Assignment 6

Due at 8:00pm, Fri 1 Nov

In this assignment, we will explore the connections between logic and computation (Lecture 14) and define and implement the type system of the Gradual ALFA language (Lecture 15), which extends the ALFA language from Assignment 4 with support for gradual typing. You will submit answers to the 1 programming exercise using Learn OCaml and the 3 written problem using Gradescope. Your late day is determined by your Gradescope submission time.

Part I
Logic and Computation

1 Constructive Propositional Logic

Hypothetical reasoning is central to human cognition. First-order propositional logic captures the essence of hypothetical reasoning by way of the *hypothetical judgement*, $\Gamma \vdash A$, which specifies how to derive that a proposition A is true if we assume, without proof, the truth of a finite set of propositions Γ of the following form, up to reordering, for $n \geq 0$:

$$A_1, \cdots, A_n$$

You can pronounce the judgement $\Gamma \vdash A$ as "Γ entails A". We call the propositions in Γ the "hypotheses" or "assumptions".

Assumptions The simplest way to conclude that a proposition A is hypothetically true is if we are already hypothesizing it is true! This is called proof-by-assumption.

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ \text{(assumption)}$$

Conjunctions We can conclude that a conjunction, written $A \wedge B$, is true if both A and B are true assuming the given hypotheses. This rule is known as the "conjunction introduction" rule.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge\text{-I})$$

Inversely, if we know that a conjunction is true, we can conclude that both of the constituent propositions must be true. These are known as the left and right "conjunction elimination" rules.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge\text{-E-L}) \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge\text{-E-R})$$

Disjunctions We can conclude that a disjunction, $A \vee B$, is true in two ways: if we can conclude that A is true, or if we can conclude that B is true. In other words, there are two "disjunction introduction" rules, one for when we establish the disjunction from the left, the other from the right. (If both sides can be derived, then we can choose a rule arbitrarily.)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee\text{-I-L}) \qquad\qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee\text{-I-R})$$

If we know that a disjunction $A \vee B$ is true, then we know that either A or B is true. From this, we can conclude some other proposition C is true if it can be derived in both cases. In other words, we can reason by exhaustive case analysis. This is called the "disjunction elimination" rule. Notice how we add A and B as assumptions.

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \ (\vee\text{-E})$$

Implication Implication, written $A \supset B$, allows us to express hypotheticals directly as propositions. In particular, to prove that A implies B, written $A \supset B$, we add A to our assumptions. If we can now establish B, then the implication is true under the original set of assumptions. (A slogan to think about here is "implication internalizes entailment".)

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \ (\supset\text{-I})$$

Implication tells us nothing about the truth of A under Γ. We are only taking A as an additional hypothesis and reporting that B would hypothetically be true in a world where A were true. It is only if we also discover that A is true under Γ that we can apply the "implication elimination" rule to conclude B.

$$\frac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B} \ (\supset\text{-E})$$

(What if we know that both $A \supset B$ and B are true? That doesn't tell us anything about the truth of A! It just tells us that it wasn't necessary to assume A to prove B.)

Tautologies Some propositions are tautologies, i.e. true under any set of assumptions. We model these using the proposition $\top$, pronounced "truth".

$$\frac{}{\Gamma \vdash \top} \ (\top\text{-I})$$

We can prove $A \supset \top$ for any proposition A (try it!), but of course this is not a very interesting implication and certainly tells us nothing about the truth of A!

Indeed, knowing that $\top$ is true is never very useful, because that is its very nature! Consequently, there is no elimination rule for $\top$.

Absurdities We can also define a proposition $\bot$, pronounced "absurdity", for which there is no introduction rule, i.e. there is no way to prove it under an empty context!

What good is this proposition? Well, although a proof of this proposition is impossible in "base reality", i.e. without taking any assumptions, we are free to reason hypothetically by putting any proposition we wish into Γ, true or not! So it is possible to derive $\bot$ under absurd contexts, e.g. a context where we've taken $\bot$ as an assumption, or less obviously, one where $\bot$ shows up inside another assumption from which we can extract it using elimination rules.

For example, we can prove the following (for all A and B).

$$A \supset \bot, B \supset \bot, A \vee B \vdash \bot$$

You may wish to derive this judgement using the rules above as practice for the problems below. The solution is at the end of handout.

Although the assumptions cannot themselves be proven independently, that doesn't change the fact that if we nevertheless make these "absurd" assumptions, we can draw the absurd conclusion that the false proposition is true.

To really bring home the point that making absurd assumptions is bad because it allows you to draw absurd conclusions, propositional logic includes the following "falsity elimination rule" (also known as the "principle of explosion", *ex falso [sequitur] quodlibet* (Latin for "from falsehood, anything [follows]"), or the "when pigs fly" rule), which allows you to conclude any proposition you wish if you have somehow proven the false proposition under Γ.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot\text{-E})$$

Propositional Negation If you are familiar with logic, you may have noticed that so far, we have not defined rules for a "not" connective. In fact, we do not do so in propositional logic! Instead, we can define "not A", written $\neg A$, as syntactic sugar for $A \supset \bot$. After all, if "not A" is true, then A cannot also be true, and the combination of the two is an absurdity!

$$\neg A = A \supset \bot$$

2 Classical Propositional Logic

Axiom of the Excluded Middle "Constructive logics", described above, differ from "classical logics" in that they do not include the following axiom, known as the "axiom (or law) of the excluded middle", which says that for every proposition A, the disjunction A or $\neg A$ is true.

$$\frac{}{\Gamma \vdash A \vee \neg A}\ (\text{AEM})$$ (only in classical logic, not in constructive logic)

Why do constructive logicians not accept this rule? It seems intuitive enough, right? Well, the problem is right there in the name of the logic: this rule isn't constructive. By that, I mean that taking this as an axiom means that we have a way to derive a disjunction without needing to construct a derivation for either the left or the right side of the disjunction, as we must for all other disjunctions! What if we discover and add a proposition that is independent of our logic, i.e. one for which neither the truth of A nor the truth of $\neg A$ (which, recall, is an implication), can be derived? We've axiomatized that one of them is true in contradiction of the fact that there is no derivation of either side! The conceit is to treat the truth of a proposition as a boolean value, for which there are indeed only two possibilities. Instead, in constructive logic, a proposition is a primitive structure, and we are obliged to remain neutral until we construct direct evidence of either the truth of the proposition, or the truth of its negation!

3 Boolean Logic

In boolean logics, i.e. those where propositions are really just expressions that evaluate to boolean values, negation is taken as primitive and implication is defined as syntactic sugar in terms of negation as follows:

$$A \supset B = \neg A \vee B$$ (in boolean logic only!)

In propositional logic, we take the opposite perspective, defining implication directly, and $\neg A$ as syntactic sugar, as discussed above. However, in the problem below we'll see how classical propositional logic has the same essence as boolean logic via the AEC.

Exercise 1 (Boolean Logic, Propositionally, Written, 20%)

Part 1. Show that in **classical** propositional logic, the following implication holds (for arbitrary A and B):

$$\vdash (A \supset B) \supset (\neg A \vee B)$$

Provide a derivation of this judgement using the rules of **classical** propositional logic, i.e. any of the rules in the write-up above including (AEM), but **NOT** using the definition of implication for boolean logic just given.

Part 2. Can you provide a derivation for this judgement in **constructive** propositional logic, i.e. without using the (AEM) rule? If so, do so. If not, show **two** different examples of incomplete derivations where you've gone as far as you can and for each explain what goes wrong, i.e. why there are no rules left to apply.

