Software Verification Project Report

Farhad Azimzade 4788206

Introduction

This topic of this project is formalising and proving certain properties of a type inferencer for a small domain-specific functional language. The main parts involve the typing rules, the type checker, and the corresponding soundness and completeness rules for the type system. The language contains constructs like anonymous functions, recursive functions, natural numbers, booleans, and with the extension, the product type.

Code Overview

The code for this project is spread across several files, one of which is the base language definition and the others refer to individual exercises:

- **base_language.lean** → core language definitions (i.e. *ty*, *exp*, and *ctx*) and the context lookup function.
- **typing_rules.lean** → inductive typing that models the typing judgements of the object language.
- **exercise_2.lean**
- **exercise_3.lean**
- **type_inferencer.lean** → function producing the type of an expression in the object language.
- **exercise_6.lean**
- **completeness_.lean** → theorems regarding completeness of the typing rules and the type checker.
- **soundness_.lean** → theorems regarding soundness of the typing rules and the type checker.

Note: Since exercise 5 is not a coding exercise, it only appears in the report.

Exercises

3.1 Exercise 1: Typing Judgements

Exercise 1 requires representing the typing judgment $\Gamma \vdash e : A$ as an inductively defined proposition. The idea is to create constructors for the inductive type `typed` that would produce proofs of a typing judgement for each expression constructor in the language.

3.2 Exercise 2

In proving lemmas from Exercise 2, a few insights could be extracted. Firstly, the `cc` tactic can be used to prove inequalities in the goal. For example, in the proof if $\texttt{lemma}_3$, at one stage, the goal is $\vdash \neg (\text{"}f\text{"} = \text{"}x\text{"})$. Here, the two string literals are unequal, and the `cc` tactic allows proving this fact.

Secondly, because the lemmas rely on defined objects, such as the context, object language types, and strings for the variables, those need to be either defined as constants at the top of the file or added to the lemmas as arguments. Given the style suggestion in the project description, these identifiers need to be introduced at the beginning outside the lemmas. However, lemmas that rely on constants become non-computable and are required to be annotated by a *noncomputable* keyword. Presumably, since constants do not have explicit values, any function dependent on them cannot be evaluated, only type checked.

Instead, it seems better to provide the identifiers alongside explicit definitions, such as `def x` : *string* := "x". This way, the identifiers used in the lemmas have explicity definitions, which drops Lean's requirement for the non-computability property to be annotated.

Another reason for defining identifiers with immediate string literals is that equality and inequality of strings can be easily invoked on there identifiers. For example, a goal with an if-statement over two string identifiers, x and y, like:

$$\vdash \texttt{ite}\ (x = y)\ (...)\ (...)$$

can be unfolded and then computed using the `unfold` and `rw` tactics:

```
...
unfold x, -- Γ ⊢ ite ("x" = y) (branch₁) (branch₂)
unfold y, -- Γ ⊢ ite ("x" = "y") (branch) (branch₂)
rw if_neg, -- Γ ⊢ branch₂
...
```

Listing 1: Unfolding to string literals.

where each

3.3 Exercise 3

Using the syntax of the object language, a function that compares two natural numbers for equality can be defined as in the pseudocode in Listing 2.

```
def eq_nat : exp :=
    λ (x : ℕ),
        λ (y : ℕ),
            if (x == 0)
            then
                if (y == 0)
                then "true"
                else "false"
             else
                if (y == 0)
                then "false"
                else eq_nat (pred x) (pred y)
```

Listing 2: Function for comparing natural numbers for equality.

3.4 Exercise 4: Type Inference

3.4.1 `let` Bindings vs. Anonymous Lambda Applications

Initially, I defined all cases of expressions for the type checker using the more readable style with `let` bindings for my intermediate variables. For example, the initial definition for the case `ELam x` A `e` is provided in Listing 3.

```
| Γ (exp.ELam x A e) := let Γ' : ctx := (ctx.ctx_snoc Γ x A) in
                        let output_type : option ty := (type_infer Γ' e) in
                        bind (output_type) (λ o, some (ty.TFun A o))
```

Listing 3: Initial type inference definition for `ELam x`

However, as I proceeded to prove test lemmas in Exercise 6, I noticed that Lean did not automatically unfold the definitions within the variables introduced by `let`. This could be due to the property of `let` bindings not being subject to reduction rules, as per the book. This is contrast to λ-expressions, which are indeed reduced automatically. In order to utilise this property, I converted my definitions from `let` bindings to anonymous functions (Listing 4).

```
| Γ (exp.ELam x A e) :=
                      (
                          λ Γ' : ctx,
                              (
                                  (
                                      λ output_type : option ty,
                                          bind (output_type) (λ o, some (ty.TFun A o))
                                  )
                                  (type_infer Γ' e)
                              )
                      )
                      (ctx.ctx_snoc Γ x A)
```

Listing 4: Updated type inference definition for `ELam`.

