Exercise 1: Transition Systems and Types

Transition Systems

1. Recall the specification of the compiler for Hutton’s Razor in http://www.cl.cam.ac.uk/~mbg28/oope- ex2.pdf we looked at last year. In Section 2 of the linked PDF, you will see that the behaviour of the instructions of the abstract machine is only described using two bullet points:

- PUSH v pushes the value v on top of the stack
- ADD takes two values off the stack, adds them up, and then pushes the result on top of the stack

These descriptions are informal and could be interpreted incorrectly. However, conveniently we have just learned how to do better!

(a) Let a stack be an element s of $\mathbb{N}^*$ where $\varepsilon$ is the empty stack and sequences of natural numbers represent non-empty stacks. Suggest how configurations should be represented in a transition system for the abstract machine described in last year’s exercise. (1 mark)

(b) What is included in the set of values for this transition system? Discuss how this compares with the set of values for L1 from the notes. (3 marks)

(c) Give a formal, operational semantics for the abstract machine. (4 marks)

(d) Do there exist configurations which are stuck? If so, give an example. If not, prove it. (2 marks)

2. Suppose that we add two new instructions $i = \ldots | \text{LOAD } \ell | \text{STORE } \ell$

where LOAD $\ell$ loads a value from a memory location labelled $\ell$ before pushing it on top of the stack and STORE $\ell$ pops a value off the stack and stores it in the memory location labelled $\ell$ afterwards. Labels are elements of a set L and memory locations store elements of $\mathbb{N}$.

(a) Choose an appropriate representation for stores and show how the representation of configurations needs to be changed to accommodate for them. (2 marks)

(b) Give a new operational semantics for the abstract machine. (4 marks)

(c) Show a complete derivation from an empty stack and empty store for the program

$$\text{PUSH } 4; \text{PUSH } 8; \text{STORE } \ell; \text{PUSH } 15; \text{ADD}; \text{LOAD } \ell; \text{ADD}; \ldots$$

(6 marks)

Types

3. A type may be seen as an approximation of the value of the term it is assigned to. For example, if an arbitrary expression $e$ has type $\text{int}$, then we know that after evaluating $e$ we will end up with a value of type $\text{int}$. We also know that if e.g. a function $f$ expects an argument of type $\text{bool}$, then $e$ is not a suitable argument for it, thus allowing us to catch such errors at compile-time. However, manually annotating expressions with types is cumbersome, so we would like a program to do it for us. Suppose that a programming language whose syntax contains type annotations is stripped
of them, then type inference (or type reconstruction) describes the process of assigning suitable types to the untyped terms of the language to reconstruct the original, annotated terms.

For this exercise, let us first extend Hutton’s Razor with boolean values (elements of \( B \)) and conditional expressions:

\[
e = e + e \mid n \in \mathbb{N} \mid b \in B \mid \text{if } e \text{ then } e \text{ else } e
\]

We define the semantics of this language denotationally. That is, instead of using inductively-defined transitions between states of an abstract machine, we define a total, recursive function which maps expressions to some set of values:

\[
\llbracket e \rrbracket : e \rightarrow (\mathbb{N} + B)\perp
\]

\[
\llbracket n \rrbracket = \text{inl } n
\]

\[
\llbracket b \rrbracket = \text{inr } b
\]

\[
\llbracket e_0 + e_1 \rrbracket = \begin{cases} 
\text{inl } (x + y) & \text{if } \llbracket e_0 \rrbracket = \text{inl } x \text{ and } \llbracket e_1 \rrbracket = \text{inl } y \\
\perp & \text{otherwise}
\end{cases}
\]

\[
\llbracket \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rrbracket = \begin{cases} 
\llbracket e_1 \rrbracket & \text{if } \llbracket e_0 \rrbracket = \text{inr } \text{true} \\
\llbracket e_2 \rrbracket & \text{if } \llbracket e_0 \rrbracket = \text{inr } \text{false} \\
\perp & \text{otherwise}
\end{cases}
\]

Note that the codomain of \( \llbracket \cdot \rrbracket \) (i.e. the set of values) is \( (\mathbb{N} + B)\perp \). \( \mathbb{N} + B \) represents a tagged union. Given two sets, \( A \) and \( B \), the tagged union of the two is defined as:

\[
A + B = \{ \text{inl } a \mid a \in A \} \cup \{ \text{inr } b \mid b \in B \}
\]

In other words, a tagged union is simply a union where the elements are tagged with either \( \text{inl} \) or \( \text{inr} \) to indicate which set they originate from.

The subscript \( \perp \) indicates that the set is “lifted”. This simply means that we include \( \perp \) (read as bottom or undefined) in the set. Therefore, we have:

\[
(\mathbb{N} + B)\perp = \{ \perp, \text{inl } 0, \text{inl } 1, \ldots, \text{inr } \text{true}, \text{inr } \text{false} \}
\]

To summarise, the co-domain of the evaluation function is the set consisting of \( \perp \), all natural numbers tagged with \( \text{inl} \), and all boolean values tagged with \( \text{inr} \). We use \( \perp \) in the definition of the evaluation function to indicate failure: i.e. things we cannot evaluate. Therefore \( \perp \) means that the evaluation got stuck.

According to the grammar for \( e \), an example of a valid expression in this language would be the following:

\[
\text{if } 4 \text{ then } \text{false} \text{ else } 8
\]

This expression does intuitively not make much sense. If we apply the evaluation function, it will evaluate to \( \perp \). It gets stuck. In practice, it is not feasible to run a program to find out if it will get stuck or not. To catch such errors at compile-time, we will add a type system to our language. For this, we need to define suitable types. A good start would be one type for each kind of value in our language:

\[
\tau = \text{nat} | \text{bool}
\]

We know that a well-formed expression will either evaluate to a natural number or to a boolean value. Since types are approximations of the values of expressions, our definition of \( \tau \) seems adequate.
(a) Suggest a set of type inference rules to form a deduction system for a typed version of Hutton’s Razor with boolean and conditional expressions. *Hint:* both of the branches of a conditional expression should have the same type. (5 marks)

(b) Prove using your inference rules that `if 4 then false else 8` is not well-typed. (3 marks)

(c) Turn your deduction system into an algorithm and implement it in a language of your choice. Discuss the difference between a deduction system and a type inference algorithm. (6 marks)

(d) Translate the denotational semantics shown above into an operational semantics. Discuss how your answer compares with the denotational semantics. (6 marks)

(e) *Type checking* is a related problem to type inference. It asks: given an expression `e` and a type `τ`, is `τ` a suitable type for `e`? Show how type checking can be implemented by reducing it to type inference. Explain your answer and any assumptions you make. Think of other inferred languages you know (e.g. ML or Haskell), could you do the same there? (6 marks)

(f) Suppose that we want to allow each branch of a conditional expression to return a different type. Discuss how this could be implemented in the type system. Justify your decisions and describe all changes you would make to the types and the typing rules. (10 marks)