Part 3. Not every rule from boolean logic fails to hold constructively. Derive the following judgement, which states De Morgan's rule for negation of disjunctions, using the rules of **constructive** logic (i.e. without using the (AEM) rule).

$$\vdash \neg(A \vee B) \supset (\neg A \wedge \neg B)$$

Part 4 (5% extra credit). Another rule you may be familiar with from boolean logic is *double negation elimination.* This too relies on the conceit that propositions are boolean values, and therefore it is not constructively derivable. Show, however, that the following judgement can be derived in **classical** propositional logic (for arbitrary A):

$$\vdash \neg\neg A \supset A$$

Part 5 (5% extra credit). Although constructive logic does not include the AEM axiom, it also does not deny its truth! In other words, it is *not not* the case that AEM is true! Prove this by deriving the following judgement in **constructive** logic (i.e. not using AEM, but for arbitrary A):

$$\vdash \neg\neg(A \vee \neg A)$$

This is a surprising but powerful fact, because it means that constructive logic is strictly more powerful than classical logic. If you want to avoid using the questionable AEM, then you can. If you want to use it, you are also free to do so by adding it as an assumption – doing so does not in and of itself create an absurdity. We say that the AEM is independent of constructive logic.

HINT: One way to prove this involves using Part 3 as a lemma (i.e. a separately proved proposition used in the proof of another proposition). You can do so in your solution to Part 5, citing its derivation as $\mathcal{D}_{\text{part3}}$ instead of reproducing it inline.j

In any of the parts above, you may define context abbreviations Γ_1, Γ_2, and so on, as well as derivation abbreviations $\mathcal{D}_1$, $\mathcal{D}_2$, and so on (as in the exam, prior assignment solutions, and the appendix) as needed if you need space.

4 Proof Expressions

You might have noticed from Problem 1 that derivations are a bit unwieldy to work with, so it is natural to wonder if there might be a more concise representation of a derivation.

Indeed there is, at least for constructive propositional logic: we can define a language of proof expressions, e. Valid proofs are those for which the proof checking judgement, $\Gamma \vdash e : A$, can be derived. This judgement can be read "e is a proof of A assuming Γ". To allow the proof expression e to refer to specific assumptions in Γ, we associate each with a variable, x, so Γ is now a set of $n \geq 0$ assumptions written as follows, up to reordering:

$$x_1 : A_1, \cdots, x_n : A_n$$

Before giving the rules, we'll state the following theorem, which connects valid proof expressions to the hypothetical judgement just defined in the previous section.

Theorem 1. *If* $\Gamma \vdash e : A$ *then* $\Gamma \vdash A$*.*

In other words, it says that if e is a proof of proposition A assuming Γ, then A is true assuming Γ according to the rules in the previous section (implicitly, ignoring the variables that we added to Γ to give names to assumptions.) The reason this is true is that e captures all of the rules applied in the derivation, in the form of an expression.

Let us go through each of the rules from the previous section, equipping it with a proof expression, and we'll see how we are hardly treading new ground at all!

Assumptions If we are assuming that a proposition A is hypothetically true, then there is a variable x in Γ that stands for its hypothetical proof. We can use this variable as the proof expression for a proof-by-assumption.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{(assumption)}$$

Conjunctions To prove a conjunction, $A \wedge B$, we need proofs of A and B. Pairing those proofs together serves as our proof of the conjunction and corresponds to the ($\wedge$-I) rule above.

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \wedge B}\ (\wedge\text{-I})$$

The elimination rules for conjunction can be identified as the corresponding projections out of the proof of the conjunction.

$$\frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash e.\mathsf{fst} : A}\ (\wedge\text{-E-1}) \qquad \frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash e.\mathsf{snd} : B}\ (\wedge\text{-E-2})$$

Disjunction To prove a disjunction, $A \vee B$, we must make a choice: either we prove A, or we prove B. This corresponds to the left and right introduction rules for disjunctions in the previous section. To unambiguously identify which rule we used, let us leave a mark, either L or R, in the corresponding proof expression. We will call a marked proof expression an *injection* into the disjunction.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{L}\ e : A \vee B}\ (\vee\text{-I-L}) \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{R}\ e : A \vee B}\ (\vee\text{-I-R})$$

If we know that a disjunction $A \vee B$ is true, then we know (constructively) that either A or B is true. From this, we can conclude some other proposition C is true if it can be derived under either of these two hypotheses. In other words, we can reason by case analysis. Our proof expression is therefore a case analysis, where we specify which variables stand for the hypotheticals in the two branches.

$$\frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{L}(x) \to e_1\ \mathsf{else}\ \mathsf{R}(y) \to e_2 : C}\ (\vee\text{-E})$$

Implication To prove an implication, written $A \supset B$, we must write a *proof function*, which takes as input a proof of A and, using it as an assumption (i.e. via some variable x), constructs a proof of B.

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\mathsf{fun}\ x \to e) : A \supset B}\ (\supset\text{-I})$$

We can apply an implication, $A \supset B$, proved by e_1, by supplying a proof, e_2, of A. Doing so gives us a proof of B.

$$\frac{\Gamma \vdash e_1 : A \supset B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B}\ (\supset\text{-E})$$

Tautologies Tautological propositions are always true, so their proofs are trivial, i.e. they have no internal structure, like a 0-tuple.

$$\frac{}{\Gamma \vdash () : \top}\ (\top\text{-I})$$

Absurdities The falsity proposition, $\bot$, has no proof, i.e. there is no introduction rule.

If we somehow manage to derive it anyway from our assumptions, then we can *abort*, because we have made an absurd assumption. This exceptional situation can be expressed as a proof expression of the form $\mathsf{abort}(e)$, where e is the proof of $\bot$ that has shown up. Such an expression can serve as a proof of any proposition at all.

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash \mathsf{abort}(e) : A}\ (\bot\text{-E})$$

Luckily, proofs of falsity can only occur in absurd contexts, e.g. inside proof functions where we have made absurd, unprovable, assumptions!

4.1 Propositions as Types

So as you can see, we can quite neatly define a language of proof expressions for first-order constructive propositional logic. Checking that a proof expression e is a valid proof of proposition A means deriving the judgement $\Gamma \vdash e : A$.

This judgement is eerily similar to the typing judgement for ALFA, and now you see why! Our proof expressions are expressions from ALFA! The proof checking rules are exactly the typing rules from ALFA, if you notice how the types of ALFA correspond directly to the propositions of first-order constructive propositional logic:

1. Product types $A \times B$ correspond to conjunction $A \wedge B$
2. Sum types $A + B$ correspond to disjunction $A \vee B$
3. Arrow types $A \to B$ correspond to implication $A \supset B$
4. The unit type $\mathsf{1}$ corresponds to the tautological proposition, $\top$

In each case, the introduction forms for these types become the proof expressions corresponding to the introduction rules for the corresponding proposition, and similarly, the elimination forms for these types become the proof expressions corresponding to the elimination rules for the corresponding proposition. This terminology is no accident: Gentzen, a logician, invented the principle of introduction and elimination, long before the computational ideas in ALFA had crystalized.

What about the $\bot$ proposition? We did not talk about it previously, but just like unit is the nullary product, there is also a type called `void` (not related to `void` from C++, which is more like the unit type), also known as `0` or the empty type, which serves as a nullary sum: the type where you have no choices of an introduction form. It does have an elimination form, $\mathsf{crash}(e)$, which can have any type at all. Think of it like an (uncatchable) exception.

$$\frac{\Gamma \vdash e : \mathsf{0}}{\Gamma \vdash \mathsf{crash}(e) : \tau} \text{ (T-Crash)}$$

However, you will never actually get a value out of this expression, because e cannot finish evaluating: there is no value of the empty type. Consequently, the empty type is useful for expressing that a function is non-terminating. Usually, non-termination indicates a bug, but we will see next week that sometimes non-termination is intentional, e.g. when you start a web server or an operating system, it goes into an intentional infinite loop awaiting input. Unfortunately, contemporary imperative languages rarely distinguish between the unit return type (returns a trivial value) and the empty return type (never returns). Their designers didn't take EECS 490!

