Homework 1: Abstract Syntax and Rule Definitions

15-814: Types and Programming Languages
Fall 2015
TA: Evan Cavallo (ecavallo@cs.cmu.edu)

Out: 9/10/15
Due: 9/24/15

Notes:

- **Correction 9/15/15:** Fixed the treatment of contexts and added $\Delta \vdash e\ \mathsf{val}_\mathsf{e}$ requirement to canonical forms lemma.
- Welcome to 15-814's first homework assignment!
- Please submit your work as a PDF file to `ecavallo@cs.cmu.edu` and include the phrase "15-814 Homework 1" in the subject line of your email.
- We will be using Piazza to answer questions as well as make clarifications, corrections and announcements. Please sign up for the class on Piazza to make sure you don't miss anything: `piazza.com/cmu/fall2015/15814`

In this homework, we will examine a simple C-like language from the perspective of abstract syntax and type theory.

1 Arithmetic

We'll start with a simple expression language similar to that covered in class. There are two types, integers `int` and booleans `bool`. The language consists of literals, a few arithmetic and comparison operators, and a conditional expression.

$$\begin{array}{lcl} \tau & ::= & \texttt{int} \mid \texttt{bool} \\ e & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid e + e \mid e * e \mid e \leq e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \end{array}$$

Since this language has no variable-binding constructs, translating the concrete presentation into abstract syntax is quite simple.

concrete	abstract
n	$\texttt{num}[n]$
`true`	`true`
`false`	`false`
$e_1 + e_2$	$\texttt{plus}(e_1; e_2)$
$e_1 * e_2$	$\texttt{times}(e_1; e_2)$
$e_1 \leq e_2$	$\texttt{leq}(e_1; e_2)$
$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	$\texttt{if}(e_1; e_2; e_3)$

We define the typing judgment $\Gamma \vdash e :_\mathsf{e} \tau$ and operational semantics judgments $\Delta \vdash e\ \mathsf{val}_\mathsf{e} e'$ and $\Delta \vdash e \mapsto_\mathsf{e} e'$ in Appendix A. Type contexts Γ and evaluation contexts Δ are defined with with the following grammar:

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\ \Delta & ::= & \cdot \end{array}$$

For the moment, the only evaluation context is the empty context $\cdot$, but we will change this as we build on the language.

Before we add anything new, however, let's check some properties to make sure you're comfortable with rule induction. For the proofs in this assignment, you may condense similar cases in arguments by induction, but be clear and rigorous in your reasoning.

Task 1 *Prove the following* inversion lemma*:*

> *(If Inversion) If* $\Gamma \vdash \texttt{if}(e_1; e_2; e_3) :_\texttt{e} \tau$*, then* $\Gamma \vdash e_1 :_\texttt{e} \texttt{bool}$*,* $\Gamma \vdash e_2 :_\texttt{e} \tau$*, and* $\Gamma \vdash e_3 :_\texttt{e} \tau$*.*

This seems immediate, but really follows from the induction principle for the typing judgment. (Tip: Prove that for all Γ*,* e*,* τ *such that* $\Gamma \vdash e :_\texttt{e} \tau$*, if* $e = \texttt{if}(e_1; e_2; e_3)$ *for some* e_1*,* e_2*,* e_3*, then* $\Gamma \vdash e_1 :_\texttt{e} \texttt{bool}$*,* $\Gamma \vdash e_2 :_\texttt{e} \tau$*, and* $\Gamma \vdash e_3 :_\texttt{e} \tau$*.)*

In general, an inversion lemma is one which recovers the premises of a rule from its conclusion. In the rest of the assignment, you may use these without proof, but be sure to note explicitly when you apply them and check for yourself that they hold.

Task 2 *Prove unicity of typing for this language.*

> (Unicity of Typing) *For any* Γ*,* e*,* τ*,* τ' *such that* $\Gamma \vdash e :_\texttt{e} \tau$ *and* $\Gamma \vdash e :_\texttt{e} \tau'$*, we have* $\tau = \tau'$*. You may assume that any variable appears at most once in a given context.*

2 Functions

2.1 Function Blocks

On top of the expression language, we now add a syntax for programs p, which consist of a series of one-argument function definitions and a final expression computation. (In such a small language, it would be useful to have multi-argument functions, but we will restrict ourselves for the sake of simplicity.) In order to make use of these definitions, we add syntax for function calls to the expression language. The sort of functions consists only of function variables, which we denote here with the letter u.

$$\begin{array}{lcl} f & ::= & u \\ e & ::= & \cdots \mid f(e) \\ p & ::= & \texttt{fun}\ u\ (x : \tau)\ \{e\};\ p \mid \texttt{result}\ e \end{array}$$

We allow calls to previously defined functions to appear in the definitions of later functions as well as in the result expression. As an example, the program

$$\begin{array}{l} \texttt{fun}\ \mathit{not}\ (b : \texttt{bool})\ \{\texttt{if}\ b\ \texttt{then false else true}\}; \\ \texttt{fun}\ \mathit{abs}\ (x : \texttt{int})\ \{\texttt{if}\ \mathit{not}(0 \leq x)\ \texttt{then}\ -1 * x\ \texttt{else}\ x\}; \\ \texttt{fun}\ \mathit{square}\ (z : \texttt{int})\ \{z * z\}; \\ \texttt{fun}\ \mathit{cube}\ (z : \texttt{int})\ \{\mathit{square}(z) * z\}; \\ \texttt{result}\ \mathit{cube}(\mathit{abs}(3)) + \mathit{cube}(\mathit{abs}(-2)) \end{array}$$

should compute to 35.

