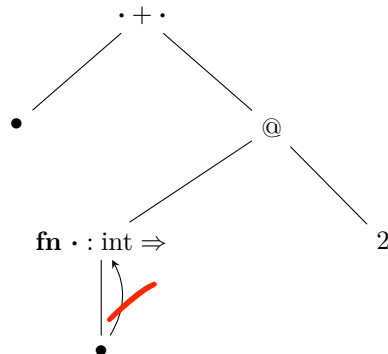


1 Notes Page 71

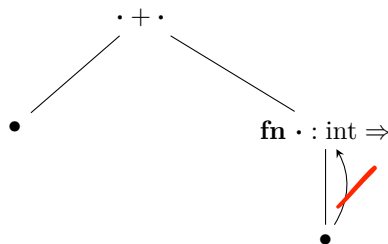
Exercise 18

What are the free variables of the following? Draw their abstract syntax trees up to alpha equivalence.

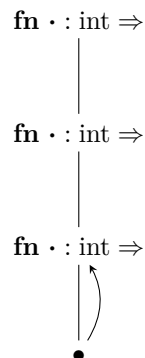
1. $\text{fv}(x + ((\mathbf{fn} \ y : \text{int} \Rightarrow z)2)) = \{x, z\}$



2. $\text{fv}(x + (\mathbf{fn} \ y : \text{int} \Rightarrow z)) = \{x, z\}$



3. $\text{fv}(\mathbf{fn} \ y : \text{int} \Rightarrow \mathbf{fn} \ y : \text{int} \Rightarrow \mathbf{fn} \ y : \text{int} \Rightarrow y) = \emptyset$



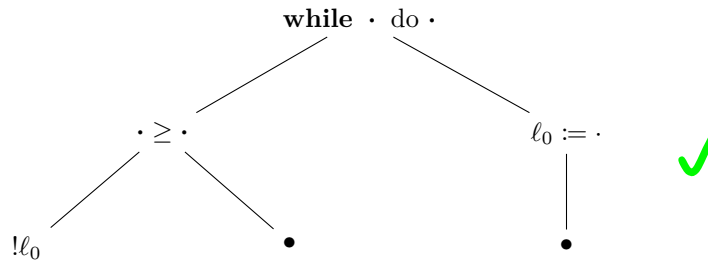
4. $\text{fv}(!\ell_0) = \emptyset$

$!\ell_0$



5. $\text{fv}(\mathbf{while} \ !\ell_0 \geq y \ \mathbf{do} \ \ell_0 := x) = \{x, y\}$





Exercise 19

What are the results of the following substitutions?

1. $\{\mathbf{fn} \ x : \text{int} \Rightarrow y/z\} \mathbf{fn} \ y : \text{int} \Rightarrow z \ y$

$$\mathbf{fn} \ z : \text{int} \Rightarrow (\mathbf{fn} \ x : \text{int} \Rightarrow y) \ z$$

2. $\{\mathbf{fn} \ x : \text{int} \Rightarrow x/x\} \mathbf{fn} \ y : \text{int} \Rightarrow x \ y$

$$\mathbf{fn} \ y : \text{int} \Rightarrow (\mathbf{fn} \ x : \text{int} \Rightarrow x) \ y$$

3. $\{\mathbf{fn} \ x : \text{int} \Rightarrow x/x\} \mathbf{fn} \ x : \text{int} \ x \ x$

$$\mathbf{fn} \ x : \text{int} \ x \ x$$

Exercise 21

Give a grammar for types, and typing rules for functions and application that allow only first order functions and prohibit partial applications.

Types:

$$\begin{aligned} A &::= \text{int} \mid \text{bool} \mid \text{unit} \\ T &::= A \mid A \rightarrow A \\ T_{loc} &::= \text{intref} \end{aligned}$$

Typing rules:

$$\overline{\Gamma \vdash n : \text{int}}$$

$$\overline{\Gamma \vdash b : \text{bool}} \quad \text{variables?}$$

$$\overline{\Gamma \vdash \mathbf{skip} : \text{unit}}$$

$$\frac{\Gamma \vdash e : A'}{\Gamma \vdash \mathbf{fn} \ x : A \Rightarrow e : A \rightarrow A'}$$

$$\frac{\Gamma \vdash \mathbf{fn} \ x : A \Rightarrow e : A \rightarrow A' \quad \Gamma \vdash e' : A}{\Gamma \vdash (\mathbf{fn} \ x : A \Rightarrow e) e' : A'}$$



