Assignment #4, Update X: Imperative Programming with Algol

15-312: Principles of Programming Languages

Out: Tuesday, March 25, 2014
Due: Tuesday, April 8, 2014 1:29PM

Introduction

In this assignment, we will explore imperative programming in Algol.

Submission

We will collect *exactly* the following files from the `/afs/andrew/course/15/312/` directory:

```
handin/<yourandrewid>/assn4/assn4.pdf
handin/<yourandrewid>/assn4/typechecker.sml
handin/<yourandrewid>/assn4/dynamics.sml
handin/<yourandrewid>/assn4/elaborator.sml
handin/<yourandrewid>/assn4/sort.alg
handin/<yourandrewid>/assn4/demo.alg
```

Make sure that your files have the right names (especially `assn4.pdf`!) and are in the correct directories.

1 Modernized Algol

Imperative programming is distinguished by its use of *state*. Stateful procedures differ from pure functions in that their output does not depend only on explicitly-provided input, but also on the state of any *assignables* in their scope. Assignables are a form of *symbol* that refers to a piece of mutable state in a memory. In many imperative languages, assignables are called variables. But the concept of an assignable is different from the concept of a variable – variables stand for an unknown expression and are given meaning by substitution, while assignables name a piece of mutable state that can be manipulated by various operators.

In many programming languages, including Standard ML, stateful operations can occur anywhere. This simplifies the use of state and effects when it is necessary and allows for what are called *benign effects*, such as the use of state internally to speed up or otherwise simplify the implementation of an function that behaves purely functionally externally.

In other languages, notably Haskell and originally Algol 60, there is a stricter separation between terms that can have effects (we will call such terms *commands*) and purely functional terms (we will call such terms *expressions*). This allows us to reason separately about each sort of term and isolates regions where each kind of reasoning may be required. On the other hand, it eliminates the possibility of benign effects and otherwise makes incidental uses of effects more unwieldy.

Sort			Abstract Form	Concrete Form
Type	τ	::=	$\texttt{nat}$	$\texttt{nat}$
			$\texttt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$
			$\texttt{unit}$	$\texttt{unit}$
			$\texttt{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$
			$\texttt{void}$	$\texttt{void}$
			$\texttt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$
			$\texttt{rec}(t.\tau)$	$\mu t.\tau$
			$\texttt{cmd}(\tau)$	$\texttt{cmd}(\tau)$
			$\texttt{ref}(\tau)$	$\texttt{ref}(\tau)$
Exp	e	::=	x	x
			$\texttt{z}$	$\texttt{z}$
			$\texttt{s}(e)$	$\texttt{s}(e)$
			$\texttt{ifz}(e;e_0;x.e_1)$	$\texttt{ifz}\ e\ \{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$
			$\texttt{let}(e_0;x.e_1)$	$\texttt{let}\ x = e_0\ \texttt{in}\ e_1$
			$\texttt{lam}[\tau](x.e)$	$\texttt{fn}\,(x{:}\tau)\,e$
			$\texttt{ap}(e_1;e_2)$	$e_1\,(e_2)$
			$\texttt{fix}[\tau](x.e)$	$\texttt{fix}\ x{:}\tau\ \texttt{is}\ e$
			$\texttt{abort}[\tau](e)$	$\texttt{abort}[\tau](e)$
			$\texttt{in}[\tau_1;\tau_2][\texttt{l}](e)$	$\texttt{inl}[\tau_1;\tau_2](e)$
			$\texttt{in}[\tau_1;\tau_2][\texttt{r}](e)$	$\texttt{inr}[\tau_1;\tau_2](e)$
			$\texttt{case}(e;x_1.e_1;x_2.e_2)$	$\texttt{case}\ e\ \{\texttt{inl}(x_1) \Rightarrow e_1 \mid \texttt{inr}(x_2) \Rightarrow e_2\}$
			$\texttt{triv}$	$\langle\rangle$
			$\texttt{pair}(e_1;e_2)$	$\langle e_1, e_2\rangle$
			$\texttt{pr}[\texttt{l}](e)$	$e \cdot \texttt{l}$
			$\texttt{pr}[\texttt{r}](e)$	$e \cdot \texttt{r}$
			$\texttt{fold}[t.\tau](e)$	$\texttt{fold}[t.\tau](e)$
			$\texttt{unfold}(e)$	$\texttt{unfold}(e)$
			$\texttt{cmd}(m)$	$\texttt{cmd}(m)$
			$\texttt{ref}[a]$	$\&a$
Cmd	m	::=	$\texttt{ret}(e)$	$\texttt{ret}\ e$
			$\texttt{bnd}(e;x.m)$	$\texttt{bnd}\ x \leftarrow e; m$
			$\texttt{dcl}[\tau](e;a.m)$	$\texttt{dcl}\ a := e\ \texttt{in}\ m$
			$\texttt{get}[a]$	$@a$
			$\texttt{set}[a](e)$	$a := e$
			$\texttt{getref}(e)$	$\star e$
			$\texttt{setref}(e_1;e_2)$	$e_1 := e_2$
			$\texttt{cast}[l](e,x.m)$	$\texttt{inl}\ x = e; m$
			$\texttt{cast}[r](e,x.m)$	$\texttt{inr}\ x = e; m$

Figure 1: Expressions and types in our version of Modernized Algol

There is no universally better approach, but we will study a language of this latter sort in this assignment. The language is an extension of the Modernized Algol language with typed commands defined in Ch. 36 and 37 of *PFPL*.

