COMP 321 MIDTERM Fall 2021

1 Logistics

- **Collaboration policy: you are not allowed to discuss the problems in this handout with anyone besides the professor. Once you begin working on the exam, you are not allowed to refer to any sources besides this semester's course materials (textbook and lecture materials).** However, there will be the usual TA sessions during the exam, and you are allowed to ask the TAs about any past homework problem or lecture material, including the lecture videos on product and sum types.
- To hand in, please upload `midterm.pdf` and `check.sml` and `step.sml` to the `midterm/` handin folder in your Google Drive handin folder.

2 Proofs and refutations of type safety

In the last homework, you looked at one way to add lists to a programming language. In this problem, we investigate two alternative ways of adding lists as a primitive type. These alternatives are not necessarily good ideas; it is your job to tell us whether they are or are not. In particular, we ask you to prove or refute progress and preservation for them.

For concreteness, we start from a language with numbers and strings and function types with the typing rules and operational semantics we have discussed for the following programs:

$$\begin{array}{rcl} \tau & ::= & \mathsf{number} \mid \mathsf{string} \mid \tau_1 \to \tau_2 \\ e & ::= & x \mid \mathsf{num}[k] \mid \mathsf{plus}(e_1, e_2) \mid \mathsf{str}[s] \mid \mathsf{cat}(e_1, e_2) \mid \lambda x : \tau.e \mid e_1\ e_2 \end{array}$$

In the following questions, you do not need to consider the cases of the proofs for the constructs in this fragment.

2.1 Head and Tail

Consider the following extension of the above language with lists of strings:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mathsf{list} \\ e & ::= & \ldots \mid \mathsf{nil} \mid \mathsf{cons}(e_1, e_2) \mid \mathsf{head}(e) \mid \mathsf{tail}(e) \end{array}$$

The idea is that $\mathsf{head}(e)$ gets the first element of a list, while $\mathsf{tail}(e)$ gets the list with the first element removed. Here are some attempted typing and operational semantics rules:

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}}\ \texttt{of-nil} \qquad \frac{\Gamma \vdash e_1 : \mathsf{string} \quad \Gamma \vdash e_2 : \mathsf{list}}{\Gamma \vdash \mathsf{cons}(e_1, e_2) : \mathsf{list}}\ \texttt{of-cons}$$

$$\frac{\Gamma \vdash e : \mathsf{list}}{\Gamma \vdash \mathsf{head}(e) : \mathsf{string}}\ \texttt{of-head} \qquad \frac{\Gamma \vdash e : \mathsf{list}}{\Gamma \vdash \mathsf{tail}(e) : \mathsf{list}}\ \texttt{of-tail}$$

$\boxed{e\ \mathsf{done}}$

$$\frac{}{\mathsf{nil}\ \mathsf{done}}\ \texttt{done-empty} \qquad \frac{e_1\ \mathsf{done} \quad e_2\ \mathsf{done}}{\mathsf{cons}(e_1, e_2)\ \mathsf{done}}\ \texttt{done-cons}$$

$\boxed{e_1 \mapsto e_2}$

$$\frac{e_1 \mapsto e_1'}{\mathsf{cons}(e_1, e_2) \mapsto \mathsf{cons}(e_1', e_2)}\ \texttt{step-cons-search} \qquad \frac{e_1\ \mathsf{done} \quad e_2 \mapsto e_2'}{\mathsf{cons}(e_1, e_2) \mapsto \mathsf{cons}(e_1, e_2')}\ \texttt{step-cons-search-2}$$

$$\frac{e \mapsto e'}{\mathsf{head}(e) \mapsto \mathsf{head}(e')}\ \texttt{step-head-search} \qquad \frac{\mathsf{cons}(e_1, e_2)\ \mathsf{done}}{\mathsf{head}(\mathsf{cons}(e_1, e_2)) \mapsto e_1}\ \texttt{step-head-instr}$$

$$\frac{e \mapsto e'}{\mathsf{tail}(e) \mapsto \mathsf{tail}(e')}\ \texttt{step-tail-search} \qquad \frac{\mathsf{cons}(e_1, e_2)\ \mathsf{done}}{\mathsf{tail}(\mathsf{cons}(e_1, e_2)) \mapsto e_2}\ \texttt{step-tail-instr}$$