Exercise 24

Prove Theorem 19 (Type Preservation) for recursive functions

Continue the proof from last supervision with $\Phi(e, s, e', s', T)$ defined the same:

$$\Phi(e, s, e', s', T) \triangleq (\Gamma \vdash e : T \wedge \langle e, s \rangle \rightarrow \langle e', s' \rangle) \implies \Gamma \vdash e' : T$$

Case (letrecfn) Recall the evaluation rule:

$$\begin{aligned} & \langle \text{let val rec } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}, s \rangle \\ & \quad \rightarrow \\ & \langle \{(\text{fn } y : T_1 \Rightarrow \text{let val rec } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end})/x\}e_2, s \rangle \end{aligned}$$

Recall also the typing rule (let rec fn):

$$\frac{\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \quad \Gamma, x : T_1 \rightarrow T_2 \vdash e_2 : T}{\Gamma \vdash \text{let val rec } x : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} : T}$$

There is no other typing rule with a conclusion of the form **let val rec ... end**. Therefore (let rec fn) must have been the last typing rule used and so we can infer the premises of the rule $\Gamma, x : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2$ and $\Gamma, x : T_1 \rightarrow T_2 \vdash e_2 : T$.

Since the reduced expression is e_2 with an expression of type $T_1 \rightarrow T_2$ for x and the premise states that under Γ and the assumption that $x : T_1 \rightarrow T_2$, $e_2 : T$; we can conclude that $\{...\}/x\}e_2$ has type T . **Uses the Substitution lemma**

Since the type of the reduced expression is the same as the type of the original expression, we can conclude $\Phi(e, s, e', s', T)$ as required. ✓

2 2015 Paper 6 Question 10

(b) Define a small-step right-to-left call-by-value operational semantics for this syntax.

$$\begin{aligned} \text{(reduce)} \quad & \frac{\langle e_1 \rangle \xrightarrow{L} \langle e'_1 \rangle}{\langle e_1 \ e_2 \rangle \xrightarrow{L} \langle e'_1 \ e_2 \rangle} & \text{(arg)} \quad & \frac{\langle e_2 \rangle \xrightarrow{L} \langle e'_2 \rangle}{\langle (\text{fn } x \Rightarrow e_1) \ e_2 \rangle \xrightarrow{L} \langle (\text{fn } x \Rightarrow e_1) \ e'_2 \rangle} \\ \text{(printarg)} \quad & \frac{\langle e \rangle \xrightarrow{L} \langle e' \rangle}{\langle \text{print } e \rangle \xrightarrow{L} \langle \text{print } e' \rangle} & \text{(call)} \quad & \frac{}{\langle (\text{fn } x \Rightarrow e) \ v \rangle \xrightarrow{\tau} \langle \{v/x\}e \rangle} \\ \text{(print)} \quad & \frac{}{\langle \text{print } n \rangle \xrightarrow{n} \langle \text{skip} \rangle} \end{aligned}$$

(d) We are normally interested in closed programs (with no free variables). Prove with respect to your call-by-value semantics of part (b) that if e is closed and $e \xrightarrow{L} e'$ then e' is closed. You can omit the cases for **print**.

An expression e is closed if and only if $\text{fv}(e) = \emptyset$. Therefore we must prove for all closed expressions e , the property Φ :

$$\Phi(e) \triangleq \forall e'. \left(\text{fv}(e) = \emptyset \wedge \langle e \rangle \xrightarrow{L} \langle e' \rangle \right) \implies (\text{fv}(e') = \emptyset)$$

This can be performed by structural induction on the evaluation rules using the induction hypothesis $\Phi(e)$ for all subexpressions e .