1.1 Algol with assignables and references

We begin with an extension the definition from Section 2 of Homework 2. The expression language is extended with three new features:

- Recursive types: $\texttt{fold}[t.\tau](e)$ and $\texttt{unfold}(e)$ are the introduction and elimination forms for the type $\mu t.\tau$. (We are adding in recursive types now to make programming more interesting; there's no essential connection between Algol and recursive types.)
- Encapsulated commands $\texttt{cmd}(m)$. If m is an Algol command that should eventually return a value of type τ, then $\texttt{cmd}(m)$ has type $\texttt{cmd}(\tau)$.
- Encapsulated assignables $\&a$, that act like pointers. We'll get back to these in Section 1.2.

On the Algol side, there are five constructs that handle sequencing and assignables:

- $\texttt{ret}\ e$ is the command that returns the value of e.
- $\texttt{bnd}\ x \leftarrow e_1; m_2$ is the command that evaluates the expression e_1 until it becomes an encapsulated command $\texttt{cmd}(m_1)$, then evaluates that encapsulated command m_1 until it returns a value v_1, and *then* passes v_1 to m_2 by substituting for x.
- $\texttt{dcl}\ a := e\ \texttt{in}\ m$ is the command that declares an assignable a, with initial value e, within the scope of m.
- $@a$ is the command that retrieves the value associated with assignable a.
- $a := e$ is the command that sets the value associated with assignable a to the value of e.

Statics We have to make one key change to the statics from Homework 2: each rule in the statics is now uniformly indexed by a *signature*, Σ, which associates types with the symbols in scope using the form $a \sim \tau$.

$$\frac{\Gamma \vdash_\Sigma e : [\mu t.\tau/t]\tau}{\Gamma \vdash_\Sigma \texttt{fold}[t.\tau](e) : \mu t.\tau}(\mu\text{-}I) \qquad \frac{\Gamma \vdash_\Sigma e : \mu t.\tau}{\Gamma \vdash_\Sigma \texttt{unfold}(e) : [\mu t.\tau/t]\tau}(\mu\text{-}E) \qquad \frac{\Gamma \vdash_\Sigma m \sim \tau}{\Gamma \vdash_\Sigma \texttt{cmd}(m) : \texttt{cmd}(\tau)}(\texttt{cmd}\text{-}I)$$

Commands also have types associated with them:

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad \tau\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \texttt{ret}(e) \sim \tau}(\texttt{ret}\text{-}M) \qquad \frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\tau) \qquad \Gamma, x : \tau \vdash_\Sigma m \sim \tau'}{\Gamma \vdash_\Sigma \texttt{bnd}(e; x.m) \sim \tau'}(\texttt{bnd}\text{-}M)$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad \tau\ \mathsf{mobile} \qquad \Gamma \vdash_{\Sigma, a \sim \tau} m \sim \tau'}{\Gamma \vdash_\Sigma \texttt{dcl}[\tau](e; a.m) \sim \tau'}(\texttt{dcl}\text{-}M)$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{get}[a] \sim \tau}(\texttt{get}\text{-}M) \qquad \frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{set}[a](e) \sim \tau}(\texttt{set}\text{-}M)$$

The types of values that can be stored in memory and returned from commands is restricted according to a *mobility condition*, denoted τ mobile, which serves to ensure that assignables can only be used within their scope – this is called the *stack discipline*. To guarantee this, any types classifying values that may internally refer to assignables cannot be deemed *mobile*, and thus cannot be stored in memory or returned from a command. Specifically, function types are not mobile because functions may contain encapsulated commands that refer to assignables, and encapsulated commands themselves are not mobile.

Mobility We're using recursive types and would like to think of types like $\mu t.\texttt{unit} + t$ as being mobile. To do so, we need to define a hypothetical judgment t_1 mobile, $\ldots, t_n$ mobile $\vdash \tau$ mobile which says that τ is mobile if $t_1 \ldots t_n$ are. We use Δ as shorthand for the list of assumptions t_1 mobile, $\ldots, t_n$ mobile and, as usual, write τ mobile as shorthand for $\emptyset \vdash \tau$ mobile.

$$\frac{t \text{ mobile} \in \Delta}{\Delta \vdash t \text{ mobile}}(0) \qquad \frac{}{\Delta \vdash \texttt{nat} \text{ mobile}}(1) \qquad \frac{}{\Delta \vdash \texttt{unit} \text{ mobile}}(2) \qquad \frac{}{\Delta \vdash \texttt{void} \text{ mobile}}(3)$$

$$\frac{\Delta \vdash \tau_1 \text{ mobile} \qquad \Delta \vdash \tau_2 \text{ mobile}}{\Delta \vdash \tau_1 + \tau_2 \text{ mobile}}(4) \qquad \frac{\Delta \vdash \tau_1 \text{ mobile} \qquad \Delta \vdash \tau_2 \text{ mobile}}{\Delta \vdash \tau_1 \times \tau_2 \text{ mobile}}(5) \qquad \frac{\Delta, t \text{ mobile} \vdash \tau \text{ mobile}}{\Delta \vdash \mu t.\tau \text{ mobile}}(6)$$

Dynamics The dynamics of expressions are also based on the dynamics given in Section 2 of Homework 2, with one change: the judgments are indexed by a signature, just as with the statics. This information is helpful to allow us to prove safety theorems, as we will see shortly. There are two forms of judgments as before:

$$\begin{array}{ll} e \text{ val}_\Sigma & e \text{ is a value relative to } \Sigma \\ e \underset{\Sigma}{\mapsto} e' & e \text{ steps to } e' \text{ relative to } \Sigma \end{array}$$

