COMP3161/COMP9164

Properties and Datatypes Exercises

Liam O'Connor

November 1, 2019

1. **Safety and Liveness Properties**

   (a) [$\star$] For each of the following properties, identify if it is a safety or a liveness property.

   i. When I come home, there must be beer in the fridge.

   > **Solution:** Safety (violated by the finite steps where I come home and there is no beer in the fridge.)

   ii. When I come home, I'll drop onto the couch and drink a beer.

   > **Solution:** Liveness (violated only after infinite time, where I come home and never drop on to the couch or drink a beer)

   iii. I'll be home later.

   > **Solution:** Liveness (for an unbounded definition of "later")

   iv. When process p has executed line 5, then process q must execute line 17 again.

   > **Solution:** Liveness

   v. When process p has executed line 5, then process q cannot execute line 17 again.

   > **Solution:** Safety

   vi. Process q cannot execute line 17 again unless process p has executed line 5.

   > **Solution:** Safety

   vii. Process p has to execute line 5 before q can execute line 17 again.

   > **Solution:** Liveness

   (b) [$\star\star\star\star$] By considering a property as a set of behaviours (infinite sequences of states), show that if the state space Σ has at least two states, then any property can be expressed as the intersection of two liveness properties.

   Hint: It may be helpful to know that the union of a liveness property and any other property is also a liveness property (this result follows from the fact that liveness properties are dense sets).

Solution: As the state space has at least two states, we can assume there exists a state $\mathsf{a} \in \Sigma$ and a different state $\mathsf{b} \in \Sigma$.

Then, we can construct two liveness properties, M and N:

$$M = \{p\mathsf{a}^\omega \mid p \in \Sigma^\star\}$$
$$N = \{p\mathsf{b}^\omega \mid p \in \Sigma^\star\}$$

Here $\Sigma^\star$ refers to the set of finite sequences of states. Stated in English, the property M says that "the program will eventually loop forever (or terminate) in state a", and the property N says that "the program will eventually loop forever (or terminate) in state b". Before ending up in that final state, the program is free to do any finite sequence of actions.

These two properties are *liveness* properties as we cannot refute them merely by observing a finite prefix of the behaviour. Also, they are *disjoint*, that is, $M \cap N = \emptyset$, because there is no behaviour that can both terminate in state a and in state b, as they are different states.

Recall that the union of a liveness property and any other property is also a liveness property. This means that for some arbitrary property P, the properties $P \cup M$ and $P \cup N$ are both liveness properties. Therefore, to show that any property P is the intersection of two liveness properties, it suffices to show that:

$$(P \cup M) \cap (P \cup N) = P$$

We do this with set theory below:

$$\begin{array}{rcll}
(P \cup M) \cap (P \cup N) & = & ((P \cup M) \cap P) \cup ((P \cup M) \cap N) & \text{distrib. of } \cap \text{ over } \cup \\
& = & P \cup ((P \cup M) \cap N) & \cap \text{ absorption} \\
& = & P \cup ((P \cap N) \cup (M \cap N)) & \text{distrib. of } \cap \text{ over } \cup \\
& = & P \cup ((P \cap N) \cup \emptyset) & \text{disjointness of } M, N \\
& = & P \cup (P \cap N) & \cup \text{ identity} \\
& = & P & \cup \text{ absorption}
\end{array}$$

2. **Type Safety**: Consider this very simple language with function application and two built-in functions:

$$\begin{array}{rcl}
e & ::= & (\mathtt{App}\ e_1\ e_2) \\
& \mid & \mathsf{S} \\
& \mid & \mathsf{K}
\end{array}$$

The dynamic semantics evaluate the left hand side of applications as much as possible:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$$

The K function takes two arguments and returns the first one.

$$\frac{}{(\mathtt{App}\ (\mathtt{App}\ \mathsf{K}\ x)\ y) \mapsto x}$$

The S function takes three arguments, applies the first argument to the third, and applies the result of that to the second argument applied to the third. More clearly:

$$\frac{}{(\mathtt{App}\ (\mathtt{App}\ (\mathtt{App}\ \mathsf{S}\ x)\ y)\ z) \mapsto (\mathtt{App}\ (\mathtt{App}\ x\ z)\ (\mathtt{App}\ y\ z))}$$