For example, under these semantics,

$$\cdot \vdash \mathsf{head}(\mathsf{tail}(\mathsf{cons}(\mathsf{str}[\texttt{vanilla}], \mathsf{cons}(\mathsf{str}[\texttt{chocolate}], \mathsf{cons}(\mathsf{str}[\texttt{swirl}], \mathsf{nil}))))) : \mathsf{string}$$

and

$$\begin{aligned} & \mathsf{head}(\mathsf{tail}(\mathsf{cons}(\mathsf{str}[\texttt{vanilla}], \mathsf{cons}(\mathsf{str}[\texttt{chocolate}], \mathsf{cons}(\mathsf{str}[\texttt{swirl}], \mathsf{nil}))))) \\ \mapsto\ & \mathsf{head}(\mathsf{cons}(\mathsf{str}[\texttt{chocolate}], \mathsf{cons}(\mathsf{str}[\texttt{swirl}], \mathsf{nil}))) \\ \mapsto\ & \mathsf{str}[\texttt{chocolate}] \end{aligned}$$

One of preservation/progress is true and the other is false. For the one that is true, prove it. For the one that is false, refute it by giving a concrete counterexample. (To refute preservation, give a closed, well-typed expression that steps to something of a different type. To refute progress, give a closed, well-typed expression that is not done and does not take a step.)

Task 1 (25%).

- Prove or refute preservation:

  If $e \mapsto e'$ and $\cdot \vdash e : \tau$ then $\cdot \vdash e' : \tau$.

- Prove or refute progress:

  If $\cdot \vdash e : \tau$ then $e\ \mathsf{done}$ or there exists an e' such that $e \mapsto e'$.

- Briefly describe a way to make both progress and preservation hold for a language with these expressions for lists ($\mathsf{nil}, \mathsf{cons}(e_1, e_2), \mathsf{head}(e), \mathsf{tail}(e)$) by changing the type system or operational semantics.

2.2 Heterogeneous Lists

Now, we instead extend the language with a type of *heterogeneous* lists, which can contain either numbers or strings. That is, a single list can contain elements of more than one type. There are two kinds of cons expressions, one that adds a number to a list, and one that adds a string. There are also two case analysis operations, one where the head of the list is assumed to be a number, and one where it is assumed to be a string. (For this problem, we focus on case analysis for lists only, leaving out recursion.)

$$\tau ::= \ldots \mid \mathsf{list}$$
$$e ::= \ldots \mid \mathsf{nil} \mid \mathsf{cons}_{\mathsf{number}}(e_1, e_2) \mid \mathsf{cons}_{\mathsf{string}}(e_1, e_2) \mid \mathsf{listcase}_{\mathsf{number}}(e)\{e_1, x.y.e_2\} \mid \mathsf{listcase}_{\mathsf{string}}(e)\{e_1, x.y.e_2\}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}}\ \texttt{of-nil}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{number} \quad \Gamma \vdash e_2 : \mathsf{list}}{\Gamma \vdash \mathsf{cons}_{\mathsf{number}}(e_1, e_2) : \mathsf{list}}\ \texttt{of-consn} \qquad \frac{\Gamma \vdash e_1 : \mathsf{string} \quad \Gamma \vdash e_2 : \mathsf{list}}{\Gamma \vdash \mathsf{cons}_{\mathsf{string}}(e_1, e_2) : \mathsf{list}}\ \texttt{of-conss}$$

$$\frac{\Gamma \vdash e : \mathsf{list} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x{:}\mathsf{number}, y{:}\mathsf{list} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{listcase}_{\mathsf{number}}(e)\{e_1, x.y.e_2\} : \tau}\ \texttt{of-listcasen}$$

$$\frac{\Gamma \vdash e : \mathsf{list} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x{:}\mathsf{string}, y{:}\mathsf{list} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{listcase}_{\mathsf{string}}(e)\{e_1, x.y.e_2\} : \tau}\ \texttt{of-listcases}$$

$\boxed{e\ \mathsf{done}}$

$$\frac{}{\mathsf{nil\ done}}\ \texttt{done-empty} \qquad \frac{e_1\ \mathsf{done} \quad e_2\ \mathsf{done}}{\mathsf{cons}_{\mathsf{number}}(e_1, e_2)\ \mathsf{done}}\ \texttt{done-consn} \qquad \frac{e_1\ \mathsf{done} \quad e_2\ \mathsf{done}}{\mathsf{cons}_{\mathsf{string}}(e_1, e_2)\ \mathsf{done}}\ \texttt{done-conss}$$