In addition to augmenting each judgment with the signature, we need to modify the rules from Homework 2 to add rules for recursive types and for the new expression $\texttt{cmd}(m)$:

$$\frac{e \text{ val}_\Sigma}{\texttt{fold}[t.\tau](e) \text{ val}_\Sigma}(7) \qquad \frac{}{\texttt{cmd}(m) \text{ val}_\Sigma}(8)$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{fold}[t.\tau](e) \underset{\Sigma}{\mapsto} \texttt{fold}[t.\tau](e')}(9) \qquad \frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{unfold}(e) \underset{\Sigma}{\mapsto} \texttt{unfold}(e')}(10) \qquad \frac{\texttt{fold}[t.\tau](e) \text{ val}_\Sigma}{\texttt{unfold}(\texttt{fold}[t.\tau](e)) \underset{\Sigma}{\mapsto} e}(11)$$

Commands themselves also have dynamics associated with them. Because commands work with state, we need to define the notion of a *memory*. A memory is simply a finite mapping from symbols to values:

$$\mu ::= \emptyset \mid \mu \otimes (a \hookrightarrow e)$$

A pairing of a command with a memory is called a *state* and is denoted $m \parallel \mu$. The dynamics of the command language are defined over states by the following judgments:

$$\begin{array}{ll} m \parallel \mu \text{ final}_\Sigma & \text{The state } m \parallel \mu \text{ is fully executed relative to } \Sigma. \\ m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu' & \text{The state } m \parallel \mu \text{ steps to the state } m' \parallel \mu' \text{ relative to } \Sigma. \end{array}$$

The following rules define the behavior of core Algol features:

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{ret}(e) \parallel \mu\ \mathsf{final}_\Sigma}(12) \qquad \frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{ret}(e) \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{ret}(e') \parallel \mu}(13) \qquad \frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{bnd}(e; x.m) \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{bnd}(e'; x.m) \parallel \mu}(14)$$

$$\frac{m_1 \parallel \mu \underset{\Sigma}{\longmapsto} m_1' \parallel \mu'}{\mathtt{bnd}(\mathtt{cmd}(m_1); x.m_2) \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{bnd}(\mathtt{cmd}(m_1'); x.m_2) \parallel \mu'}(15) \qquad \frac{e\ \mathsf{val}_\Sigma}{\mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e)); x.m) \parallel \mu \underset{\Sigma}{\longmapsto} [e/x]m \parallel \mu}(16)$$

$$\frac{}{\mathtt{get}[a] \parallel \mu \otimes (a \hookrightarrow e) \underset{\Sigma, a\sim\tau}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes (a \hookrightarrow e)}(17)$$

$$\frac{e \underset{\Sigma, a\sim\tau}{\longmapsto} e'}{\mathtt{set}[a](e) \parallel \mu \underset{\Sigma, a\sim\tau}{\longmapsto} \mathtt{set}[a](e') \parallel \mu}(18) \qquad \frac{e\ \mathsf{val}_{\Sigma, a\sim\tau}}{\mathtt{set}[a](e) \parallel \mu \otimes (a \hookrightarrow \_) \underset{\Sigma, a\sim\tau}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes (a \hookrightarrow e)}(19)$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{dcl}[\tau](e; a.m) \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{dcl}[\tau](e'; a.m) \parallel \mu}(20)$$

$$\frac{e\ \mathsf{val}_\Sigma \qquad m \parallel \mu \otimes (a \hookrightarrow e) \underset{\Sigma, a\sim\tau}{\longmapsto} m' \parallel \mu' \otimes (a \hookrightarrow e')}{\mathtt{dcl}[\tau](e; a.m) \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{dcl}[\tau](e'; a.m') \parallel \mu'}(21) \qquad \frac{e\ \mathsf{val}_\Sigma \qquad e'\ \mathsf{val}_{\Sigma, a\sim\tau}}{\mathtt{dcl}[\tau](e; a.\mathtt{ret}(e')) \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{ret}(e') \parallel \mu}(22)$$

1.2 References

References *could* be defined by elaboration in the next section by defining the type $\mathtt{ref}(\tau)$ to be the type $\mathtt{cmd}(\tau) \times (\tau \rightharpoonup \mathtt{cmd}(\tau))$, but we'll instead define references explicitly. This introduces a new value to the expression language, $\&a$, and two new commands. The command $\star e$ evaluates e to some reference value $\&a$, then gets the value associated with a, and the command $e_1 := e_2$ evaluates e_1 to some reference value $\&a$ and then stores the value of e_2 there.

Statics

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{ref}[a] : \mathtt{ref}(\tau)}(\mathtt{ref}\text{-}I) \qquad \frac{\Gamma \vdash_\Sigma e : \mathtt{ref}(\tau)}{\Gamma \vdash_\Sigma \mathtt{getref}(e) \sim \tau}(\mathtt{getref}\text{-}M)$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{ref}(\tau) \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{setref}(e_1; e_2) \sim \tau}(\mathtt{setref}\text{-}M)$$

Dynamics

$$\frac{}{\texttt{ref}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}}(23) \qquad \frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{getref}(e) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{getref}(e') \parallel \mu}(24)$$

$$\frac{}{\texttt{getref}(\texttt{ref}[a]) \parallel \mu \otimes (a \hookrightarrow e) \underset{\Sigma}{\longmapsto} \texttt{ret}(e) \parallel \mu \otimes (a \hookrightarrow e)}(25)$$

$$\frac{e_1 \underset{\Sigma,a\sim\tau}{\longmapsto\!\!\!\!\!\rightarrow} e_1'}{\texttt{setref}(e_1; e_2) \parallel \mu \underset{\Sigma,a\sim\tau}{\longmapsto\!\!\!\!\!\rightarrow} \texttt{setref}(e_1'; e_2) \parallel \mu}(26) \qquad \frac{e_1\ \mathsf{val}_\Sigma \qquad e_2 \underset{\Sigma}{\longmapsto} e_2'}{\texttt{setref}(e_1; e_2) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{setref}(e_1; e_2') \parallel \mu}(27)$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{setref}(\texttt{ref}[a]; e) \parallel \mu \otimes (a \hookrightarrow \_) \underset{\Sigma}{\longmapsto} \texttt{ret}(e) \parallel \mu \otimes (a \hookrightarrow e)}(28)$$