(a) [⋆⋆] Define a set of typing rules for this language, where the set of types is described by:

$$\begin{array}{rcl} \tau & ::= & \tau_1 \rightarrow \tau_2 \\ & \mid & \iota \end{array}$$

Note that $\rightarrow$ is right-associative, so $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ means $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

Solution:

$$\frac{e_1 : \tau_1 \rightarrow \tau_2 \quad e_2 : \tau_1}{e_1\ e_2 : \tau_2}$$

$$\frac{}{\mathsf{K} : \tau_1 \rightarrow \tau_2 \rightarrow \tau_1}$$

$$\frac{}{\mathsf{S} : (\tau_1 \rightarrow \tau_2 \rightarrow \tau_3) \rightarrow (\tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_3}$$

(b) [⋆⋆⋆] In order to prove that your typing rules are type-safe, we must prove *progress* and *preservation*. For progress, we will define the set of final states as all states that have no successor:

$$F = \{s \mid \nexists s'.\ s \mapsto s'\}$$

This trivially satisfies progress, as progress states that all well-typed states either have a successor state or are final states.

Preservation, however, requires a nontrivial proof. Prove preservation for your typing rules with respect to the dynamic semantics of this language.

Assignment Project Exam Help

Solution: We must show that, assuming $e : \tau$ and $e \mapsto e'$, that $e' : \tau$. We will proceed by rule induction on $e \mapsto e'$.

https://powcoder.com

Base case. When $e = (\texttt{App}\ (\texttt{App}\ (\texttt{App}\ \mathsf{S}\ x)\ y)\ z)$ and $e' = (\texttt{App}\ (\texttt{App}\ x\ z)\ (\texttt{App}\ y\ z))$, from the rule for S.

We know from the fact that $e : \tau$ that there exists types τ_1 and τ_2 such that:

Add WeChat powcoder

- $x : \tau_1 \rightarrow \tau_2 \rightarrow \tau$
- $y : \tau_1 \rightarrow \tau_2$
- $z : \tau_1$

Then we can show that $e' : \tau$:

$$\dfrac{\dfrac{\dfrac{}{x : \tau_1 \rightarrow \tau_2 \rightarrow \tau} \quad \dfrac{}{z : \tau_1}}{(\texttt{App}\ x\ z) : \tau_2 \rightarrow \tau} \quad \dfrac{\dfrac{}{y : \tau_1 \rightarrow \tau_2} \quad \dfrac{}{z : \tau_1}}{(\texttt{App}\ y\ z) : \tau_2}}{(\texttt{App}\ (\texttt{App}\ x\ z)\ (\texttt{App}\ y\ z)) : \tau}$$

Base case. When $e = (\texttt{App}\ (\texttt{App}\ \mathsf{K}\ x)\ y)$ and $e' = x$, from the rule for K. We know from $e : \tau$ that there exists a type τ_1 such that:

- $x : \tau$
- $y : \tau_1$

Seeing as $e' = x$, we know that $e' : \tau$ already.

Inductive case. When $e = (\texttt{App}\ e_1\ e_2)$, and $e_1 \mapsto e_1'$, and $e' = (\texttt{App}\ e_1'\ e_2)$. We get that the induction hypothesis (from $e_1 \mapsto e_1'$) that, for any type τ, if $e_1 : \tau$ then $e_1' : \tau$.

We know from $e : \tau$ that there exists a type τ_1 such that:

- $e_1 : \tau_1 \to \tau$
- $e_2 : \tau_1$

Seeing as e_1 has type $\tau_1 \to \tau$, we know from our inductive hypothesis that $e_1' : \tau_1 \to \tau$. Therefore $(\texttt{App}\ e_1\ e_2) : \tau$ from the application typing rule. $\square$

3. **Haskell Types**: Determine a MinHS type that is isomorphic to the following Haskell type declarations:

   (a) [$\star$] `data MaybeInt = Just Int | Nothing`

   Solution: Solutions may vary, but $\texttt{Int} + \mathbf{1}$ is the simplest.