Having assumed that all of my initial definitions would need to be rewritten using raw lambdas, I decided to use a more monadic style for the definitions. For instance, the case for `EIf`

in the type checker is in Listing 5.

```
| Γ (exp.EIf e1 e2 e3) :=
   bind (type_infer Γ e1)
   (λ cond_type,
       if (cond_type = ty.TBool)
       then (
           bind (type_infer Γ e2)
           (λ if_branch_type,
               bind (type_infer Γ e3)
               (λ else_branch_type,
                   if (if_branch_type = else_branch_type)
                   then some if_branch_type
                   else none
               )
           )
       )
       else none
```

Listing 5: Updated type inference definition for `EIf`.

Ultimately, however, the choice of approach to definitions did not matter for the proofs about the type checker. While I could not unfold the definitions further for the definitions, Lean managed to perform all the necessary reductions in all 3 different implementations when using the reflexive tactic `exact rfl`. Via the specified conversion rules, definitions using `let`s, raw lambdas, and monadic binding could all be reduced and the appropriate equality inferred.

3.4.2 Equality of Types *ty*

One issue in the implementation of type inference is the comparison of object language types *ty*. For example, when type checking a function application `EApp e1 e2` (6), the type of the argument `e2` : T has to be equal to the input type of the function `e1` : $TFun\ A \to B$. This means that within the type checker, a comparison $A = T$ has to be performed.

```
| Γ (exp.EApp e1 e2) := let input_type : option ty := (type_infer Γ e2) in
                        let output_type : option ty := (type_infer Γ e1) in
                        match output_type with
                        | (some (ty.TFun A B)) :=
                           match input_type with
                           | (some T) := if (A = T) then (some B) else none
                           | _ := none
                           end
                        | _ := none
                        end
```

Listing 6: Type inference of `EApp`.

In order to use the equality proposition $p : A = T$, where $A, T : ty$, as the condition of an if-expression in Lean, this proposition needs to be shown to be decidable. Presumably, once a proposition is shown to be decidable, it can act as a boolean, since it can then only take on values $\{true, false\}$. Otherwise, an undecidable proposition can also be a non-terminating

computation.

Decidability of equality over the object language types *ty* can be shown by introducing an instance of *decidable_eq ty*. The instance is defined in Listing 7.

```
instance decidable_eq_ty : decidable_eq ty :=
λ (A B : ty),
   match A, B with
   | ty.TNat, ty.TNat := is_true rfl
   | ty.TBool, ty.TBool := is_true rfl
   | (ty.TFun A B), (ty.TFun C D) := sorry
   | (ty.TProd A B), (ty.TProd C D) := sorry
   | _, _ := is_false sorry
   end
```

Listing 7: Instance of decidability of equlity of *ty*.

However, I had difficulty in completing the definition in 7 for function and product types and in ruling out all other combinations. Instead, as I later found out, a trivial instance of decidable equality over a type can be derived automatically by Lean. In order to achieve this, the main definition of the inductive type *ty* should be adjusted by annotating it with a `derive` tag (8).

```
@[derive decidable_eq] inductive ty : Type
| TNat : ty
| TBool : ty
| TFun : ty → ty → ty
| TProd : ty → ty → ty
```

Listing 8: Instance of decidability of equlity of *ty* using `derive` tag.

3.5 Exercise 5

- ℚ: Explain the difference between the typing judgment typed and the type inferencer `type_infer`.

  𝔸: Typing judgement `typed` Γ e A is a syntactic statement about the type system, whereas the type checker provides the semantics of the type system. The two are very closely related.

  According to Pierce [1], "the typing rules tell us that terms of certain forms are well typed under certain conditions, but by looking at an individual typing rule, we can never conclude that some term is not well typed, since it is always possible that another rule could be used to type this term."

  On the other hand, the type checker allows to determine the type of a given syntactic form of the language (i.e. expression `e` : *exp*). In particular, the type checker can also inform us whether an expression is ill-typed.

- ℚ: Explain why the type annotations A and B in lambda abstractions $\lambda(x : A).e$ and recursive functions `rec` f $(x : A) : B := e$ are needed for the implementation of your

type inferencer. Explain if they are also needed to describe the typing rules.

𝔸: Type annotations for the input and output types of anonymous functions need to be known when inferring the types of these functions, because the input/output types have to be added to the current context in order to evaluate the bodies of the functions. For example, in an expression $\texttt{f} := \lambda \texttt{x}, \texttt{x} + 1$, the type of the parameter `x` has to be determined in order to type check the overall function as $\texttt{f} : Nat \to Nat$.

However, it can still be possible to determine the function type without the type annotation $\texttt{x} : Nat$. This requires more sophisticated mechanics of type inference that would resolve constraints that the operations within the function body impose on the parameter `x`.

Similarly, for the recursive functions, the overall type of the function of the function needs to be known prior to type checking the body, since the body may use the function name as a variable in its context.

3.6 Exercise 7: Soundness and Completeness

3.6.1 Soundness

Soundness refers to the correctness of all type checking definitions. That is, obtaining a certain type for a given expression in the object language implies that the corresponding typing judgement can be derived using the typing rules. So, the hypothesis in the proof is the type evaluation $\texttt{type\_infer}\ \Gamma\ \texttt{e} = A \to \Gamma \vdash \texttt{e} : A$ of an expression $\texttt{e} : exp$.

