CSCI-GA-2110: Programming Languages, Fall 2024

PSet 5 - Written

Aditeya Baral
N19186654

1 Typechecking Let without Annotations

Task 1.1 Explain why it is easy to type check `let` without the annotation but more challenging to type check `lambda` without the annotation.

Type inference is straightforward for `let` bindings because

- The bound expression's type can **always be inferred** first.
- The inferred type can then be used to check the body of the `let` expression.
- No explicit type annotation is required for type checking.

Thus, in a `let` binding, the **body of the binding is already typed, so we can infer the variable type** based on its definition. This means we can determine the variable's type **without explicit annotation**.

However, for `lambda` expressions,

- The **argument's type is unknown** before type checking.
- Since `lambda` is **polymorphic**, the argument's type could vary based on the type of data used.
- Without annotations, there is **no initial type information** provided.
- The type of the `lambda`'s body may depend on the type of the argument provided.

Thus, the type annotation in `lambda` is an essential tool for type inference, providing the **initial type information** that allows the type checker to systematically infer and **check the types of the function's body and overall expression**. Without prior context to determine the input parameter's type, it becomes **challenging to check consistency** within the `lambda` body, determine polymorphic behaviour or ensure the type safety of a function.

Hence, **annotation for `lambda` serves as a crucial type constraint** that guides type checking, whereas **`let` can derive type information from its definition context**. This fundamental difference makes `lambda` type-checking more complex without annotations.

2 Polymorphism and Boxes

Task 2.1 What are the types of `f1` and `f2`? Why does `(begin (f1 #t) (f1 1))` cause a type error, but `(begin (f2 #t) (f2 1))` does not?

The type of `f1` is `('_a -> void)` where,

- `f1` creates a `box` containing an empty `list` **outside** the `lambda`, so it's **typed with a single, fixed type**.
- Since the `box` is created outside the `lambda`, it is **shared across all calls**.
- The `lambda` then takes any type `'a` and adds it to the boxed `list`. The lambda function modifies the boxed value by updating it to be a new `list` with `'a` added to the front of the current unboxed `list`.
- This **fixes the type of the `list` to the first type used**, thus setting a type constraint that all elements in the box must be of the **first** type.

Thus, `(begin (f1 #t) (f1 1))` throws a type error because the **first function call `(f1 #t)` fixes the type of the boxed `list` to `(listof boolean)`**. The second call `(f1 1)` attempts to add an **integer** to a `(listof boolean)`, which is a **type mismatch**. Since the `box` is shared across all calls to `f1`, the type checker enforces that all elements stored in the `box` must have the same type. This causes a type error when you mix types (e.g., `boolean` and `integer`).

The type of `f2` is `('a -> void)` where,

- `f2` creates a new `box` inside the `lambda` containing an empty `list` **for each function call**.
- Since a **new** `box` is created for each call, it allows for **polymorphic behaviour**.
- Each call can accept **any type `'a`, independent of previous calls**, thus different types of elements to be added. Each time the lambda function is called, it creates a new `box` initialized to `empty` and updates it locally by prepending `'a`.

Thus, `(begin (f2 #t) (f2 1))` does not cause a type error because each call to `f2` creates a new `box` initialised to a new empty `list`, **each allowing different types of elements**. When the first function call `(f2 #t)` is made, it creates and modifies a `(listof boolean)`. The `box` is **updated locally within that call** and updates are not shared with other calls. The next function call `(f2 1)` creates and modifies a **separate** `(listof integer)`. A new `box` is created and **updated locally** with integers, **independent** of the `box` created during the previous call. Since the two calls are **independent and don't share the same** `list`, there is no type error thrown.