   (b) [$\star$] `data Nat = Zero | Suc Nat`

   Solution: $\texttt{rec}\, t.\ \mathbf{1} + t$

   (c) [$\star$] `data IntTree = Tree Int IntTree IntTree | Leaf Int`

Assignment Project Exam Help

   Solution: $\texttt{rec}\, t.\ (\texttt{Int} \times t \times t) + \texttt{Int}$

https://powcoder.com

4. **Inhabitation**: Do the following MinHS types contain any (finite) values? If not, explain why. If so, give an example value.

   (a) [$\star$] $\texttt{rec}\, t.\ \texttt{Int} + t$

Add WeChat powcoder

   Solution: Yes, $(\texttt{Roll}\ (\texttt{InR}\ (\texttt{Roll}\ (\texttt{InL}\ 3))))$ is an example finite value.

   (b) [$\star$] $\texttt{rec}\, t.\ \texttt{Int} \times t$

   Solution: No, the only way to express a value of this type is something like

   $$(\texttt{Recfun}\ f.x.\ (\texttt{Roll}\ (\texttt{Pair}\ 4\ x)))$$

   Which in a call-by-value (strict) semantics would be non-terminating, but acceptable in a non-strict (lazy) semantics.

   (c) [$\star$] $(\texttt{rec}\, t.\ \texttt{Int} \times t) + \texttt{Bool}$

   Solution: Yes, the only finite values are $(\texttt{InR}\ \texttt{True})$ and $(\texttt{InR}\ \texttt{False})$. All other values are infinite.

5. **Encodings**: For each of the following sets, give a MinHS type that corresponds to it. Justify why your MinHS type is equivalent to the set, for example by providing a bijective function that, given a element of that set, gives the corresponding MinHS value of the corresponding type.

   (a) [$\star$] The natural number set $\mathbb{N}$.

Solution: The representation of unary natural numbers seen in question 2 suffices here:

$$\texttt{rec}\, t.\, \mathbf{1} + t$$

The mapping is defined as:

$$g(x) = \begin{cases} (\texttt{Roll (InL ())}) & \text{if } x = 0 \\ (\texttt{Roll (InR } g(x-1))) & \text{if } x > 0 \end{cases}$$

(b) [$\star\star$] The set of integers $\mathbb{Z}$.

Solution: One of the simplest is $(\texttt{rec}\, t.\, \mathbf{1} + t) \times \texttt{Bool}$, i.e. a natural number combined with a sign bit. The mapping works as follows, where `False` represents negative numbers and `True` represents positive. Note that we offset the negative numbers by one so that there are not two zero values:

$$f(x) = \begin{cases} x < 0, & (\texttt{pair } g(-x-1), \texttt{False}) \\ x \geq 0, & (\texttt{pair } g(x), \texttt{True}) \end{cases}$$

(c) [$\star\star$] The set of rational numbers $\mathbb{Q}$.

Solution: Seeing as a rational number is just a pair of integers to represent the numerator and denominator respectively, we can use our integer type from before $(\mathbb{Z} = (\texttt{rec}\, t.\, \mathbf{1} + t) \times \texttt{Bool})$ and just use the type $\mathbb{Z} \times \mathbb{Z}$.

Technically there are more pairs of integers than there are rational numbers. We can simplify this by judging two values of this type to be equal even if they are not structurally identical. A pair (p_1, q_1) and a pair (p_2, q_2) are equal iff $p_1 q_2 = p_2 q_1$.

(d) [$\star\star\star$] The set of (computable) real numbers $\mathbb{R}_{\text{TM}}$. It may be useful to assume a lazy semantics.

Solution: A real number consists of an integer whole component and a possibly infinite sequence of fractional decimal digits.

For the integer component, it suffices to use our existing $\mathbb{Z}$ type.