1.3 Casting and errors

In class, we only discussed using Algol's commands to capture one kind of *effect*: keeping around and modifying memory. It's possible to capture other kinds of effects. For instance, without adding errors to our evaluation of expressions, we can add errors to the evaluation of commands by adding the commands $\texttt{inl}\ x = e; m$ and $\texttt{inr}\ x = e; m$ that attempt to forcibly extract the `inl` or `inr` portion of a sum type, substituting the subterm into m if the tags match and failing otherwise. This is quite similar to the casting operations in Hybrid PCF, but casts were expressions in Hybrid PCF. We have made them commands in this assignment to restrict errors to the evaluation of commands.

Statics

$$\frac{\Gamma \vdash_\Sigma e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash_\Sigma m \sim \tau}{\Gamma \vdash_\Sigma \texttt{cast}[\texttt{l}](e, x.m) \sim \tau}(\texttt{castl}\text{-}M) \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_2 \vdash_\Sigma m \sim \tau}{\Gamma \vdash_\Sigma \texttt{cast}[\texttt{r}](e, x.m) \sim \tau}(\texttt{castr}\text{-}M)$$

Dynamics We need a new judgment $m \parallel \mu\ \mathsf{err}_\Sigma$ that captures errors in the evaluation of commands, even though the evaluation of expressions remains error-free.

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{cast}[\texttt{l}](e, x.m) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{cast}[\texttt{l}](e', x.m) \parallel \mu}(29) \qquad \frac{\texttt{in}[\tau_1; \tau_2][\texttt{l}](e)\ \mathsf{val}_\Sigma}{\texttt{cast}[\texttt{l}](\texttt{in}[\tau_1; \tau_2][\texttt{l}](e), x.m) \parallel \mu \underset{\Sigma}{\longmapsto} [e/x]m \parallel \mu}(30)$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{cast}[\texttt{r}](e, x.m) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{cast}[\texttt{r}](e', x.m) \parallel \mu}(31) \qquad \frac{\texttt{in}[\tau_1; \tau_2][\texttt{r}](e)\ \mathsf{val}_\Sigma}{\texttt{cast}[\texttt{r}](\texttt{in}[\tau_1; \tau_2][\texttt{r}](e), x.m) \parallel \mu \underset{\Sigma}{\longmapsto} [e/x]m \parallel \mu}(32)$$

$$\frac{\texttt{in}[\tau_1; \tau_2][\texttt{r}](e)\ \mathsf{val}_\Sigma}{\texttt{cast}[\texttt{l}](\texttt{in}[\tau_1; \tau_2][\texttt{r}](e), x.m) \parallel \mu\ \mathsf{err}_\Sigma}(33) \qquad \frac{\texttt{in}[\tau_1; \tau_2][\texttt{l}](e)\ \mathsf{val}_\Sigma}{\texttt{cast}[\texttt{r}](\texttt{in}[\tau_1; \tau_2][\texttt{l}](e), x.m) \parallel \mu\ \mathsf{err}_\Sigma}(34)$$

$$\frac{m \parallel \mu\ \mathsf{err}_\Sigma}{\texttt{bnd}(\texttt{cmd}(m); x.m') \parallel \mu\ \mathsf{err}_\Sigma}(35) \qquad \frac{m \parallel \mu \otimes (a \hookrightarrow \_)\ \mathsf{err}_{\Sigma,a\sim\tau}}{\texttt{dcl}[\tau](e; a.m) \parallel \mu\ \mathsf{err}_\Sigma}(36)$$

1.4 Safety

You may have found it strange that the dynamics for commands maintain typing information via the signature, Σ. The purpose of this is to facilitate proofs of safety for our language (indeed, that is basically its only purpose – in the implementation below, you will not keep track of this information in the dynamics, nor will the type ascription on `dcl` appear in the dynamics). More specifically, the memory simply maps assignables to values but we need to access the types of those values to prove safety. Thus, we define a judgment of the form $\mu : \Sigma$ that states that the memory μ is well-typed with respect to Σ. This judgment is inductively defined according to the following rules:

$$\frac{}{\emptyset : \emptyset}(37) \qquad \frac{\mu : \Sigma \qquad e\ \mathsf{val}_\emptyset \qquad \emptyset \vdash_\emptyset e : \tau \qquad \tau\ \mathsf{mobile}}{\mu \otimes (a \hookrightarrow e) : \Sigma, a \sim \tau}(38)$$

Note the requirement that the values in the memory be well-typed relative to an empty signature. This is consistent with the stack discipline we described earlier: the memory should contain only values that do not refer to assignables. You will end up deriving how this invariant is actually maintained in the proof of preservation below by using the premise τ mobile present in the statics for `dcl`. A key fact about mobility that you will need is the following:

Lemma 1 (Mobility)**.** *If* τ mobile *and* $\emptyset \vdash_\Sigma e : \tau$ *and* $e\ \mathsf{val}_\Sigma$ *then* $\emptyset \vdash_\emptyset e : \tau$ *and* $e\ \mathsf{val}_\emptyset$.

