`	`typing`	Type checking of the typed language.
`translate.hs`	`tagify`, `cast`, `translateExp`	Translation from the typed to untyped language.

Availability of Code

You can find a directory full of code off of the course web site. Once you have implemented the functions above, use `make` to build an executable named `dminml`.

```
usage: dminml <filename>
```

The `dminml` executable will check the well-formedness of a DML program, translate it to TML, type check the resulting TML program, and evaluate the TML program. This directory also contains sample DML files to test your implementation: `fact_typed.dml`, `fact_untyped.dml`, `malformed.dml`, `simple_untyped.dml`, and `test.dml`. You should create additional DML programs to more thoroughly evaluate your implementation (although you are not required to submit additional tests).

Note: As part of this assignment, you will implement a type checker for TML, and the `dminml` executable will invoke it to type check your translation. Type checking the result of your translation should help you find errors in your translation. If your implementation is correct, the output it produces from well-typed input should *always* be well-typed. The use of "typed intermediate languages" in compilers is a good, general way of ensuring that compiled code is safe. They help detect many foolish errors in a compiler, just like a type checker for a source programming language does.