<https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2015p6q10.pdf>



(Case) (reduce) Recall the rule (reduce):

$$\frac{\langle e_1 \rangle \xrightarrow{L} \langle e'_1 \rangle}{\langle e_1 \ e_2 \rangle \xrightarrow{L} \langle e'_1 \ e_2 \rangle}$$

By assumption $\mathbf{fv}(e) = \emptyset$. Using this and the formula for the free variables of $e_1 \ e_2$ gives:

$$\begin{aligned} \mathbf{fv}(e_1 \ e_2) &= \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) \wedge \mathbf{fv}(e_1 \ e_2) = \emptyset \implies \\ \mathbf{fv}(e_1) &= \mathbf{fv}(e_2) = \emptyset \end{aligned}$$

$$\begin{aligned} \Phi(e_1) \wedge \mathbf{fv}(e_1) = \emptyset \wedge \langle e_1 \rangle \xrightarrow{L} \langle e'_1 \rangle &\implies \\ \mathbf{fv}(e'_1) &= \emptyset \implies \\ \mathbf{fv}(e'_1 \ e_2) &= \emptyset \implies \\ \Phi(e_1 \ e_2) & \end{aligned}$$



(Case) (arg) Recall the rule:

$$\frac{\langle e_2 \rangle \xrightarrow{L} \langle e'_2 \rangle}{\langle (\mathbf{fn} \ x \Rightarrow e_1) \ e_2 \rangle \xrightarrow{L} \langle (\mathbf{fn} \ x \rightarrow e_1) \ e'_2 \rangle}$$

By assumption, $\mathbf{fv}((\mathbf{fn} \ x \Rightarrow e_1) \ e_2) = \emptyset$.

$$\begin{aligned} \mathbf{fv}((\mathbf{fn} \ x \Rightarrow e_1) \ e_2) &= \emptyset \implies \\ \mathbf{fv}(\mathbf{fn} \ x \Rightarrow e_1) &= \emptyset \wedge \mathbf{fv}(e_2) = \emptyset \end{aligned}$$

Since e_2 is a subexpression of e , the induction hypothesis $\Phi(e_2)$ holds.

$$\begin{aligned} \Phi(e_2) \wedge \mathbf{fv}(e_2) = \emptyset \wedge \langle e_2 \rangle \xrightarrow{L} \langle e'_2 \rangle &\implies \\ \mathbf{fv}(e'_2) &= \emptyset \end{aligned}$$

$$\begin{aligned} \mathbf{fv}(\mathbf{fn} \ x \Rightarrow e_1) &= \emptyset \wedge \mathbf{fv}(e'_2) = \emptyset \implies \\ \mathbf{fv}((\mathbf{fn} \ x \Rightarrow e_1) \ e'_2) &= \emptyset \implies \\ \Phi((\mathbf{fn} \ x \Rightarrow e_1) \ e'_2) & \end{aligned}$$



(Case) (call) Recall the rule:

$$\frac{}{\langle (\mathbf{fn} \ x \Rightarrow e) \ v \rangle \xrightarrow{\tau} \langle \{v/x\}e \rangle}$$

By induction hypothesis:

$x \Rightarrow e$ is not an expression

$$\Phi(\mathbf{fn} \ x \Rightarrow e) \implies \mathbf{fv}(x \Rightarrow e) = \emptyset$$

The IH is that $((\mathbf{fn} \ x \Rightarrow e) \ v)$ has no free variables

$$\begin{aligned} \mathbf{fv}(\mathbf{fn} \ x \Rightarrow e) &= \mathbf{fv}(e) - \{x\} \wedge \mathbf{fv}(x \Rightarrow e) = \emptyset \implies \\ \mathbf{fv}(e) &\subseteq \{x\} \implies \\ \mathbf{fv}(\{v/x\}e) &\subseteq \{x\} - \{x\} \implies \\ \mathbf{fv}(\{v/x\}e) &= \emptyset \implies \\ \Phi((\mathbf{fn} \ x \Rightarrow e) \ v) & \end{aligned}$$

Since all cases have been proved, we can conclude that $\forall e. \Phi(e)$. As required.



3 2008 Paper 6 Question 11

Below is the syntax and type system for a simple functional language.

Integers $n \in \mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$

Variables $x \in \mathbb{X} = \{x, y, z\}$

Types $T ::= \text{int} \mid T \rightarrow T$

Type environments Γ , finite partial functions from variables to types.