It is straightforward to prove this (an appropriately modified canonical forms lemma for the expression language in Homework 2 is helpful), but you can just cite the Mobility lemma without proof below. You may also cite the following lemmas without proof below as well:

Lemma 2 (Inversion of (38))**.** *If* $\mu = \mu' \otimes (a \hookrightarrow e)$ *and* $\Sigma = \Sigma', a \sim \tau$ *and* $\mu : \Sigma$*, then* $\mu' : \Sigma'$ *and* $e\ \mathsf{val}_\emptyset$ *and* $\emptyset \vdash_\emptyset e : \tau$ *and* τ mobile*.*

Lemma 3 (Canonical Forms for Encapsulated Commands)**.** *If* $\Gamma \vdash_\Sigma e : \mathtt{cmd}(\tau)$ *and* $e\ \mathsf{val}_\Sigma$ *then* $e = \mathtt{cmd}(m)$ *for some* m *such that* $\Gamma \vdash_\Sigma m \sim \tau$.

Lemma 4 (Canonical Forms for Commands)**.** *If* $\Gamma \vdash_\Sigma m \sim \tau$ *and* $m \parallel \mu\ \mathsf{final}_\Sigma$ *then* τ mobile *and* $m = \mathtt{ret}(e)$ *for some* e *such that* $e\ \mathsf{val}_\Sigma$ *and* $\Gamma \vdash e : \tau$.

Lemma 5 (Weakening for Signatures)**.** *If* $\Gamma \vdash_\Sigma e : \tau$ *and* $dom(\Sigma) \cap dom(\Sigma') = \emptyset$ *then* $\Gamma \vdash_{\Sigma,\Sigma'} e : \tau$.

We can now state the preservation and progress theorems for our language. We define them each in two parts, one for the expression language and another for the command language.

Theorem 1 (Preservation)**.**

1. If $e \underset{\Sigma}{\mapsto} e'$ and $\emptyset \vdash_\Sigma e : \tau$, then $\emptyset \vdash_\Sigma e' : \tau$.
2. If $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ and $\emptyset \vdash_\Sigma m \sim \tau$ and $\mu : \Sigma$ then $\emptyset \vdash_\Sigma m' \sim \tau$ and $\mu' : \Sigma$.

Task 1.1 (15%). Even though expressions and commands intermingle, we can first prove preservation for expressions (part 1) and then use that as a lemma when we prove preservation for commands (part 2).

Prove the second part of preservation theorem by rule induction over the dynamics of commands. Be explicit about what the induction hypothesis $\mathcal{P}(m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu')$ is. Explicitly state (do not prove) any inversion lemmas you need aside from the ones listed above. You only need to show the cases corresponding to $\mathtt{get}[a]$ and $\mathtt{dcl}\ a := e\ \mathtt{in}\ m$, rules (17), (20), (21), and (22). □

Theorem 2 (Progress)**.**

1. If $\emptyset \vdash_\Sigma e : \tau$, then either $e\ \mathsf{val}_\Sigma$ or there exists e' such that $e \underset{\Sigma}{\longmapsto} e'$.
2. If $\emptyset \vdash_\Sigma m \sim \tau$ and $\mu : \Sigma$ then either
   - $m \parallel \mu\ \mathsf{final}_\Sigma$,
   - $m \parallel \mu\ \mathsf{err}_\Sigma$,
   - or there exists m', μ' such that $m \parallel \mu \underset{\Sigma}{\longmapsto} m' \parallel \mu'$.

Task 1.2 (15%). The progress theorem is proved by mutual rule induction on the statics of expressions and commands. Be explicit about what your two induction hypotheses $\mathcal{P}(\Gamma \vdash_\Sigma e : \tau)$ and $\mathcal{P}(\Gamma \vdash_\Sigma m \sim \tau)$ are, and explicitly state (do not prove) any canonical forms lemmas you need aside from the ones listed above. You only need to show the cases corresponding to the rules (`bnd`-M) and (`castl`-M).

Hint: Usually the induction hypothesis is very similar to the theorem statement, and this is true for the relationship between Part 2 of the progress theorem and $\mathcal{P}(\Gamma \vdash_\Sigma m \sim \tau)$. However, in order to carefully establish the `bnd` case, $\mathcal{P}(\Gamma \vdash_\Sigma e : \tau)$ needs to be changed quite a bit.

We suggest letting $\mathcal{P}(\Gamma \vdash_\Sigma e : \tau)$ be "If $\Gamma = \emptyset$, then 1) either e val or there exists an e' such that $e \underset{\Sigma}{\longmapsto} e'$, and 2) for all m and μ, if $e = \mathtt{cmd}(m)$ and $\mu : \Sigma$, either $m \parallel \mu\ \mathsf{final}_\Sigma$, or $m \parallel \mu\ \mathsf{err}_\Sigma$, or there exist m' and μ' such that $m \parallel \mu \underset{\Sigma}{\longmapsto} m' \parallel \mu'$." This means that there's slightly more work to do in the `cmd`-I case (which you don't have to prove), but it makes the `bnd`-M case (which you do). You should make sure you understand how the `cmd`-I case works, and why the obvious candidate for $\mathcal{P}(\Gamma \vdash_\Sigma e : \tau)$ does not. □