We will not do so here, but you could also define a judgement $e \Downarrow v$ that evaluates a proof expression, producing another proof expression v, which will be an equivalent but perhaps simpler proof of A. This corresponds to evaluation in ALFA, as you might now expect. Similarly, one can speak of equivalence between proofs, corresponding to equivalence between expressions.

Examples of Proof Expressions As an exercise for yourself, you might want to derive the following judgements using the rules above. In the examples below, A, B, and C range over arbitrary propositions.

The proof that $A \wedge B$ implies A requires taking in, as a function argument, the proof of $A \wedge B$ and projecting out its left side, which will be a proof of A:

$$\vdash (\mathsf{fun}\ x \to x.\mathsf{fst}) : (A \wedge B) \supset A$$

We can see that, by the Propositions as Types principle above, the proposition $(A \wedge B) \supset A$ corresponds to the type $A \times B \to A$. The same expression serves both as a proof of this propositon and an expression of this ALFA type!

The proof that implication is transitive is function composition. We can write it in curried style as follows:

$$\vdash (\mathsf{fun}\ f \to \mathsf{fun}\ g \to \mathsf{fun}\ a \to g(f(a))) : (A \supset B) \supset (B \supset C) \supset (A \supset C)$$

The proposition $(A \supset B) \supset (B \supset C) \supset (A \supset C)$ corresponds to the type $(A \to B) \to (B \to C) \to (A \to C)$, and again the proof expression above has the corresponding type.

We could also write in uncurried style using conjunction as follows:

$$\vdash \mathsf{fun}\ in \to \mathsf{fun}\ a \to in.\mathsf{snd}(in.\mathsf{fst}(a)) : (A \supset B) \wedge (B \supset C) \supset (A \supset C)$$

The proposition $(A \supset B) \wedge (B \supset C) \supset (A \supset C)$ corresponds to the type $(A \to B) \times (B \to C) \to (A \to C)$, and again the proof expression has the corresponding type. **Proofs are programs, and propositions are types!**

Exercise 2 (Proof Expressions, Written, 10%)

Part 1. Write the ALFA type corresponding to the proposition from Problem 1, Part 3 (DeMorgan's rule for negation of disjunctions), assuming A and B are arbitrary types.

$$\tau_{\text{dm}} = ??$$

Part 2. Come up with the proof expression, e_{dm}, corresponding to your derivation from Problem 1, Part 3 (DeMorgan's rule for negation of disjunctions), and show that it is correct by providing a derivation, using the rules in Sec. 4, of the following judgement:

$$\vdash e_{\text{dm}} : \neg(A \vee B) \supset (\neg A \wedge \neg B)$$

Part 3. Show that you can also derive the corresponding ALFA typing judgement (using the rules from the Recursive ALFA language definition packet released in the midterm directory, plus the T-Crash rule defined above as necessary).

$$\vdash e_{\text{dm}} : \tau_{\text{dm}}$$

HINT: Your answers to Parts 2 and 3 of this question will closely correspond to each other and to your answer to Problem 1, Part 3!

HINT: If Part 2 seems tricky, try coming up with an expression for which you can derive Part 3 instead!

Part II
Gradual Typing

5 Background

5.1 Dynamic Classification

Many popular languages are "dynamically typed", including Python, JavaScript, Racket, Julia, and MATLAB. These languages do not define a static semantics.[1] Instead, values are tagged with a class during execution and class checking occurs before each primitive operation is performed.[2] For example, evaluating `e1 + e2` in such a language would involve all of the following:

1. Evaluating `e1` and `e2` to (tagged) values `v1<c1>` and `v2<c2>`
2. Checking that the class tags `c1` and `c2` are both `num` (and raising an error if not)
3. Removing the class tags to extract the underlying machine numbers `v1 = n1` and `v2 = n2`
4. Instructing the CPU to add `n1` and `n2`, producing `n`
5. Placing the `num` class tag on `n` to produce the final tagged value, `n<num>`.

[1] You could also think of these languages as having a trivial static semantics, with only one type, `dyn`, classifying every expression. Some people therefore call these languages "unityped".

[2] Because the core safety mechanism is class checking, it is more accurate to call these languages "dynamically classified" or just "dynamic" rather than "dynamically typed". Types are static classifiers of expressions whereas classes are dynamic classifiers of values. For example, the class on a function value, say `fun`, merely tells you that the value is a function; it provides no information about input-output behavior, as a function type like `int -> int` would, because that would require statically analyzing the function body, which is not a value. Help make computing more clear by using different words for different concepts!

This tag-related overhead (items 2, 3, and 5) incurs substantial run-time overhead (10x-100x or more), even when using sophisticated modern run-time systems like V8 for JavaScript that attempt to optimize some tag checks away using run-time (a.k.a. "just-in-time") optimization.

Why are these languages nevertheless quite popular? A comprehensive answer to that question is a long discussion for another time. However, one common argument is that these languages are excellent for prototyping and experimentation because they allow the user to execute programs that would not be accepted by a static type system due to localized but benign errors or inconsistencies. For example, consider the following incomplete OCaml function, implementing some of the logic of a combat game:

```
let calc_damage player attack =
  match (player, attack) with
  | (Wizard, Melee n) -> (10 * n, "oh my!")
  | (Mage, Melee n) -> 9 * n
  | _ -> failwith "Unimplemented"
```

This is not a well-typed function because the two branches of the **match** expression have different types (`int * string` for the first branch, and `int` for the second branch). Consequently, you cannot execute any program containing this function definition, even if you are only interested in the behavior of its first branch. For example, you might be in the midst of refactoring `calc_damage` to include a message like `"oh my!"` for each attack. You may want to test the updated behavior for wizard characters, before you have updated the logic for all of the other character classes.

A hypothetical dynamically classified dialect of OCaml–let's call it Dynamic OCaml–would not check that all branches of the **match** expression have the same type. Both of the following expressions would therefore evaluate as expected:

```
calc_damage Wizard (Melee 10)  (* evaluates to (100, "oh my!") *)
calc_damage Mage (Melee 10) (* evaluates to 90 *)
```

It is only if the resulting value is subsequently used in a class-inconsistent manner that there would be a run-time error. For instance, there would be an error during evaluation of the expression `dmg * 2` due to a class mismatch below:

```
let dmg = calc_damage Wizard (Melee 10) in
dmg * 2
(* Run-time class error: dmg has the tuple class, expected int class *)
```

Dynamic classification imposes more than just performance overhead. It also imposes a reasoning burden on client programmers, because without types, there is no simple, formal way of summarizing a function's interface. In this example, the output value's class depends in a complex way on the input *value*. To reason at all about the function's behavior, clients must carefully read its implementation (or hope that the informal documentation is sufficiently comprehensive, precise, and up-to-date). Consequently, it is known to be difficult to build modular, large-scale systems in languages like these. Twitter, for example, had to rewrite their software from Ruby (a dynamically classified language) to Scala (a statically typed functional language similar to OCaml) early in its development due to these performance and reasoning issues.

The difficulties of static reasoning about functions in dynamic languages also makes building editor tooling for these languages very difficult, because the tool needs to be able to provide hints and services without running the code. Most modern tooling for dynamically tagged languages uses *ad hoc* and error-prone heuristics and approximate static analyses.

6 Gradual Typing

Due to the difficulty of optimizing and reasoning about programs written in dynamically classified languages, it has become increasingly clear that they are not ideal for building large software systems. Nevertheless, language designers have come to recognize the value of allowing certain static type checks to be omitted, with safety instead being maintained by corresponding run-time tag checks, during prototyping and exploratory programming activities.