For readability, soundness proof is constructed using two sub-proofs, each of which tackles one context case. The proof `sound_in_empty_ctx`, as the name suggest proves that the type system is sound when evaluated in an empty context. Proof `sound_in_non_empty_ctx` does so in a non-empty context. Furthermore, the general structure of the proofs follows a style of pattern matching, whereby once the proof was split by `cases` into two cases for its two constructors, each was further split into cases based on the expression `e`.

The key insight in soundness proof appears to be the usage of the `unfold...at...` tactic, which preforms one step of the defined computation at the level of hypothesis. This way, insights into the typing judgement could be derived from the hypothesis that the expression in question has been evaluated to type $A : ty$.

One such common pattern in the soundness proof is unfolding the hypothesis and allowing `cc` tactic to derive the desire conclusion from the contradiction within the untrue hypothesis. For example, the proof of soundness for the empty context is in Listing 9.

```
lemma sound_in_empty_ctx (e : exp) (A : ty) : type_infer (ctx.ctx_nil) e =
    option.some A → typed (ctx.ctx_nil) e A :=
 λ e_inf : type_infer (ctx.ctx_nil) e = option.some A,
   begin
     cases e,
     { ... }
     ...
     { -- ETrue
       cases A,
       { -- A = TNat
         unfold type_infer at e_inf,
         cc
       },
       { -- A = TBool
         apply typed.True_typed
       },
       { -- A = TFun _ _
         unfold type_infer at e_inf,
         cc
       },
       { -- A = TProd _ _
         unfold type_infer at e_inf,
         cc
       }
     }
     ...
     { ... }
```

Listing 9: Snippet of the soundness proof in an empty context showing the case for `ETrue`.

3.7 Extension: Product Type

The extension I attempted adding is the tuple construction, which includes:

- new product type $A \times B$
- pairing function `pair` : $A \to B \to A \times B$
- left projection `fst` : $A \times B \to A$
- right projection `fst` : $A \times B \to B$

One of the desired properties of the pairing function and the projections is partial applicability. This means, that individual expressions `pair`, `fst`, and `snd` should be valid standalone expressions in the object language. This way, they can be passed around as arguments and curried.

To achieve this, these three constructions could be defined similarly to the successor function `ESucc` : *exp*. Encoding them this way, means that the typing judgements over the pair

functions can be:

$$\frac{}{\Gamma \vdash \texttt{Pair} : \mathit{Fun}\ A\ (\mathit{Fun}\ B\ (\mathit{Prod}\ A\ B))}$$

$$\frac{}{\Gamma \vdash \texttt{Fst} : \mathit{Fun}\ ((\mathit{Prod}\ A\ B))\ A}$$

$$\frac{}{\Gamma \vdash \texttt{Snd} : \mathit{Fun}\ ((\mathit{Prod}\ A\ B))\ B}$$

However, as I proceeded to implement type inferencer for these new constructs, I had an issue with inferring the types of the subexpressions, since those are not available in the current definition. Unlike expressions that deal with natural numbers exclusively, such as $\texttt{ESucc} : \mathit{Nat} \to \mathit{Nat}$, the overall type of the `Pair` function depends on the types of the arguments that will be provided. This means that the type of `Pair` cannot be inferred directly using the current definition.

In order to have access to the types of the subexpressions (arguments), I have redefined the pairing functions to require arguments to be applied immediately, like so:

$$\begin{aligned}
&\texttt{Pair} : \mathit{exp} \to \mathit{exp} \to \mathit{exp} \\
&\texttt{Fst} : \mathit{exp} \to \mathit{exp} \\
&\texttt{Snd} : \mathit{exp} \to \mathit{exp}
\end{aligned}$$

This means that the pairing functions are no longer standalone expressions in the language. For example, the following judgement involving partial application of `Pair` can no longer be derived:

$$\frac{}{\Gamma \vdash \lambda(\texttt{x} : A),\ \texttt{Pair x}\ :\ \mathit{Fun}\ A\ (\mathit{Fun}\ B\ (\mathit{Prod}\ A\ B))}$$

However, an equivalent expression can still be defined, albeit in a somewhat more cumbersome way, meaning that there is no apparent negative effect on the expressivity of the object language. In this case, an equivalent that does not require curried `Pair` could be:

$$\frac{}{\Gamma \vdash \lambda(\texttt{x} : A),\ \lambda(\texttt{y} : B),\ \texttt{Pair x y}\ :\ \mathit{Fun}\ A\ (\mathit{Fun}\ B\ (\mathit{Prod}\ A\ B))}$$

Defining pairing functions in such a limiting way is unfortunate. I could not find a workaround that would enable both curried pairing functions and allow for type checking them. It seems that the base object language may need to be modified further to accommodate that, perhaps by adding a construction for a generic type $\mathit{Type} : \mathit{ty}$ to act as a placeholder inside $\mathit{Prod} : \mathit{ty} \to \mathit{ty} \to \mathit{ty}$.

References

[1] Benjamin C. Pierce. Types and programming languages. 2002.