Task 1.3 (5%). It seems as if the command for cast could be written as $m ::= e \mathbin{@} \mathtt{l} \mid e \mathbin{@} \mathtt{r}$, making casts look more like the casts in Hybrid PCF. Such cast commands would have the following statics and dynamics:

$$\frac{\Gamma \vdash_\Sigma e : \tau_1 + \tau_2}{\Gamma \vdash_\Sigma e \mathbin{@} \mathtt{l} \sim \tau_1} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 + \tau_2}{\Gamma \vdash_\Sigma e \mathbin{@} \mathtt{r} \sim \tau_2}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{e \mathbin{@} \mathtt{l} \parallel \mu \underset{\Sigma}{\longmapsto} e' \mathbin{@} \mathtt{l} \parallel \mu} \qquad \frac{\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e)\ \mathsf{val}_\Sigma}{\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e) \mathbin{@} \mathtt{l} \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{ret}\ e \parallel \mu} \qquad \frac{\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e)\ \mathsf{val}_\Sigma}{\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e) \mathbin{@} \mathtt{l} \parallel \mu\ \mathsf{err}_\Sigma}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{e \mathbin{@} \mathtt{r} \parallel \mu \underset{\Sigma}{\longmapsto} e' \mathbin{@} \mathtt{r} \parallel \mu} \qquad \frac{\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e)\ \mathsf{val}_\Sigma}{\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e) \mathbin{@} \mathtt{r} \parallel \mu \underset{\Sigma}{\longmapsto} \mathtt{ret}\ e \parallel \mu} \qquad \frac{\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e)\ \mathsf{val}_\Sigma}{\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e) \mathbin{@} \mathtt{r} \parallel \mu\ \mathsf{err}_\Sigma}$$

Explain why this isn't actually a good idea by given a counterexample to either progress and preservation using these semantics. (Hint: it would be possible to fix these rules, but in doing so we'd make casting useless for dealing with the arrays we introduce in Section 3.1.) □

1.5 Implementation

The ABT infrastructure we've been working with so far needs to be generalized quite a bit in order to handle Algol. Previously, we have associated the operator `let` with the arity $[0, 1]$ in our Standard ML

implementation, which means that `let` is an operator that forms an expression by taking two arguments, the first binding no variables and the second binding a single expression variable. Now we have two mutually recursive sorts, expressions e and commands m, which include both bound expression variables x and bound assignables a, so the way we've been describing arities so far is insufficient for clearly explaining what is going on in Algol.

One way we could deal with this is to extend the ABT library significantly to handle more complex arities and multiple sorts. For this assignment, we're going to experiment with a different way of handling this. Just like we have lexer and parser generators that turn syntax specifications into lexers and parsers, we'll use an ABT generator that takes an ABT specification and generates code for correctly creating fresh variables and performing substitution.

A partial specification for Algol is below, and Figure 2 shows how commands in this specification are compiled by the ABT generation tool. Note that we don't have to deal with abstract binding sites `\(x,e)` in this representation; the abstract binding sites associated with `bnd` and `dcl` are automatically expanded when we pattern match against them.

```
symbol assign

abt exp
 = Z
 | S (exp)
 ...
 | Cmd (cmd)

abt cmd
 = Bnd (exp, exp.cmd)
 | Ret (exp)
 | Dcl (exp, assign.cmd)
 | Get (assign)
 | Set (assign, exp)
 ...
```

The full specification is in the code handout as `syn/algol.abt`, and the SML interface is in the code handout `syn/algol.abt.user.sml`. This file includes several constructs that you *won't* implement static and dynamic semantics for, as their semantics will be defined by *elaboration* in the next section's tasks.

Task 1.4 (25%). Implement the following structures. Your code doesn't have to behave reasonably if it is given things like while loops that will be defined by elaboration in the next section.

- `TypeChecker` satisfying the signature `TYPECHECKER_ALGOL`. The typecheck function should assign types to both expressions and commands, according to the corresponding statics.
- `Dynamics` satisfying the signature `DYNAMICS_ALGOL`. The `trystep_exp` function should implement the dynamics of expressions; a re-implementation of the portion of this assignment that you've already done is provided. The `trystep_cmd` function should implement the dynamics of commands.

```
signature CMD =
sig
   type exp (* = Exp.t *)
   type expVar (* = Exp.Var.t *)
   type cmd (* = Cmd.t *)
   type t = cmd

   datatype ('exp, 'cmd) cmdView
    = Ret of 'exp
    | Bnd of 'exp * (expVar * 'cmd)
    | Dcl of 'exp * (Assign.t * 'cmd)
    ...

   val Ret': exp -> cmd
   val Bnd': exp * (expVar * cmd) -> cmd
   val Dcl': exp * (Assign.t * cmd) -> cmd
   ...

   val into:  (exp, cmd) cmdView -> cmd
   val out: cmd -> (exp, cmd) cmdView
   val aequiv: cmd * cmd -> bool
   val toString: cmd -> string
   val substExp: exp -> expVar -> cmd -> cmd
end
structure Cmd: CMD ... = ...
```

Figure 2: Output of the ABT generation tool (commands)

2 Elaboration

While the basic syntax of Modernized Algol is expressive enough for a variety of tasks, it can be a tad tedious to express some common idioms. One way to ameliorate the situation would be to add more compact forms for these idioms into the language directly. This could also allow us to write optimized implementations for these constructs. However, this bloats the language and creates more cases that we have to involve in our proofs. Another approach is to add a layer of interpretation between the core language and a more convenient surface syntax. This technique, used in many languages, is known as *elaboration*.