Then, we just need an infinite sequence of digits, which we can define for binary digits with:

$$\texttt{rec}\, t.\, (\texttt{Bool} \times t)$$

Therefore, a computable real number is just $\mathbb{Z} \times (\texttt{rec}\, t.\, (\texttt{Bool} \times t))$.

6. **Curry-Howard**: Give a term in typed λ-calculus that is a proof of the following propositions. If there is no such term, explain why.

(a) [$\star$] $A \Rightarrow A \vee B$

Solution: The type required is $A \to A + B$.

InL

(b) [$\star$] $A \wedge B \Rightarrow A$

Solution: The type required is $A \times B \to A$.

$$\mathsf{fst}$$

(c) [⋆⋆] $P \vee P \Leftrightarrow P$
Hint: Recall that $A \Leftrightarrow B$ is shorthand for $A \Rightarrow B \wedge B \Rightarrow A$.

Solution: The type required is $(A + A \to A) \times (A \to A + A)$.

$$((\lambda s.\ \textbf{case}\ s\ \textbf{of}\ \mathsf{InL}\ x.\ x; \mathsf{InR}\ x.\ x), \mathsf{InL})$$

(d) [⋆⋆] $(A \wedge B \Rightarrow C) \Leftrightarrow (A \Rightarrow B \Rightarrow C)$

Solution: The type required is a product of:

$$x_1 : (A \times B \to C) \to (A \to B \to C)$$
$$x_1 = \lambda abc.\ \lambda a.\ \lambda b.\ abc\ (a, b)$$

And:

$$x_2 : (A \to B \to C) \to (A \times B \to C)$$
$$x_2 = \lambda abc.\ \lambda ab.\ abc\ (\mathsf{fst}\ ab)\ (\mathsf{snd}\ ab)$$

So the final answer is (x_1, x_2).

(e) [⋆⋆] $P \vee (Q \wedge R) \Rightarrow (P \vee Q) \wedge (P \vee R)$

Solution: The type required is $P + (Q \times R) \to (P + Q) \times (P + R)$.

$$\begin{aligned}&\lambda pOrQR.\ \textbf{case}\ pOrQR\ \textbf{of}\\ &\quad \mathsf{InL}\ p.\ (\mathsf{InL}\ p, \mathsf{InL}\ p);\\ &\quad \mathsf{InR}\ qr.\ (\mathsf{InR}\ (\mathsf{fst}\ qr), \mathsf{InR}\ (\mathsf{snd}\ qr))\end{aligned}$$

(f) [⋆⋆] $P \Rightarrow \neg(\neg P)$
Hint: Recall that $\neg A$ is shorthand for $A \Rightarrow \bot$.

Solution: The type required is $P \to (P \to \mathbf{0}) \to \mathbf{0}$, which we can implement with:

$$\lambda p.\ \lambda notP.\ notP\ p$$

(g) [⋆⋆⋆] $\neg(\neg P) \Rightarrow P$

Solution: This theorem does not hold constructively, so there is no term in standard typed lambda calculus.

(h) [⋆⋆⋆] $\neg(\neg(\neg P)) \Rightarrow \neg P$

Solution: The required type is $(((P \to \mathbf{0}) \to \mathbf{0}) \to \mathbf{0}) \to P \to \mathbf{0}$.
Recall our solution for part (d) was of type $P \to (P \to \mathbf{0}) \to \mathbf{0}$. Call this function d. Then we can implement this type with:

$$\lambda nnnp.\ \lambda p.\ nnnp\ (d\ p)$$

(i) [⋆⋆⋆] $(P \vee \neg P) \Rightarrow \neg(\neg P) \Rightarrow P$

Solution: The required type is $(P + (P \to \mathbf{0})) \to ((P \to \mathbf{0}) \to \mathbf{0}) \to P$

$$\begin{array}{l} \lambda pOrNotP.\ \lambda notNotP.\ \textbf{case}\ pOrNotP\ \textbf{of} \\ \quad \mathsf{InL}\ p.\ p; \\ \quad \mathsf{InR}\ notP.\ \mathsf{absurd}\ (notNotP\ notP) \end{array}$$