An important observation is that prototypes often evolve continuously into production systems. For this reason, there has been surging interest in *gradual type systems*, which are static type systems that allow the programmer to explicitly leave the type of an expression unknown if they so choose. Values of unknown type (but not of known types) have run-time tags, and tag checks are inserted only when operating on values of unknown type. This allows the programmer to make use of the flexibility of dynamic tagging during prototyping while gradually shifting toward full static typing as the system grows and matures, without having to rewrite the code in a completely different language at any point during this process.

There are a number of industry-supported gradually typed languages seeing increasing use today, including TypeScript (Microsoft's gradually typed dialect of JavaScript), Hack (Facebook's dialect of PHP), mypy (Dropbox's dialect of Python), and Sorbet (Stripe's dialect of Ruby). Some statically typed languages have also added support for features related to gradual typing, e.g. C♯'s `dynamic` type. Hazel is also a gradually typed functional language being developed here at Michigan.

Consider a hypothetical gradually typed dialect of OCaml–let's call it Gradual OCaml. The programmer can explicitly mark the return type of `calc_damage` as unknown[3] by writing a *type hole*, notated ?, while letting OCaml's usual type inference system determine the types of the two inputs:

```
let calc_damage (player) (attack) : ? (* unknown return type *) =
  match (player, attack) with
  | (Wizard, Melee n) -> (10 * n, "oh my!")
  | (Mage, Melee n) -> 9 * n
  | _ -> failwith "Unimplemented"
```

The branches of the **`match`** expression in the body of this function can now be of any type at all – here `int * string` in the first branch and `int` in the second – because **every type is considered type-consistent with the unknown type**. To compensate for this permissiveness at compile-time, the return value of `calc_damage` is tagged at run-time. Expressions that operate on a tagged value must first perform a tag check. For example, all of the examples above would also work in the same way in Gradual OCaml (whereas they are ill-typed in OCaml). However, there would be no tag-related overhead for the fully statically typed portions of the program.

7 Gradual ALFA

To more deeply understand how gradual typing works, let us develop a gradually typed dialect of our ALFA language from Assignment 4. Our focus will not be on the mechanics of tag checking (which are straightforward but tedious), but rather on the static semantics. The key learning goal here is to understand how we can perform static type checking for the parts of the program where we have a known static type while being permissive when checking expressions that the programmer specifies to be of unknown type via a *type hole*, written ?.

[3] In some other gradually typed languages, the unknown type is the default; the programmer instead explicitly specifies when the type is *known*. In OCaml, type inference is powerful enough that this would not be sensible.

If we are allowing the programmer to omit types, why not also allow them to omit expressions? Good point! While we're adding type holes to ALFA, we'll also go ahead and add expression holes, written _. We discuss expression holes, and how they relate to type holes, in more detail below.

7.1 Syntax

We define the structural syntax and the concrete syntax for Gradual ALFA in Fig. 1. There are two new forms, type holes and expression holes, which are the focus of this assignment.

OCaml datatypes that implement these structures will be provided for you in the exercise. We have organized them into modules each with a type called `t`, so the type `Exp.t` classifies the OCaml values corresponding to Gradual ALFA expressions. As in previous assignments, we write $\lfloor e \rfloor$ to refer to the OCaml implementation of the Gradual ALFA expression e, and similarly for other structures in our syntax. For example,

$$\lfloor \mathsf{fun}\ (x : ?) \to \_ \rfloor = \texttt{Exp.EFun("x", Some Typ.THole, Exp.EHole)}$$

In addition, we will provide a parser and pretty-printer that implements a textual syntax that closely follows the concrete syntax shown in Fig. 1. You can inspect the definition of this textual syntax at the following URL:

`https://github.com/eecs490/parsers/tree/master/gradual-alfa`

7.2 Type System

We will implement a local type inference system for Gradual ALFA. Recall from A4 that we can specify a local type inference system by defining our type system in a *bidirectional* style, which is a type system specified by two mutually defined typing judgements:

- The type synthesis judgement, $\Gamma \vdash e \Rightarrow \tau$, can be read "$e$ synthesizes type τ assuming Γ". The type τ is unique for any given expression e and typing context Γ. Algorithmically, the type is an output.
- The type analysis judgement, $\Gamma \vdash e \Leftarrow \tau$, can be read "$e$ analyzes against τ assuming Γ". Expressions can be analyzed by more than one type. Algorithmically, the type is an input and the output is a boolean indicating whether type analysis against that type succeeds.

Variables The following rule establishes that variables synthesize the type that Γ provides.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}\ \text{(S-Var)}$$

Literals The following rules establish that number literals synthesize type Num and that the two boolean literals synthesize type Bool in all contexts.

$$\frac{}{\Gamma \vdash \underline{n} \Rightarrow \mathsf{Num}}\ \text{(S-NumLiteral)} \qquad \frac{}{\Gamma \vdash \mathsf{True} \Rightarrow \mathsf{Bool}}\ \text{(S-True)} \qquad \frac{}{\Gamma \vdash \mathsf{False} \Rightarrow \mathsf{Bool}}\ \text{(S-False)}$$

Subsumption Rather than also defining analysis rules for every form, we can in many cases turn to a general subsumption rule, which allows us to analyze any expression e against the type it synthesizes, τ, or in Gradual ALFA, any *consistent* type, τ':

$$\frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau'}\ \text{(A-Subsumption)}$$

			Structural Syntax	**Concrete Syntax**
Typ	τ	$::=$	Num	Num
		$\mid$	Bool	Bool
		$\mid$	$\mathsf{Arrow}(\tau,\tau)$	$\tau \to \tau$ (right assoc., precedence 1)
		$\mid$	$\mathsf{Prod}(\tau,\tau)$	$\tau \times \tau$ (right assoc., precedence 3)
		$\mid$	Unit	Unit or on paper, 1
		$\mid$	$\mathsf{Sum}(\tau,\tau)$	$\tau + \tau$ (right assoc., precedence 2)
		$\mid$	TypHole	?
Expr	e, v	$::=$	$\mathsf{NumLit}[n]$	$\underline{n}$
		$\mid$	$\mathsf{Neg}(e)$	$-e$ (prefix, precedence 7)
		$\mid$	$\mathsf{Plus}(e_1, e_2)$	$e_1 + e_2$ (left assoc., precedence 3)
		$\mid$	$\mathsf{Minus}(e_1, e_2)$	$e_1 - e_2$ (left assoc., precedence 3)
		$\mid$	$\mathsf{Times}(e_1, e_2)$	$e_1 * e_2$ (left assoc., precedence 4)
		$\mid$	$\mathsf{Eq}(e_1, e_2)$	$e_1 =?e_2$ (left assoc., precedence 2)
		$\mid$	$\mathsf{Lt}(e_1, e_2)$	$e_1 < e_2$ (left assoc., precedence 2)
		$\mid$	$\mathsf{Gt}(e_1, e_2)$	$e_1 > e_2$ (left assoc., precedence 2)
		$\mid$	True	True
		$\mid$	False	False
		$\mid$	$\mathsf{If}(e_1, e_2, e_3)$	if e_1 then e_2 else e_3 (prefix, precedence 0)
		$\mid$	$\mathsf{Var}[x]$	x
		$\mid$	$\mathsf{Let}(e_1, x.e_2)$	let x be e_1 in e_2 (prefix, precedence 0)
		$\mid$	$\mathsf{Fix}(\tau, x.e)$	fix $(x : \tau) \to e$ (prefix, precedence 0)
		$\mid$	$\mathsf{Fun}(\tau_{\mathsf{in}}, x.e)$	fun $(x : \tau_{\mathsf{in}}) \to e$ (prefix, precedence 0)
		$\mid$	$\mathsf{Ap}(e_1, e_2)$	$e_1\ e_2$ (left assoc., precedence 6)
		$\mid$	$\mathsf{Pair}(e_1, e_2)$	(e_1, e_2)
		$\mid$	Triv	$()$
		$\mid$	$\mathsf{PrjL}(e)$	e.fst (postfix, precedence 8)
		$\mid$	$\mathsf{PrjR}(e)$	e.snd (postfix, precedence 8)
		$\mid$	$\mathsf{LetPair}(e_1, x.y.e_2)$	let (x, y) be e_1 in e_2 (prefix, precedence 0)
		$\mid$	$\mathsf{InjL}(e)$	L e (prefix, precedence 5)
		$\mid$	$\mathsf{InjR}(e)$	R e (prefix, precedence 5)
		$\mid$	$\mathsf{Case}(e, x.e, y.e)$	case e of $\mathsf{L}(x) \to e$ else $\mathsf{R}(y) \to e$ (prefix, precedence 0)
		$\mid$	ExpHole	_

Figure 1: Syntax of Gradual ALFA

Type Consistency Type consistency is a weakened form of equivalance: two types are consistent if they differ only up to the appearance of an unknown type.