For each of the commands described in this section, you should extend the function `elaborate_cmd` in `elaborator.sml`. Elaboration is applied to the parsed syntax before it is sent to the typechecking or evaluation. The simplest elaborator is the identity function. To get you started, we've gone ahead and implemented the expression $\texttt{proc}[\tau](x.m)$ (concrete syntax $\texttt{proc}\,(x{:}\tau)\,m$), for creating procedures of type $\tau \rightharpoonup \texttt{cmd}(\tau')$, and the command $\texttt{call}(e_1; e_2)$ (concrete syntax $e_1\,(e_2)$) for calling a such procedures.

$$\frac{\Gamma, x : \tau \vdash_\Sigma m \sim \tau'}{\Gamma \vdash_\Sigma \texttt{proc}[\tau](x.m) : \tau \rightharpoonup \texttt{cmd}(\tau')} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \rightharpoonup \texttt{cmd}(\tau_2) \qquad \Gamma \vdash_\Sigma e_2 : \tau_1}{\Gamma \vdash_\Sigma \texttt{call}(e_1; e_2) \sim \tau_2}$$

Task 2.1 (10%). Some basic Algol idioms can be defined by elaboration:

Sort			Abstract Syntax	Concrete Syntax	Description
Cmd	m	$::=$	$\texttt{seq}(m_1; x.m_2)$	$\{x \leftarrow m_1; m_2\}$	Executes command m_1, then passes its return value to m_2 via the variable x.
			$\texttt{do}(e)$	$\texttt{do}(e)$	Executes an encapsulated command, e.
			$\texttt{letm}(e; x.m)$	$\texttt{letm}\ x = e\ \texttt{in}\ m$	Substitutes the value of e for x in m.
			$\texttt{if}(m_1; m_2; m_3)$	$\texttt{if}\ (m_1)\ m_2\ \texttt{else}\ m_3$	If m_1 is true, executes m_2 otherwise m_3.
			$\texttt{while}(m_1; m_2)$	$\texttt{while}(m_1)\ m_2$	Runs m_2 repeatedly, until m_1 returns false.

The `seq` and `do` commands are the same as the ones from *PFPL* chapter 36, but the `if` and `while` commands should work on Booleans as on the midterm, where values of type $\texttt{unit} + \texttt{unit}$, where $\texttt{inl}[\texttt{unit}; \texttt{unit}](\langle\rangle)$ stands for true and $\texttt{inr}[\texttt{unit}; \texttt{unit}](\langle\rangle)$ stands for false. The `letm` command just repeats the behavior of `let` but for commands. These constructs should meet the following type specification: □

$$\frac{\Gamma \vdash_\Sigma m_1 \sim \tau \qquad \Gamma, x : \tau \vdash_\Sigma m_2 \sim \tau'}{\Gamma \vdash_\Sigma \texttt{seq}(m_1; x.m_2) \sim \tau'} \qquad \frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\tau) \qquad \tau\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \texttt{do}(e) \sim \tau}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad \Gamma, x : \tau \vdash_\Sigma m \sim \tau'}{\Gamma \vdash_\Sigma \texttt{letm}(e; x.m) \sim \tau'} \qquad \frac{\Gamma \vdash_\Sigma m \sim \texttt{unit} + \texttt{unit} \qquad \Gamma \vdash_\Sigma m_t \sim \tau \qquad \Gamma \vdash_\Sigma m_f \sim \tau}{\Gamma \vdash_\Sigma \texttt{if}(m; m_t; m_f) \sim \tau}$$

$$\frac{\Gamma \vdash_\Sigma m_1 \sim \texttt{unit} + \texttt{unit} \qquad \Gamma \vdash_\Sigma m_2 \sim \tau}{\Gamma \vdash_\Sigma \texttt{while}(m_1; m_2) \sim \texttt{unit}}$$

3 Programming in Algol

We will finish off by programming a few imperative algorithms in Algol in the file `sort.alg`.

3.1 Arrays

Arrays in Algol can be (inefficiently) defined as lists of references: that is, we can define the type $\texttt{array}(\tau)$ as the recursive type $\mu t.\texttt{unit} + (\texttt{ref}(\tau) \times t)$ where τ is mobile. As in most languages, out-of-bounds array access should be a runtime error.

Task 3.1 (6%). Implement the procedures `aget` and `aset` that implement reading and writing from an array, as well as a procedure `tolist` that returns a regular (and mobile!) list of nats when given an array. □

Task 3.2 (6%). Going from arrays to lists is relatively straightforward, but the other direction is more interesting. The procedure `dclarray n C` creates a new zero-initialized array `A` of length `n` and then runs the command `C(A)` with this array is in scope. You will need to write a helper function to implement this procedure. □

Task 3.3 (12%). Complete an implementation of in-place sorting in Algol by implementing the two functions `swap`, which does an in-place swap of array elements, and `sort`, which does an in-place sort of array elements. Selection sort (with a `findmin` helper function) is probably the easiest option, but you can implement any sort you'd like to use.

As an exception to the usual course policy on outside sources, you're welcome to use sorting algorithm descriptions from Wikipedia or other CMU classes as a reference. I used `http://www.cs.cmu.edu/~rjsimmon/15122-f13/lec/07-sorting/selectsort.c0` as the reference for my selection sort implementation. □

3.2 Choose your own adventure

Task 3.4 (6%). Write `demo.alg`, an Algol program that does something interesting. Here are two example ideas that would be sufficiently interesting to receive full credit:

- Implement a queue as two lists (an *in* list and an *out* list: if the *out* list is empty and you dequeue, then the *in* list gets flipped over onto the *out* list. The type of your queue will either be $\texttt{ref}(\texttt{list}(\tau) \times \texttt{list}(\tau))$ or $\texttt{ref}(\texttt{list}(\tau)) \times \texttt{ref}(\texttt{list}(\tau))$
- Implement heaps (priority queues) as Algol arrays.
- Implement a hash table as an array of lists. (Not linked lists: it is not possible to write pointer structures create obeying the stack discipline.)