Expressions $e ::= n \mid x \mid \text{fn } x : T \Rightarrow e \mid e_1 e_2$



<https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2008p6q11.pdf>

$$\begin{array}{ll} (\text{int}) & \Gamma \vdash n : \text{int} \text{ for } n \in \mathbb{Z} \\ (\text{fn}) & \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \text{fn } x : T \Rightarrow e : T \rightarrow T'} \end{array} \quad \begin{array}{ll} (\text{var}) & \Gamma \vdash x : T \text{ if } \Gamma(x) = T \\ (\text{app}) & \frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'} \end{array}$$

- (a) Give a call-by-value operational semantics for this language, defining a judgement $e \rightarrow e'$ (the language does not have store operations so you can take configurations to be just expressions).

$$(\text{call1}) \frac{\langle e \rangle \rightarrow \langle e' \rangle}{\langle (\text{fn } x \Rightarrow e'') e \rangle \rightarrow \langle (\text{fn } x \Rightarrow e'') e' \rangle}$$

$$(\text{call2}) \frac{}{\langle (\text{fn } x \Rightarrow e) n \rangle \rightarrow \langle \{n/x\}e \rangle}$$



- (b) Give an example of a stuck configuration.

$$\langle x \rangle$$



- (c) Prove the substitution lemma stated below:

If $\Gamma, x : T \vdash e' : T'$ and $\Gamma \vdash e : T$ with $x \notin \text{dom}(\Gamma)$ then $\Gamma \vdash \{e/x\}e' : T'$

Define $\Phi(e, e', x)$ as follows:

$$\Phi(e, e', x) \triangleq \forall \Gamma, x, T, T'. (\Gamma, x : T \vdash e' : T' \wedge \Gamma \vdash e : T \wedge x \notin \text{dom}(\Gamma)) \implies \Gamma \vdash \{e/x\}e' : T'$$

This can be proved by structural induction over the typing rules. By assumption, $\Gamma, x : T \vdash e' : T'$. Therefore, e' must be well typed and must have been derived by the typing rules. The induction hypothesis is $\Phi(e, e'', x)$ for all subexpressions e'' of e' .

Case (int) Recall the typing rule:

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int}}$$

If the last typing rule applied was (int), then e' must be of the form n . $\{e/x\}n = n$.

$$\begin{array}{l} \Gamma \vdash n : T \wedge \{e/x\}n = n \implies \\ \Gamma \vdash \{e/x\}n : T \text{ as required} \end{array}$$



Case (var) Recall the typing rule:

$$\frac{\Gamma(y) = T}{\Gamma \vdash y : T}$$



case $x \neq y$

If $x \neq y$ then $\{e/x\}y = y$

$$\begin{aligned} \Gamma \vdash y : T \wedge \{e/x\}y = y &\implies \\ \Gamma \vdash \{e/x\}y : T &\text{ as required} \end{aligned}$$



case $x = y$

If $x = y$ then $\{e/x\}y = e$

$$\begin{aligned} \Gamma \vdash x : T \wedge \{e/x\}y = e \wedge \Gamma \vdash e : T &\implies \\ \Gamma \vdash \{e/x\}y : T &\text{ as required} \end{aligned}$$



Case (fn) Recall the typing rule:

$$\frac{\Gamma, y : T \vdash e'' : T'}{\Gamma \vdash \mathbf{fn} \ y : T \Rightarrow e'' : T \rightarrow T'}$$

By the induction hypothesis $\Phi(e, e'', x)$ and the premise of the rule (fn):

$$\begin{aligned} \Gamma, y : T \vdash e'' : T' \wedge \Phi(e, e'', x) &\implies \\ \Gamma, y : T \vdash \{e/x\}e'' : T' & \end{aligned}$$

Using the rule (fn):

$$\frac{\Gamma, y : T \vdash \{e/x\}e'' : T'}{\Gamma \vdash \mathbf{fn} \ y : T \Rightarrow \{e/x\}e'' : T \rightarrow T'}$$

Since the premise of this is implied by the induction hypothesis, we can assume the conclusion.