For example, $? \to \mathsf{Num} \sim ? \to ?$ because both types are arrow types with the same input type and at the output type, they differ only in that there is an unknown type on the right hand side. Type consistency is symmetric, so it is also the case that $? \to ? \sim ? \to \mathsf{Num}$.

The type consistency judgement is defined by the following rules:

$$\frac{}{\tau \sim \tau}\ \text{(C-Refl)} \qquad \frac{}{? \sim \tau}\ \text{(C-UnkL)} \qquad \frac{}{\tau \sim ?}\ \text{(C-UnkR)} \qquad \frac{\tau_{\mathsf{in}} \sim \tau'_{\mathsf{in}} \qquad \tau_{\mathsf{out}} \sim \tau'_{\mathsf{out}}}{\tau_{\mathsf{in}} \to \tau_{\mathsf{out}} \sim \tau'_{\mathsf{in}} \to \tau'_{\mathsf{out}}}\ \text{(C-Arrow)}$$

$$\frac{\tau_\ell \sim \tau'_\ell \qquad \tau_r \sim \tau'_r}{\tau_\ell \times \tau_r \sim \tau'_\ell \times \tau'_r}\ \text{(C-Prod)} \qquad \frac{\tau_\ell \sim \tau'_\ell \qquad \tau_r \sim \tau'_r}{\tau_\ell + \tau_r \sim \tau'_\ell + \tau'_r}\ \text{(C-Sum)}$$

The (C-UnkL) and (C-UnkR) rules establish that the type hole is consistent with every type. Like type equivalence, type consistency is reflexive (by the C-Refl rule). In other words, equal types are consistent. Consistency is also symmetric (it is straightforward to prove that these rules obey symmetry by induction). Unlike type equivalence, however, **type consistency is not transitive**. If it were transitive, we could make any two types consistent with each other by proceeding transitively through the type hole. That would make it impossible to ensure that known types are respected. For example, if type consistency were transitive then by subsumption, we could successfully analyze a number literal against the Bool type!

We can, however, analyze a number literal against the unknown type using just these rules:

$$\frac{\dfrac{}{\vdash \underline{3} \Rightarrow \mathsf{Num}}\ \text{(S-NumLiteral)} \qquad \dfrac{}{\mathsf{Num} \sim ?}\ \text{(C-UnkR)}}{\vdash \underline{3} \Leftarrow ?}\ \text{(A-Subsumption)}$$

Functions For plain ALFA, the type analysis rule for unannotated functions requires that the expected type be an arrow type:

$$\frac{\Gamma, x : \tau_{\mathsf{in}} \vdash e_{\mathsf{body}} \Leftarrow \tau_{\mathsf{out}}}{\Gamma \vdash \mathsf{fun}\ x \to e_{\mathsf{body}} \Leftarrow \tau_{\mathsf{in}} \to \tau_{\mathsf{out}}}$$

In Gradual ALFA, this rule is still valid. However, there is one additional situation that we need to consider: when the expected type is unknown, rather than known to be an arrow type. We cannot rely on subsumption to handle this situation because unannotated functions do not synthesize a type. One approach in this situation is to add a second rule for exactly this situation, which gives the bound variable unknown type and then checks the function body against unknown type:

$$\frac{\Gamma, x : ? \vdash e_{\mathsf{body}} \Leftarrow ?}{\Gamma \vdash \mathsf{fun}\ x \to e_{\mathsf{body}} \Leftarrow ?}$$

However, having two different rules for type analysis of unannotated functions is inelegant. We can avoid this by noticing that type analysis against the unknown type operates just as if we had instead analyzed it using the above rule from ALFA but with the expected type $? \to ?$ rather than ?. Let us call this type the "matched arrow type" for the unknown type. Let us also define the matched arrow type for arrow types as an identity relation, i.e. we simply say arrow types are matched to themselves. We can encode this with a simple judgement, $\tau \blacktriangleright_{\to} \tau_{\mathsf{in}} \to \tau_{\mathsf{out}}$, which can be read "$\tau$ has matched arrow type $\tau_{\mathsf{in}} \to \tau_{\mathsf{out}}$":

$$\frac{}{? \blacktriangleright_{\to} ? \to ?}\ \text{(MA-Hole)} \qquad \frac{}{\tau_\ell \to \tau_r \blacktriangleright_{\to} \tau_\ell \to \tau_r}\ \text{(MA-Arrow)}$$

Using this judgement, we can combine both of the rules above into a single rule:

$$\frac{\tau \blacktriangleright_{\rightarrow} \tau_{\mathsf{in}} \rightarrow \tau_{\mathsf{out}} \qquad \Gamma, x : \tau_{\mathsf{in}} \vdash e_{\mathsf{body}} \Leftarrow \tau_{\mathsf{out}}}{\Gamma \vdash \mathsf{fun}\ x \rightarrow e_{\mathsf{body}} \Leftarrow \tau} \ \text{(A-Fun)}$$

When τ is already an arrow type, it matches to itself and so this rule behaves just like the (A-Fun) rule from ALFA. When τ is the unknown type, then it gets matched to $? \rightarrow ?$ and operates as just described above.

The same judgement also allows us define a single rule for analyzing half-annotated functions:

$$\frac{\tau \blacktriangleright_{\rightarrow} \tau_{\mathsf{in}} \rightarrow \tau_{\mathsf{out}} \qquad \tau_{\mathsf{in}}' \sim \tau_{\mathsf{in}} \qquad \Gamma, x : \tau_{\mathsf{in}}' \vdash e_{\mathsf{body}} \Leftarrow \tau_{\mathsf{out}}}{\Gamma \vdash \mathsf{fun}\ (x : \tau_{\mathsf{in}}') \rightarrow e_{\mathsf{body}} \Leftarrow \tau} \ \text{(A-FunAnn)}$$

Notice that the requirement that the type annotation be equal to the input type is weakened to allow it to be merely consistent.