$\boxed{e_1 \mapsto e_2}$

$$\frac{e_1 \mapsto e_1'}{\mathsf{cons}_{\mathsf{number}}(e_1, e_2) \mapsto \mathsf{cons}_{\mathsf{number}}(e_1', e_2)}\ \texttt{step-consn-search}$$

$$\frac{e_1\ \mathsf{done} \quad e_2 \mapsto e_2'}{\mathsf{cons}_{\mathsf{number}}(e_1, e_2) \mapsto \mathsf{cons}_{\mathsf{number}}(e_1, e_2')}\ \texttt{step-consn-search-2}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{cons}_{\mathsf{string}}(e_1, e_2) \mapsto \mathsf{cons}_{\mathsf{string}}(e_1', e_2)}\ \texttt{step-conss-search}$$

$$\frac{e_1\ \mathsf{done} \quad e_2 \mapsto e_2'}{\mathsf{cons}_{\mathsf{string}}(e_1, e_2) \mapsto \mathsf{cons}_{\mathsf{string}}(e_1, e_2')}\ \texttt{step-conss-search-2}$$

$$\frac{e \mapsto e'}{\mathsf{listcase}_{\mathsf{number}}(e)\{e_1, x.y.e_2\} \mapsto \mathsf{listcase}_{\mathsf{number}}(e')\{e_1, x.y.e_2\}}\ \texttt{step-listcasen-search}$$

$$\frac{}{\mathsf{listcase}_{\mathsf{number}}(\mathsf{nil})\{e_1, x.y.e_2\} \mapsto e_1}\ \texttt{step-listcasen-instr-1}$$

$$\frac{\mathsf{cons}_{\mathsf{number}}(e_h, e_t)\,\mathsf{done}}{\mathsf{listcase}_{\mathsf{number}}(\mathsf{cons}_{\mathsf{number}}(e_h, e_t))\{e_1, x.y.e_2\} \mapsto e_2[e_h/x][e_t/y]}\ \texttt{step-listcasen-instr-2}$$

$$\frac{\mathsf{cons}_{\mathsf{string}}(e_h, e_t)\,\mathsf{done}}{\mathsf{listcase}_{\mathsf{number}}(\mathsf{cons}_{\mathsf{string}}(e_h, e_t))\{e_1, x.y.e_2\} \mapsto e_2[e_h/x][e_t/y]}\ \texttt{step-listcasen-instr-3}$$

$$\frac{e \mapsto e'}{\mathsf{listcase}_{\mathsf{string}}(e)\{e_1, x.y.e_2\} \mapsto \mathsf{listcase}_{\mathsf{string}}(e')\{e_1, x.y.e_2\}}\ \texttt{step-listcases-search}$$

$$\frac{}{\mathsf{listcase}_{\mathsf{string}}(\mathsf{nil})\{e_1, x.y.e_2\} \mapsto e_1}\ \texttt{step-listcases-instr-1}$$

$$\frac{\mathsf{cons}_{\mathsf{number}}(e_h, e_t)\,\mathsf{done}}{\mathsf{listcase}_{\mathsf{string}}(\mathsf{cons}_{\mathsf{number}}(e_h, e_t))\{e_1, x.y.e_2\} \mapsto e_2[e_h/x][e_t/y]}\ \texttt{step-listcases-instr-2}$$

$$\frac{\mathsf{cons}_{\mathsf{string}}(e_h, e_t)\,\mathsf{done}}{\mathsf{listcase}_{\mathsf{string}}(\mathsf{cons}_{\mathsf{string}}(e_h, e_t))\{e_1, x.y.e_2\} \mapsto e_2[e_h/x][e_t/y]}\ \texttt{step-listcases-instr-3}$$

For example, under these semantics,

$$\cdot \vdash \mathsf{listcase}_{\mathsf{number}}(\mathsf{cons}(\mathsf{plus}(\mathsf{num}[3], \mathsf{num}[4]), \mathsf{cons}(\mathsf{str}[\texttt{samurai}], \mathsf{nil})))\{9, x.y.11\} : \mathsf{number}$$

and

$$\begin{array}{rl} & \mathsf{listcase}_{\mathsf{number}}(\mathsf{cons}(\mathsf{plus}(\mathsf{num}[3], \mathsf{num}[4]), \mathsf{cons}(\mathsf{str}[\texttt{samurai}], \mathsf{nil})))\{9, x.y.11\} \\ \mapsto^* & \mathsf{listcase}_{\mathsf{number}}(\mathsf{cons}(\mathsf{num}[7], \mathsf{cons}(\mathsf{str}[\texttt{samurai}], \mathsf{nil})))\{9, x.y.11\} \\ \mapsto & 11 \end{array}$$