Since y binds, x cannot occur in y . The required result is obtained.

$$\mathbf{fn} \ y \Rightarrow \{e/x\}e'' = \{e/x\}(\mathbf{fn} \ y \Rightarrow e'')$$

This works up to alpha equivalence (to guard against $x = y$)



Case (app) Recall the rule:

$$\frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \ e_2 : T'}$$

Using the induction hypothesis $\Phi(e, e_1, x)$ and $\Phi(e, e_2, x)$, we can infer $\Gamma \vdash \{e/x\}e_1 : T \rightarrow T'$ and $\Gamma \vdash \{e/x\}e_2 : T$. Substituting this into the rule above gives:

$$\frac{\Gamma \vdash \{e/x\}e_1 : T \rightarrow T' \quad \Gamma \vdash \{e/x\}e_2 : T}{\Gamma \vdash \{e/x\}e_1 \ \{e/x\}e_2 : T'}$$

The conclusion is the same as substituting e for x in the whole expression. Substituting this equality into the evaluation rule gives the required result:

$$\frac{\Gamma \vdash \{e/x\}e_1 : T \rightarrow T' \quad \Gamma \vdash \{e/x\}e_2 : T}{\Gamma \vdash \{e/x\}(\{e/x\}e_1 \ \{e/x\}e_2) : T'}$$

Therefore, the if the substitution principle holds for all subexpressions of e then it also holds for e . Since the substitution principle holds for all nullary expressions it must therefore hold for all expressions.




(d) State and prove type preservation, using the substitution lemma.

Type preservation Φ for an expression e is defined as:


$$\Phi(e) \triangleq \forall \Gamma, T, e'. \Gamma \vdash e : T \wedge \langle e \rangle \rightarrow \langle e' \rangle \Rightarrow \Gamma \vdash e' : T$$

To prove this, we can assume the LHS of the implication and prove the RHS. We assume that $\Gamma \vdash e : T$. So e is well-typed and so must conform to the typing rules. We can perform structural induction on the last typing rule used.


Case (int)

If the last typing rule used was (int), then $\exists n \in \mathbb{Z}. e = n$. Therefore there is no reduction from e . Hence the LHS of the implication does not hold and type preservation trivially holds in this case. 

Case (var)

If the last typing rule used was (var), then e must be a variable. Since there are no evaluation rules which can have a variable in the premise, e can therefore not reduce and so type preservation trivially holds in this case. 

Case (fn)

If the last typing rule used was (fn) then e must be of the form $\mathbf{fn} x \rightarrow e_1$. There is no evaluation rule with an expression of this form as the premise and therefore type preservation holds trivially as e cannot reduce. 

Case (app)

If the last typing rule used was (app) then e must be of the form $e_1 e_2$. Perform a case split on e_2 :


case $e_2 \in \mathbb{Z}$

In this case, we can apply the rule (call2): You don't decide what evaluation rule is used. You assume it steps and case split on what rule it was

$$\frac{}{\langle (\mathbf{fn} x \Rightarrow e) n \rangle \rightarrow \langle \{n/x\}e \rangle}$$

By use of the substitution principle:

$$\Gamma, x : T \vdash e' : T' \wedge \Gamma e : T \wedge x \notin \text{dom}(\Gamma) \Rightarrow \Gamma \vdash \{e/x\}e' : T'$$

Therefore the type of $\{n/x\}e$ must be T . Since the function took n as an argument, it must have type $\text{int} \rightarrow T$. Therefore we can apply the typing rule (app) and derive that the type of $e n$ must be T . Therefore the type of the expression was preserved in this reduction and so type preservation holds in this case. 

case $e_2 \notin \mathbb{Z}$

By assumption, $\Gamma \vdash e_2 : T$:

$$\begin{aligned} \Gamma \vdash e_2 : T \wedge e_2 \notin \mathbb{Z} \Rightarrow \\ \exists e'_2. \langle e_2 \rangle \rightarrow \langle e'_2 \rangle \end{aligned}$$

By induction hypothesis, type preservation holds on reductions for e_2 . Therefore the type of e'_2 must be the same as the type of e_2 – let this type be T . By use of the rule (app):

$$\frac{\Gamma \vdash e_2 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'} \quad \frac{\Gamma \vdash e_2 : T \rightarrow T' \quad \Gamma \vdash e'_2 : T}{\Gamma \vdash e_1 e'_2 : T'}$$



4 Exercises Page 85

Exercise 18

Labelled variant types are a generalisation of sum types, just as records are a generalisation of products. Design abstract syntax, type rules and evaluation rules for labelled variants, analogously to the way in which records generalise products.