Half-annotated functions can also synthesize a type:

$$\frac{\Gamma, x : \tau_{\mathsf{in}} \vdash e_{\mathsf{body}} \Rightarrow \tau_{\mathsf{out}}}{\Gamma \vdash \mathsf{fun}\ (x : \tau_{\mathsf{in}}) \rightarrow e_{\mathsf{body}} \Rightarrow \tau_{\mathsf{in}} \rightarrow \tau_{\mathsf{out}}} \ \text{(S-FunAnn)}$$

It turns out, this same trick allows us to avoid defining two separate rules for function application as well. The standard function application rule from ALFA, written bidirectionally, is:

$$\frac{\Gamma \vdash e_{\mathsf{fun}} \Rightarrow \tau_{\mathsf{in}} \rightarrow \tau_{\mathsf{out}} \qquad \Gamma \vdash e_{\mathsf{arg}} \Leftarrow \tau_{\mathsf{in}}}{\Gamma \vdash e_{\mathsf{fun}}(e_{\mathsf{arg}}) \Rightarrow \tau_{\mathsf{out}}}$$

This rule works fine when e_{fun} synthesizes arrow type, but what about when it synthesizes unknown type? Naïvely, we would need a second rule that operates as if e_{fun} had synthesized the type $? \rightarrow ?$. However, we can combine these two rules into the following rule for function application, which handles both situations:

$$\frac{\Gamma \vdash e_{\mathsf{fun}} \Rightarrow \tau \qquad \tau \blacktriangleright_{\rightarrow} \tau_{\mathsf{in}} \rightarrow \tau_{\mathsf{out}} \qquad \Gamma \vdash e_{\mathsf{arg}} \Leftarrow \tau_{\mathsf{in}}}{\Gamma \vdash e_{\mathsf{fun}}(e_{\mathsf{arg}}) \Rightarrow \tau_{\mathsf{out}}} \ \text{(S-Ap)}$$

Numeric Operators This rule also guides the rules for the various unary and binary operators, which can be thought of as "built-in functions". Just like function arguments are analyzed in the rule above, the operands of these operators are analyzed against the appropriate type. For example, negation of an expression e synthesizes type Num as long as e can be analyzed against type Num.

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{Num}}{\Gamma \vdash -e \Rightarrow \mathsf{Num}} \ \text{(S-Neg)}$$

This intuition guides our rules for the binary arithmetic expressions as well. They synthesize type Num as long as the operands can be analyzed against Num in the given context.

$$\frac{\Gamma \vdash e_\ell \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e_r \Leftarrow \mathsf{Num}}{\Gamma \vdash (e_\ell + e_r) \Rightarrow \mathsf{Num}} \ \text{(S-Plus)} \qquad \frac{\Gamma \vdash e_\ell \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e_r \Leftarrow \mathsf{Num}}{\Gamma \vdash (e_\ell - e_r) \Rightarrow \mathsf{Num}} \ \text{(S-Minus)}$$

$$\frac{\Gamma \vdash e_\ell \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e_r \Leftarrow \mathsf{Num}}{\Gamma \vdash (e_\ell * e_r) \Rightarrow \mathsf{Num}} \ \text{(S-Times)}$$

The rules for the numeric comparison expressions are again guided by this intuition.

$$\frac{\Gamma \vdash e_\ell \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e_r \Leftarrow \mathsf{Num}}{\Gamma \vdash (e_\ell < e_r) \Rightarrow \mathsf{Bool}} \text{ (S-Lt)} \qquad \frac{\Gamma \vdash e_\ell \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e_r \Leftarrow \mathsf{Num}}{\Gamma \vdash (e_\ell > e_r) \Rightarrow \mathsf{Bool}} \text{ (S-Gt)}$$

$$\frac{\Gamma \vdash e_\ell \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e_r \Leftarrow \mathsf{Num}}{\Gamma \vdash (e_\ell =?\, e_r) \Rightarrow \mathsf{Bool}} \text{ (S-Eq)}$$

Another way to think about it is that in the conclusions of rules, we should use synthesis whenever possible (so that types can be synthesized, i.e. locally inferred, in as many situations as possible), but in the premises of rules we should use analysis whenever possible (so that the type information from parent expressions propagates down into the children for, perhaps, eventual use in situations where synthesis is not possible and analysis is the only option).

Let Expressions In an annotated let expression, $\mathsf{let}\ x : \tau_1\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2$, the type annotation, τ_1, allows us to analyze e_1. We need both synthesis and analysis rules for the let expression as a whole, because its type is the same as the type of e_2, which may or may not have a unique type.

$$\frac{\Gamma \vdash e_{\text{def}} \Leftarrow \tau_{\text{def}} \qquad \Gamma, x : \tau_{\text{def}} \vdash e_{\text{body}} \Rightarrow \tau_{\text{body}}}{\Gamma \vdash (\mathsf{let}\ x : \tau_{\text{def}}\ \mathsf{be}\ e_{\text{def}}\ \mathsf{in}\ e_{\text{body}}) \Rightarrow \tau_{\text{body}}} \text{ (S-LetAnn)}$$

$$\frac{\Gamma \vdash e_{\text{def}} \Leftarrow \tau_{\text{def}} \qquad \Gamma, x : \tau_{\text{def}} \vdash e_{\text{body}} \Leftarrow \tau}{\Gamma \vdash (\mathsf{let}\ x : \tau_{\text{def}}\ \mathsf{be}\ e_{\text{def}}\ \mathsf{in}\ e_{\text{body}}) \Leftarrow \tau} \text{ (A-LetAnn)}$$

Unannotated let expressions are similar to annotated let expressions, except e_1 must be able synthesize a type.

$$\frac{\Gamma \vdash e_{\text{def}} \Rightarrow \tau_{\text{def}} \qquad \Gamma, x : \tau_{\text{def}} \vdash e_{\text{body}} \Rightarrow \tau}{\Gamma \vdash (\mathsf{let}\ x\ \mathsf{be}\ e_{\text{def}}\ \mathsf{in}\ e_{\text{body}}) \Rightarrow \tau} \text{ (S-Let)}$$

$$\frac{\Gamma \vdash e_{\text{def}} \Rightarrow \tau_{\text{def}} \qquad \Gamma, x : \tau_{\text{def}} \vdash e_{\text{body}} \Leftarrow \tau}{\Gamma \vdash (\mathsf{let}\ x\ \mathsf{be}\ e_{\text{def}}\ \mathsf{in}\ e_{\text{body}}) \Leftarrow \tau} \text{ (A-Let)}$$

Fixpoints Unannotated fixpoints are, like unannotated functions, not able to synthesize a type, but they can be analyzed against a given type. Notice how the type of the self reference, x, is the same as the type of the fixpoint as a whole.

$$\frac{\Gamma, x : \tau \vdash e_{\text{body}} \Leftarrow \tau}{\Gamma \vdash (\mathsf{fix}\ x \to e_{\text{body}}) \Leftarrow \tau} \text{ (A-Fix)}$$

Annotated fixpoints can synthesize a type because the type is provided.

$$\frac{\Gamma, x : \tau \vdash e_{\text{body}} \Leftarrow \tau}{\Gamma \vdash (\mathsf{fix}\ (x : \tau) \to e_{\text{body}}) \Rightarrow \tau} \text{ (S-FixAnn)}$$

Products The trivial value synthesizes unit type. We can synthesize a binary product type for a pair as long as we can synthesize a type for both elements of the pair.

$$\frac{}{\Gamma \vdash () \Rightarrow \mathsf{1}} \text{ (S-Triv)} \qquad \frac{\Gamma \vdash e_\ell \Rightarrow \tau_\ell \qquad \Gamma \vdash e_r \Rightarrow \tau_r}{\Gamma \vdash (e_\ell, e_r) \Rightarrow \tau_\ell \times \tau_r} \text{ (S-Pair)}$$

We might also consider the following analysis rules to propagate types into a pair:

$$\frac{\Gamma \vdash e_\ell \Leftarrow \tau_\ell \qquad \Gamma \vdash e_r \Leftarrow \tau_r}{\Gamma \vdash (e_\ell, e_r) \Leftarrow \tau_\ell \times \tau_r}$$

However, here again we need to consider a new situation in Gradual ALFA: when the expected type is unknown, rather than a product type. And just as with functions, we can handle both situations using a single rule by defining a simple auxiliary judgement, $\tau \blacktriangleright_\times \tau_\ell \times \tau_r$, which can be read "$\tau$ has matched product type $\tau_\ell \times \tau_r$":

$$\frac{}{? \blacktriangleright_\times ? \times ?}\ \text{(MP-Hole)} \qquad \frac{}{\tau_\ell \times \tau_r \blacktriangleright_\times \tau_\ell \times \tau_r}\ \text{(MP-Prod)}$$