Please comment your code – the parser implementation supports `(* comments like this *)`. If you implement any interesting ideas that are different from the ones above, you are encouraged to share your interesting Algol programs on Piazza. It is fine if your demo program is inspired by someone else's, but please note that in a comment. □

A Putting The Code Together

As usual, you can compile your files using `CM.make "sources.cm"`. We have provided two ways to test the final implementation: an interpreter and a reference implementation.

A.1 Interpreter

As usual, to run the interpreter, execute `TopLevel.repl();`. The REPL now loads files with a more SML-like `use` command that takes a quoted string.

The syntax for each term construct is as close as possible to the concrete syntax mentioned for it. (The concrete syntax is the second column in the table which introduces the syntax for the language.) We provide below the grammar that the interpreter accepts, as well as a sample session of the interpreter.

```
ident ::= (* a letter followed by alphanumeric characters, _, or ' *)
numeral ::= (* a series of digits *)
strconst ::= (* a standard C-escaped string like "foo" or "why \"hello\"" *)

command ::=
  step <exp>;
| step;
| eval <exp>;
| eval;
| use <stringconst>;
```

```
ty ::= <ident> | nat | <ty> -> <ty> | unit | <ty> * <ty> | void | <ty> + <ty>
| mu <ident>.<ty> | cmd <ty> | ref <ty> | (<ty>)

exp ::=
  <ident>
| z
| s <exp>
| ifz <exp> { z => <exp> | s <ident> => <exp> }
| let <edecls> in <exp> end
| fn (<ident> : <ty>) <exp>
| proc (<ident> : <ty>) <cmd>
| <exp>(<exp>)
| fix <ident> : <ty> is <exp>
| abort[<ty>] <exp>
| inl[<ty>,<ty>] <exp>
| inr[<ty>,<ty>] <exp>
| case e { inl <ident> => exp | inr <ident> => exp }
| <>
| <<exp>, <exp>>
| <exp>.l
| <exp>.r
| fold[<ident>.<ty>] <exp>
| unfold <exp>
| cmd <cmd>
| &<ident>
| (<exp>)

edecls ::= <edecl> | <edecl> <edecls>

edecl ::= val <ident> = <exp>

cmd ::=
  ret <exp>
| bnd <ident> <- <exp>, <cmd>
| dcl <ident> := <exp> in <cmd>
| @<ident>                          (* Parses as get *)
| <ident> := <exp>                  (* Parses as set only if <ident> is
                                     * an assignable currently in scope *)
| *<ident>                          (* Parses as getref *)
| <exp> := <exp>                    (* Parses as setref *)
| let <mdecls> in <cmd> end
| <exp>(<exp>)                      (* Parses as call *)
| do <exp>
| if <cmd> <cmd> else <cmd>
| while <cmd> <cmd>
| {<curlycmd>}
| (<cmd>)

mdecls ::= <mdecl> | <mdecl> <mdecls>

mdecl ::=
```

```
  val <ident> = <exp>              (* Parses as letm *)
| val inl <ident> = <exp>          (* Parses as cast *)
| val inr <ident> = <exp>          (* Parses as cast *)

curlycmd ::=
  <cmd>
| <ident> = <exp>, <curlycmd>      (* Parses as letm *)
| inl <ident> = <exp>, <curlycmd>  (* Parses as cast *)
| inr <ident> = <exp>, <curlycmd>  (* Parses as cast *)
| <ident> <- <cmd>, <curlycmd>     (* Parses as seq *)
| <cmd>, <curlycmd>                (* Parses as seq *)
```

Here is an example of running the interpreter:

```
$ rlwrap sml -m sources.cm
- TopLevel.repl();
->use "sort.alg";
Statics : cmd has type Nat
 --> Bnd(Let(In(Unit, Unit, L, Triv), tt@457.Cmd(Bnd(Let(In(Unit, Unit, R, ...
->eval;
Statics : cmd has type Nat
 S(S(S(Z))) VAL
->eval bnd x <- (ifz z { z => cmd (ret 4) | s _ => cmd (ret 5) }), ret (s x);
Statics : cmd has type Nat
 S(S(S(S(Z)))) VAL
->step dcl a := 4 in { x <- @a, y = s(s(s x)), a := s y, ret y };
Statics : cmd has type Nat
Statics : cmd has type Nat
 --> Dcl(S(S(S(S(Z)))), a@25508.Bnd(Cmd(Ret(S(S(S(S(Z))))))), x@184048.Bnd(L...
->step;
Statics : cmd has type Nat
 --> Dcl(S(S(S(S(Z)))), a@25511.Bnd(Let(S(S(S(S(S(S(Z)))))), y@184060.Cm...
->step;
Statics : cmd has type Nat
 --> Dcl(S(S(S(S(Z)))), a@25514.Bnd(Cmd(Bnd(Cmd(Set(a@25514, S(S(S(S(S(S(...
->step;
Statics : cmd has type Nat
 --> Dcl(S(S(S(S(S(S(S(Z))))))), a@25517.Bnd(Cmd(Bnd(Cmd(Ret(S(S(S(S(S(S...
->eval;
Statics : cmd has type Nat
 S(S(S(S(S(S(Z))))))) VAL
->eval if (ret (inl[unit,unit] <>)) ret z else ret s z;
Statics : cmd has type Nat
 Z VAL
```

A.2 Reference implementation

You can use the reference interpreter to test your Algol code and help debug your interpreter. The reference implementation directly implements the elaborated constructs (`seq`, `do`, etc.) directly rather than defining them by elaboration. *Do not do this in your implementation.*

The reference interpreter for Algol can be loaded like this:

```
$ rlwrap sml @SMLload=/afs/andrew/course/15/312/bin/ref_impl_algol
- TopLevel.repl ();
```