Abstract Syntax:

$$\begin{aligned} lab &\in \mathbb{LAB} = \{p, q, \dots\} \\ T &::= \dots \mid T_1 + T_2 + \dots + T_n \\ e &::= \dots \mid lab_1[e : T_1] \mid lab_2[e : T_2] \mid \dots \mid lab_n(e : T_n) \end{aligned}$$

Typing Rules:

$$\begin{aligned} \text{(Variant)} \quad & \frac{\Gamma \vdash e : T_i}{\Gamma \vdash lab_i(e) : T_1 + T_2 + \dots + T_n} \\ \text{(Case)} \quad & \frac{\Gamma \vdash e : T_1 + T_2 + \dots + T_n \quad \Gamma \vdash x_i : T_i \vdash e_i : T}{\Gamma \vdash (\mathbf{case} \ e \ \mathbf{of} \ lab_1(x_1 : T_1) \Rightarrow e_1 : T \mid \dots \mid lab_n(x_n : T_n) \Rightarrow e_n : T) : T} \end{aligned}$$

Evaluation Rules:

$$\begin{aligned} \text{(lab)} \quad & \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle lab_i(e : T_i), s \rangle \rightarrow \langle lab_i(e' : T_i), s' \rangle} \\ \text{(case1)} \quad & \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \mathbf{case} \ e \ \mathbf{of} \ lab_1(x_1 : T_1) \Rightarrow e_1 \mid \dots \mid lab_n(x_n : T_n) \Rightarrow e_n, s \rangle \rightarrow \langle \mathbf{case} \ e' \ \mathbf{of} \ lab_1(x_1 : T_1) \Rightarrow e_1 \mid \dots \mid lab_n(x_n : T_n) \Rightarrow e_n, s' \rangle} \\ \text{(casematch)} \quad & \frac{}{\langle \mathbf{case} \ lab_i \ v \ \mathbf{of} \ lab_1(x_1 : T_1) \Rightarrow e_1 \mid \dots \mid lab_n(x_n : T_n) \Rightarrow e_n, s \rangle \rightarrow \langle \{v/x_i\}e_i, s \rangle} \end{aligned}$$

5 Notes Page 92

Exercise 32

For each of the two bogus T ref subtype rules on Slide 202, give an example program that is typeable with that rule but gets stuck at runtime.

This program will pass typechecking but get stuck at runtime under the first rule.

```
<
  ℓ := {p = 1}
  #q !ℓ,
  {ℓ ↦ {p = 0, q = 0}}
>
```



This program will pass typechecking but get stuck at runtime under the second rule:

$$\langle \begin{array}{l} \#q \text{ !}\ell, \\ \{\ell \mapsto \{p = 0\}\} \end{array} \rangle$$


Exercise 33

What should the subtype rules for sums $T + T'$ be?

$$(\text{sum1}) \quad \frac{T <: T''}{T + T' <: T'' + T'}$$

$$(\text{sum2}) \quad \frac{T' <: T''}{T + T' <: T + T''}$$



6 2014 Paper 6 Question 10

Consider the language L below, with call-by-value functions, ML-style references and types \mathbf{nat}_+ and \mathbf{real}_+ of positive natural and positive real numbers. L includes a primitive test for primality, $\mathbf{prime}(e)$ and a square root function, $\mathbf{sqrt}(e)$; these are defined only for positive-natural and positive-real values respectively.



<https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p6q10.pdf>

$$\begin{array}{l} T := \mathbf{bool} \mid \mathbf{nat}_+ \mid \mathbf{real}_+ \mid T \rightarrow T' \mid T \text{ ref} \\ e := x \mid n \mid r \mid \mathbf{fn } x : T \Rightarrow e \mid e \text{ e}' \mid \mathbf{ref } e \mid !e \mid e := e' \mid \mathbf{prime}(e) \mid \mathbf{sqrt}(e) \end{array}$$

Here x ranges over a set X of variables and n and r range over $\mathbb{N}_{>0}$ and $\mathbb{R}_{>0}$ respectively. Let Γ range over finite partial functions from X to types T .