Using this judgement, we can define the analysis rule for pairs to handle both situations:

$$\frac{\tau \blacktriangleright_\times \tau_\ell \times \tau_r \qquad \Gamma \vdash e_\ell \Leftarrow \tau_\ell \qquad \Gamma \vdash e_r \Leftarrow \tau_r}{\Gamma \vdash (e_\ell, e_r) \Leftarrow \tau}\ \text{(A-Pair)}$$

A similar issue comes up in the elimination rules for pairs. The let-pair rules need to handle the situation where the bound expression synthesizes a product type as well as the situation where it synthesizes the unknown type. We can combine them into the following rules:

$$\frac{\Gamma \vdash e_{\mathsf{def}} \Rightarrow \tau_{\mathsf{def}} \qquad \tau_{\mathsf{def}} \blacktriangleright_\times \tau_\ell \times \tau_r \qquad \Gamma, x : \tau_\ell, y : \tau_r \vdash e_{\mathsf{body}} \Rightarrow \tau}{\Gamma \vdash \mathsf{let}\ (x, y)\ \mathsf{be}\ e_{\mathsf{def}}\ \mathsf{in}\ e_{\mathsf{body}} \Rightarrow \tau}\ \text{(S-LetPair)}$$

$$\frac{\Gamma \vdash e_{\mathsf{def}} \Rightarrow \tau_{\mathsf{def}} \qquad \tau_{\mathsf{def}} \blacktriangleright_\times \tau_\ell \times \tau_r \qquad \Gamma, x : \tau_\ell, y : \tau_r \vdash e_{\mathsf{body}} \Leftarrow \tau}{\Gamma \vdash \mathsf{let}\ (x, y)\ \mathsf{be}\ e_{\mathsf{def}}\ \mathsf{in}\ e_{\mathsf{body}} \Leftarrow \tau}\ \text{(A-LetPair)}$$

Similarly, the projection rules can be written using type matching as follows:

$$\frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau \blacktriangleright_\times \tau_\ell \times \tau_r}{\Gamma \vdash e.\mathsf{fst} \Rightarrow \tau_\ell}\ \text{(S-PrjL)} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau \blacktriangleright_\times \tau_\ell \times \tau_r}{\Gamma \vdash e.\mathsf{snd} \Rightarrow \tau_r}\ \text{(S-PrjR)}$$

Sums The rules for working with sum types similarly need to account for unknown types. Again, we can simplify matters by defining a simple auxiliary judgement, $\tau \blacktriangleright_+ \tau_\ell + \tau_r$, which can be read "$\tau$ has matched sum type $\tau_\ell + \tau_r$":

$$\frac{}{? \blacktriangleright_+ ? + ?}\ \text{(MS-Hole)} \qquad \frac{}{\tau_\ell + \tau_r \blacktriangleright_+ \tau_\ell + \tau_r}\ \text{(MS-Sum)}$$

Using this judgement, we can define the analysis rule for injections to handle both the situation where the expected type is a sum type and where it is unknown.

$$\frac{\tau \blacktriangleright_+ \tau_\ell + \tau_r \qquad \Gamma \vdash e \Leftarrow \tau_\ell}{\Gamma \vdash \mathsf{L}\ e \Leftarrow \tau}\ \text{(A-InjL)} \qquad \frac{\tau \blacktriangleright_+ \tau_\ell + \tau_r \qquad \Gamma \vdash e \Leftarrow \tau_r}{\Gamma \vdash \mathsf{R}\ e \Leftarrow \tau}\ \text{(A-InjR)}$$

Similarly, the type analysis rule for case expressions can handle the situation when the scrutinee synthesizes a sum type or an unknown type using type matching:

$$\frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau' \blacktriangleright_+ \tau_\ell + \tau_r \qquad \Gamma, x : \tau_\ell \vdash e_\ell \Leftarrow \tau \qquad \Gamma, y : \tau_r \vdash e_r \Leftarrow \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r \Leftarrow \tau}\ \text{(A-Case)}$$

The **if** analysis rule is more straightforward because the scrutinee must be of `Bool` type:

$$\frac{\Gamma \vdash e_1 \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e_2 \Leftarrow \tau \qquad \Gamma \vdash e_3 \Leftarrow \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Leftarrow \tau}\ \text{(A-If)}$$

In a non-gradual setting, we could define a type synthesis rule for case expressions and if expressions, which required that both branches synthesize equal types:

$$\frac{\Gamma \vdash e_{\mathsf{cond}} \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e_{\mathsf{then}} \Rightarrow \tau \qquad \Gamma \vdash e_{\mathsf{else}} \Rightarrow \tau}{\Gamma \vdash \mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau_\ell + \tau_r \qquad \Gamma, x : \tau_\ell \vdash e_\ell \Rightarrow \tau \qquad \Gamma, y : \tau_r \vdash e_r \Rightarrow \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r \Rightarrow \tau}$$

In Gradual ALFA, the situation is more complicated because the two branches might synthesize types that are consistent but not equal. For example, one branch might synthesize $\mathsf{Num} \to ?$ while the other synthesizes $? \to \mathsf{Num}$. Which of these types should we choose for the type of the if/case expression as a whole? It turns out there are multiple valid approaches.

The most permissive approach is to synthesize the **greatest lower bound** of the two types, i.e. the type that joins the two types by keeping all the structure that is common and placing a type hole wherever the two types are consistent but not equal. For example, the greatest lower bound of $\mathsf{Num} \to ?$ and $? \to \mathsf{Num}$ is $? \to ?$. This expresses the fact that we know that, no matter which branch we take, we will get something of arrow type, but we cannot be certain about either the input or the output type.

We can define the greatest lower bound of two consistent types as follows:

$$\begin{array}{rcl}
\mathsf{glb}(\tau, \tau) & = & \tau \\
\mathsf{glb}(?, \tau) & = & ? \\
\mathsf{glb}(\tau, ?) & = & ? \\
\mathsf{glb}(\tau_{\mathsf{in}} \to \tau_{\mathsf{out}}, \tau'_{\mathsf{in}} \to \tau'_{\mathsf{out}}) & = & \mathsf{glb}(\tau_{\mathsf{in}}, \tau'_{\mathsf{in}}) \to \mathsf{glb}(\tau_{\mathsf{out}}, \tau'_{\mathsf{out}}) \\
\mathsf{glb}(\tau_\ell \times \tau_r, \tau'_\ell \times \tau'_r) & = & \mathsf{glb}(\tau_\ell, \tau'_\ell) \times \mathsf{glb}(\tau_r, \tau'_r) \\
\mathsf{glb}(\tau_\ell + \tau_r, \tau'_\ell + \tau'_r) & = & \mathsf{glb}(\tau_\ell, \tau'_\ell) + \mathsf{glb}(\tau_r, \tau'_r)
\end{array}$$

Notice that the greatest lower bound for inconsistent types is not defined. This is because we do not want to accept if/case expressions where the two branches are known to be inconsistent—we allow differences only if there are unknown types involved.