Note that the operational semantics are exactly the same for $\mathsf{cons}_{\mathsf{number}}(e_1, e_2)$ and $\mathsf{cons}_{\mathsf{string}}(e_1, e_2)$, and for $\mathsf{listcase}_{\mathsf{number}}(e)\{e_1, x.y.e_2\}$ and $\mathsf{listcase}_{\mathsf{string}}(e)\{e_1, x.y.e_2\}$, but the two have different typing rules. In the proofs, you can write just $\mathsf{cons}(e_1, e_2)$ and $\mathsf{listcase}(e)\{e_1, x.y.e_2\}$ and leave off the number/stringannotation when the same analogous proof applies to both.

One of preservation/progress is true and the other is false. For the one that is true, prove it. For the one that is false, refute it by giving a concrete counterexample. (To refute preservation, give a closed, well-typed expression that steps to something of a different type. To refute progress, give a closed, well-typed expression that is not done and does not take a step.)

Task 1 (25%).

- Prove or refute preservation:

  If $e \mapsto e'$ and $\cdot \vdash e : \tau$ then $\cdot \vdash e' : \tau$.

  You can assume that the substitution property holds: For all $\Gamma, \Gamma', e, e', x, \tau, \tau'$, if $\Gamma, x : \tau, \Gamma' \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$ then $\Gamma, \Gamma' \vdash e'[e/x] : \tau'$.

- Prove or refute progress:

  If $\cdot \vdash e : \tau$ then e done or there exists an e' such that $e \mapsto e'$.

- Fix the one that is false by keeping both $\mathsf{cons}_{\mathsf{number}}(e_1, e_2)$ and $\mathsf{cons}_{\mathsf{string}}(e_1, e_2)$ and their typing rules and operational semantics the same, but changing listcase. Show any typing and operational semantics rules that change. You should ensure that both progress and preservation hold for your definitions, but you do not need to hand in the proofs.

3 Implementation

In this problem, you will implement the typing rules and operational semantics for unary and binary product types and binary sum types, as described in Chapters 10 and 11 of PFPL. **Collaboration policy note for this problem: you are allowed to discuss product and sum types (the material in the textbook; the lecture videos) with each other and with the professor and TAs. However, you are not allowed to discuss or share any aspect of the implementation of these rules.**

The correspondence between the abstract syntax in PFPL and the SML datatype is

$\tau_1 \times \tau_2$	`Product(t1,t2)`
$\tau_1 + \tau_2$	`Sum(t1,t2)`
$\langle\rangle$	`Triv`
$\langle e_1, e_2 \rangle$	`Pair(e1,e2)`
$e \cdot \mathsf{l}$	`ProjLeft e`
$e \cdot \mathsf{r}$	`ProjRight e`
$\mathsf{l} \cdot e$	`InLeft(t,e)`
$\mathsf{r} \cdot e$	`InRight(t,e)`
$\mathsf{case}(e)\{\mathsf{l} \cdot x \hookrightarrow e_1 \mid \mathsf{r} \cdot y \hookrightarrow e_2\}$	`Case(e,(x,e1),(y,e2)`

Task 1 (25%). Implement the typing rules and operational semantics for nullary and binary products on page 84 (Section 10.1) in the function `progress` in `step.sml` and the function `checkOpen` in `check.sml`. You should implement the eager version of the operational semantics rules (include the bracketed rules and premises).

Task 2 (25%). Implement the typing rules and operational semantics for binary sums on page 90 (Section 11.1) in the function `progress` in `step.sml` and the function `checkOpen` in `check.sml`. You should implement the eager version of the operational semantics rules (include the bracketed rules and premises). To make type checking easier, `InLeft(t,e)` and `InRight(t,e)` are labeled with the other part of the sum type besides the type of `e`:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inleft}(\tau_2, e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inright}(\tau_1, e) : \tau_1 + \tau_2}$$

Some examples for testing are in `examples.t`. The concrete syntax understood by the parser is

`(t1 * t2)`	`Product(t1,t2)`
`(t1 + t2)`	`Sum(t1,t2)`
`<>`	`Triv`
`< e1 , e2 >`	`Pair(e1,e2)`
`projleft e`	`ProjLeft e`
`projright e`	`ProjRight e`
`inleft[t] e`	`InLeft(t,e)`
`inright[t] e`	`InRight(t,e)`
`case e {inleft x => e1 | inright y => e2}`	`Case(e,(x,e1),(y,e2))`