- (a) Give typing rules defining $\Gamma \vdash e : T$ for $\mathbf{prime}(e)$ and $\mathbf{sqrt}(e)$.

$$(\text{prime}) \quad \frac{\Gamma \vdash e : \mathbf{nat}_+}{\Gamma \vdash \mathbf{prime}(e) : \mathbf{bool}}$$

$$(\text{sqrt}) \quad \frac{\Gamma \vdash e : \mathbf{real}_+}{\Gamma \vdash \mathbf{sqrt}(e) : \mathbf{real}_+}$$



There is an obvious runtime coercion from elements of \mathbf{nat}_+ to elements of \mathbf{real}_+ . To let programmers exploit that conveniently, we would like to define a type system from L that includes a subtype relation $T_1 <: T_2$ with $\mathbf{nat}_+ <: \mathbf{real}_+$. The type system should prevent all runtime errors.

- (i) Give the other rules defining $T_1 <: T_2$ and the subsumption rule to use that relation in $\Gamma \vdash e : T$.



$$\begin{array}{ll}
 \text{(reflexivity)} & \frac{}{T <: T} \\
 \text{(transitivity)} & \frac{T_1 <: T_2 \wedge T_2 <: T_3}{T_1 <: T_3} \\
 \text{(subsumption)} & \frac{\Gamma \vdash e : T \quad T <: T'}{\Gamma \vdash e : T'} \\
 \text{(function)} & \frac{T_1 <: T'_1 \wedge T_2 <: T'_2}{(T'_1 \rightarrow T_2) <: (T_1 \rightarrow T'_2)}
 \end{array}$$

(ii) Give the 6 (standard) typing rules defining $\Gamma \vdash e : T$ for functions and references.

$$\begin{array}{ll}
 \text{(fn)} & \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \mathbf{fn} \ x : T \Rightarrow e : T \rightarrow T'} \\
 \text{(apply)} & \frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \ e_2 : T'} \\
 \text{(ref)} & \frac{\Gamma \vdash e : T}{\Gamma \vdash \mathbf{ref} \ e : T \ \mathbf{ref}} \\
 \text{(assign)} & \frac{\Gamma \vdash e : T \ \mathbf{ref} \quad \Gamma \vdash e' : T}{\Gamma \vdash e := e' : T} \\
 \text{(deref)} & \frac{\Gamma \vdash e : T \ \mathbf{ref}}{\Gamma \vdash !e : T} \\
 \text{(loc)} & \frac{\Gamma(\ell) = T \ \mathbf{ref}}{\Gamma \vdash \ell : T \ \mathbf{ref}}
 \end{array}$$

(iii) With reference to your subtype rule for function types, explain covariance and contravariance of subtyping. Give examples in L showing that your rule is the only reasonable choice.

Functions have contravariant argument types and covariant return types. It must be this way to ensure that every function takes a type with at least as much information as it requires and returns types with at least as much information as the context is expecting.

Consider the following example code which passes typechecking but gets stuck at runtime when we have covariant argument types:

$$\frac{T_1 <: T'_1 \wedge T_2 <: T'_2}{(T_1 \rightarrow T_2) <: (T'_1 \rightarrow T'_2)}$$

$$\langle \mathbf{sqrt}((\mathbf{fn} \ x : \mathbf{real}_+ \rightarrow x) \ 0) \rangle \longrightarrow^* \langle \mathbf{sqrt}(0) \rangle \not\rightarrow$$

Consider the following example which passes typechecking but get stuck at runtime when we have contravariant return types:

$$\frac{T_1 <: T'_1 \wedge T_2 <: T'_2}{(T'_1 \rightarrow T'_2) <: (T_1 \rightarrow T_2)}$$

$$\langle \mathbf{prime}((\mathbf{fn} \ x : \mathbf{nat}_+ \rightarrow x) \ 3.14) \rangle \rightarrow \langle \mathbf{prime}(3.14) \rangle \not\rightarrow$$

(iv) Similarly, justify your rule for reference types.

There is no subtyping rule for reference types. If we have either covariant or contravariant reference types, then we must disallow either reading or writing to the reference to ensure type safety. This defeats the purpose of references, rendering them useless. **Illustrate the specific problem with writing / reading**

Consider the following example which passes typechecking with covariant references but gets stuck at runtime.

$$\frac{T <: T'}{T \ \mathbf{ref} <: T' \ \mathbf{ref}}$$

$$\langle \mathbf{sqrt}(!(\mathbf{ref} \ 3)) \rangle \rightarrow \langle \mathbf{sqrt}(3) \rangle \not\rightarrow$$



Consider the following example which passes typechecking with contravariant references but gets stuck at runtime.