This gives rise to the following synthesis rules (notice the use of glb in the conclusion):

$$\frac{\Gamma \vdash e_{\mathsf{cond}} \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e_{\mathsf{then}} \Rightarrow \tau_{\mathsf{then}} \qquad \Gamma \vdash e_{\mathsf{else}} \Rightarrow \tau_{\mathsf{else}}}{\Gamma \vdash \mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \Rightarrow \mathsf{glb}(\tau_{\mathsf{then}}, \tau_{\mathsf{else}})}\ \text{(S-If)}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau \blacktriangleright_+ \tau_\ell + \tau_r \qquad \Gamma, x : \tau_\ell \vdash e_\ell \Rightarrow \tau'_\ell \qquad \Gamma, y : \tau_r \vdash e_r \Rightarrow \tau'_r}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r \Rightarrow \mathsf{glb}(\tau'_\ell, \tau'_r)}\ \text{(S-Case)}$$

Consider the following example, assuming $x : \mathsf{Bool}$ and $f : \mathsf{Num} \to ?$:

$$\mathsf{if}\ x\ \mathsf{then}\ f\ \underline{5}\ \mathsf{else}\ \underline{6}$$

The first branch has unknown type while the second branch has type Num. The greatest lower bound of these types is ?, so that is the type that synthesized for this if expression. At run-time, the result in either case will be tagged because its type was not statically known. This will maintain safety in the case where f returns a non-number.

7.2.1 Expression Holes

Type holes allow us to leave types unknown. Some languages, including Haskell and Hazel, have started to experiment with allowing expressions to be left unknown as well. *Expression holes* serve as placeholders for unknown expressions. This is similar to the common practice of raising an exception like `NotImplemented`. However, it is more concise and it can be given special treatment by tools like Hazel (see hazel.org for more on that).

What type does an expression hole have? Well, the unknown type, of course! Here is the rule:

$$\frac{}{\Gamma \vdash \_ \Rightarrow ?}\ \text{(S-Hole)}$$

By subsumption, an expression hole can appear anywhere in a program. For example, if we have a function $f : \mathsf{Num} \to \mathsf{Num}$, we can apply it to an expression hole, f _. This expression has type Num.

7.2.2 Problems

Exercise 3: Derivation (Written, 20%) Consider the following judgement.

$$\Gamma_{\mathsf{fg}} \vdash \mathsf{fun}\ (a : ?) \to \mathsf{case}\ f\ \_\ \mathsf{of}\ \mathsf{L}(x) \to a\ (g\ x)\ \mathsf{else}\ \mathsf{R}(y) \to a\ y + \underline{12} \Rightarrow \tau$$

where

$$\Gamma_{\mathsf{fg}} = f : \mathsf{Bool} \to ?,\ g : \mathsf{Num} \to \mathsf{Num}$$

Part 1. Determine the type τ and provide a derivation of the above judgement for that choice. You can use Γ_{fg} and the following abbreviations in your derivation:

$$\begin{aligned}\Gamma_{\mathsf{fga}} &= \Gamma_{\mathsf{fg}},\ a : ?\\ \Gamma_{\mathsf{fgax}} &= \Gamma_{\mathsf{fg}},\ a : ?,\ x : ?\\ \Gamma_{\mathsf{fgay}} &= \Gamma_{\mathsf{fg}},\ a : ?,\ y : ?\end{aligned}$$

You may also define sub-derivation abbreviations, named using a D with a subscript of your choice, as necessary to fit the derivation clearly on the page (like we do in solutions to prior assignments).

Do not include applications of the metafunction glb directly in your derivation. Instead, compute the necessary greatest lower bound and include it inline in your derivation.

HINT: You will need to apply subsumption multiple times in your derivation. Do not forget that subsumption has two premises!

Part 2. Is there some way to fill in the type hole that appears in the function argument annotation in the expression above ($\mathsf{fun}(a : ?)\ \to ...$) to make it well-typed in A4's ALFA (rather than Gradual ALFA)? If so, provide a type. If not, explain why not (you may assume that the other holes will be filled appropriately if you decide it is possible).

Exercise 4: Local Type Inference (Learn-OCaml, 50%) Implement a local type inference system for Gradual ALFA in OCaml. (See the A2 handout for how to load Learn OCaml).

Type Consistency First, implement a function `consistent : Typ.t -> Typ.t -> bool` that checks whether the two input types are consistent.

Theorem 2 (Consistency Correctness). *$\tau_1 \sim \tau_2$ iff* `consistent` $\lfloor\tau_1\rfloor$ $\lfloor\tau_2\rfloor \equiv$ `true`

Greatest Lower Bound Next, implement a function `glb : Typ.t -> Typ.t -> Typ.t option` that returns greatest lower bound of the two input types, if one exists.

Theorem 3 (GLB Correctness). $\mathsf{glb}(\tau_1, \tau_2) = \tau$ *iff* $\texttt{glb}\lfloor\tau_1\rfloor\ \lfloor\tau_2\rfloor \equiv \texttt{Some}\ \lfloor\tau\rfloor$

Type Synthesis and Analysis Finally, finish implementing the provided type synthesis and analysis functions (which are mutually defined, as in A4 and A5).

Theorem 4 (Local Type Inference Correctness). *Both of the following hold:*

1. $\Gamma \vdash e \Rightarrow \tau$ *iff* $\texttt{syn}\ \lfloor\Gamma\rfloor\ \lfloor e\rfloor \equiv \texttt{Some}\ \lfloor\tau\rfloor$.
2. $\Gamma \vdash e \Leftarrow \tau$ *iff* $\texttt{ana}\ \lfloor\Gamma\rfloor\ \lfloor e\rfloor\ \lfloor\tau\rfloor \equiv \texttt{true}$.

You should use the provided functions `Typ.matched_arrow`, `Typ.matched_prod`, and `Typ.matched_sum` in your implementation. These behave as follows.

Lemma 5 (Matched Type Correctness). *All of the following hold:*

1. $\tau \blacktriangleright_{\rightarrow} \tau_{in} \rightarrow \tau_{out}$ *iff* $\texttt{Typ.matched\_arrow}\ \lfloor\tau\rfloor \equiv \texttt{Some}\ (\lfloor\tau_{in}\rfloor,\ \lfloor\tau_{out}\rfloor)$
2. $\tau \blacktriangleright_{\times} \tau_\ell \times \tau_r$ *iff* $\texttt{Typ.matched\_prod}\ \lfloor\tau\rfloor \equiv \texttt{Some}\ (\lfloor\tau_\ell\rfloor,\ \lfloor\tau_r\rfloor)$
3. $\tau \blacktriangleright_{+} \tau_\ell + \tau_r$ *iff* $\texttt{Typ.matched\_sum}\ \lfloor\tau\rfloor \equiv \texttt{Some}\ (\lfloor\tau_\ell\rfloor,\ \lfloor\tau_r\rfloor)$

A Example Derivation from Section 1

The example mentioned in Section 1 has the following derivation. Try it for yourself and check that you did it correctly.

To make the proof more concise, we define the following abbreviation:

$$\Gamma_1 = A \supset \bot,\ B \supset \bot,\ A \vee B$$

in:

$$\dfrac{\dfrac{}{\Gamma_1 \vdash A \vee B}\ \text{(assumption)} \qquad \mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma_1 \vdash \bot}\ (\vee\text{-E})$$

where $\mathcal{D}_1$ is:

$$\dfrac{\dfrac{}{\Gamma_1, A \vdash A \supset \bot}\ \text{(assumption)} \qquad \dfrac{}{\Gamma_1, A \vdash A}\ \text{(assumption)}}{\Gamma_1, A \vdash \bot}\ (\supset\text{-E})$$

and $\mathcal{D}_2$ is:

$$\dfrac{\dfrac{}{\Gamma_1, B \vdash B \supset \bot}\ \text{(assumption)} \qquad \dfrac{}{\Gamma_1, B \vdash B}\ \text{(assumption)}}{\Gamma_1, B \vdash \bot}\ (\supset\text{-E})$$

(Notice that $A \supset \bot$ can be written as a not, i.e. $\neg A$, per the end of Sec. 1, and similarly for B, so we're just proving that the set of hypotheses (A or B) and (not A) and (not B), are absurd, so we can prove false.)

(As a further exercise, try writing the corresponding proof expression using the rules from Sec. 4!)