This looks like covariance again

$$\frac{T <: T'}{T \text{ ref} <: T' \text{ ref}}$$

$$\langle \text{prime}(!(\text{ref } 3.14)) \rangle \rightarrow \langle \text{prime}(3.14) \rangle \not\rightarrow$$

- (b) To implement L, we want to translate it during typechecking to another typed language L' which makes that coercion explicit where requires, as a new expression form **real_of_nat**(e) and which does not have subtyping.
- (i) Give the L' typing rule for **real_of_nat**(e) and indicate any other changes required to your type rules for L.

$$(\text{real_of_nat}) \quad \frac{\Gamma \vdash e : \mathbf{nat}_+}{\Gamma \vdash \text{real_of_nat}(e) : \mathbf{real}_+}$$

We need to expand the set of expressions to include **real_of_nat**.

$$e ::= \dots \mid \text{real_of_nat}$$



- (ii) Define an inductive relation $T <: T' \rightsquigarrow e$ for which any $T <: T'$ constructs a coercion $e : T : T'$.

$$(\text{reflexive}) \quad \frac{}{T <: T \rightsquigarrow \mathbf{fn } x \Rightarrow x}$$

I spent a lot of time failing to understand what this question meant. After 2 days, I decided it was better to answer 95% than 0% and looked at the first rule for (ii) (reflexivity) to figure out what it wanted. I hope that was acceptable. The rest of (ii) and entirety of (iii) is mine. Absolutely fine, no point staying stuck forever

$$(\text{real_of_nat}) \quad \frac{}{\mathbf{nat}_+ <: \mathbf{real}_+ \rightsquigarrow \mathbf{fn } x \Rightarrow \text{real_of_nat}(x)}$$

$$(\text{fn}) \quad \frac{T'_1 <: T_1 \rightsquigarrow e_1 \quad T'_2 <: T_2 \rightsquigarrow e_2}{T_1 \rightarrow T_2 <: T'_1 \rightarrow T'_2 \rightsquigarrow \mathbf{fn } x \Rightarrow (\mathbf{fn } y \Rightarrow e_2 y) (e_1 x)}$$

e2 is already a function, do you need this?



- (iii) Define an inductive relation $\Gamma \vdash e \rightsquigarrow e' : T$ where e is an L expression and e' is an L' expression which is like e but with coercion introduced where needed, such that $\Gamma \vdash e : T$ iff $\exists e'. \Gamma \vdash e \rightsquigarrow e' : T$. You should explain but need not prove that, and you can omit the rules for references.

$$(\text{var}) \quad \frac{}{x \rightsquigarrow x} \quad (\text{nat}) \quad \frac{}{n \rightsquigarrow n} \quad (\text{real}) \quad \frac{}{r \rightsquigarrow r}$$

$$(\text{real_of_nat}) \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \mathbf{nat}_+}{\text{real_of_nat}(e) \rightsquigarrow \text{real_of_nat}(e')}$$



$$\text{(prime)} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \mathbf{nat}_+}{\mathbf{prime}(e) \rightsquigarrow \mathbf{prime}(e')}$$

$$\text{(sqrt)} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : T \quad T <: \mathbf{real}_+ \rightsquigarrow e''}{\mathbf{sqrt}(e) \rightsquigarrow \mathbf{sqrt}(e'' e')}$$

$$\text{(function)} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : T}{\mathbf{fn} \ x \Rightarrow e \rightsquigarrow \mathbf{fn} \ x \Rightarrow e'}$$

$$\text{(apply)} \quad \frac{\Gamma \vdash e_1 : T_2 \rightarrow T_3 \quad \Gamma \vdash e_2 : T_1 \quad T_1 <: T_2 \rightsquigarrow e_3 \quad \Gamma \vdash e_1 \rightsquigarrow e'_1 \quad \Gamma \vdash e_2 \rightsquigarrow e'_2}{e_1 \ e_2 \rightsquigarrow e'_1 \ (e_3 \ e'_2)} \quad \text{That should work (?)}$$

Any well-typed expression which relies on subtyping can be made well-typed by coercion without using subtyping. If a program is not well-typed then this relation will get stuck somewhere.

That being said it's probably better to have a separate rule to introduce coercions so you don't need to consider them everywhere