Task 3 *Give a definition in abstract syntax for the language constructs described above. Include the arities (with sorts) of the operators, per Section 1.2 of PFPL.*

Before giving the precise operational semantics of this language, we have an ambiguity to settle. Consider the following program.

$$\begin{array}{l} \texttt{fun}\ u\ (x : \texttt{int})\ \{x\}; \\ \texttt{fun}\ v\ (y : \texttt{int})\ \{u(y)\}; \\ \texttt{fun}\ u\ (z : \texttt{int})\ \{z + 1\}; \\ \texttt{result}\ v(1) \end{array}$$

The result expression $v(1)$ will be computed in an environment of function definitions. Executing the call $v(1)$ looks up the definition of v, which then leads to a call to u. But which of the two functions named u does this call refer to? Depending on how the environment behaves, it may be the first or the second, and so this program might compute either 1 or 2. The first evaluation strategy, in which the u that v calls remains the u available when v was defined, is called *static scoping* (sometimes *lexical scoping*). The second, wherein the call to u in v refers to whichever u was most recently added to the environment, is called *dynamic scoping*.

Task 4 *We want to respect the* identification convention*:*

> *Abstract binding trees are always identified up to α-equivalence.*

With this in mind, which of the two strategies is correct? Explain.

Now, let's move on to defining the statics and dynamics of this language precisely. We will need thwo typing judgments to determine if programs are well-typed. Besides the judgment $\Gamma \vdash e :_{\mathsf{e}} \tau$ for typing expressions we have already mentioned, there should be a judgment $\Gamma \vdash p :_{\mathsf{p}} \tau$ which states that a program has result of type τ. We also need to add a new kind of hypothesis to our type contexts:

$$\Gamma \quad ::= \quad \cdots \mid u :_{\mathsf{f}} \tau_1 \Rightarrow \tau_2$$

The assumption $u :_{\mathsf{f}} \tau_1 \Rightarrow \tau_2$ is used to denote that u is a function variable in scope which takes an argument of type τ_1 and returns a result of type τ_2.

Task 5 *Starting with the rules in Appendix A, complete the definition of the typing judgment $\Gamma \vdash e :_{\mathsf{e}} \tau$, and define the judgement $\Gamma \vdash p :_{\mathsf{p}} \tau$.*

Task 6 *Give a structural operational semantics for this language by completing the definition of $\Delta \vdash e \mapsto_{\mathsf{e}} e'$ and defining a judgment $\Delta \vdash p \mapsto_{\mathsf{p}} p'$. Assume that the value judgment $\Delta \vdash p\ \mathsf{val}_{\mathsf{p}}$ is defined by the single rule*

$$\frac{\Delta \vdash e\ \mathsf{val}_{\mathsf{e}}}{\Delta \vdash (\texttt{result}\ e)\ \mathsf{val}_{\mathsf{p}}}\ (\textsc{Res-V})$$

In order to track the environment of function definitions, we will add a new kind of assumption to our evaluation context:

$$\Delta \quad ::= \quad \cdots \mid u \stackrel{\mathsf{def}}{=} x.e$$

The assumption $u \stackrel{\mathsf{def}}{=} x.e$ expresses that the function variable u is currently in scope and that it takes an argument x and computes the expression e.

Your definition should of the semantics should satisfy the properties of progress *and* preservation*.*

> (Progress) *If $\vdash p :_{\mathsf{p}} \tau$, then either $\vdash p\ \mathsf{val}_{\mathsf{p}}$ or there exists p' such that $\vdash p \mapsto_{\mathsf{p}} p'$.*
> (Preservation) *If $\vdash p :_{\mathsf{p}} \tau$ and $\vdash p \mapsto_{\mathsf{p}} p'$, then $\vdash p' :_{\mathsf{p}} \tau$.*

Task 7 *Prove progress for the rules you have specified. You will want to state and prove an analogous theorem for expressions first. For programs, it may be helpful to prove the following more general theorem:*

> *If $\Delta ::_{\mathsf{f}} \Gamma$ and $\Gamma \vdash p :_{\mathsf{p}} \tau$, then either $\Delta \vdash p\ \mathsf{val}_{\mathsf{p}}$ or there exists p' such that $\Delta \vdash p \mapsto_{\mathsf{p}} p'$.*

Here $\Delta ::_{\mathsf{f}} \Gamma$ *is a judgment on contexts defined by the following rules:*

$$\frac{}{\cdot ::_{\mathsf{f}} \cdot}\ (\textsc{Env-Nil}) \qquad \frac{\Delta ::_{\mathsf{f}} \Gamma}{(\Delta, f \stackrel{\mathsf{def}}{=} x.e) ::_{\mathsf{f}} (\Gamma, f :_{\mathsf{f}} \tau_1 \Rightarrow \tau_2)}\ (\textsc{Env-Cons})$$

This judgment expresses that the environment Δ *and type context* Γ *reference the same function variables, i.e. that they agree on what is in scope. This condition is enough to guarantee a canonical forms lemma, which you will need for your proof:*

> (Canonical Forms Lemma) *Let contexts* $\Delta ::_{\mathsf{f}} \Gamma$ *and an expression* e *be given. Assume that* $\Delta \vdash e\ \mathsf{val}_{\mathsf{e}}$*. If* $\Gamma \vdash e :_{\mathsf{e}} \texttt{int}$*, then* $e = \texttt{num}[n]$ *for some* $n \in \mathbb{Z}$*. If* $\Gamma \vdash e :_{\mathsf{e}} \texttt{bool}$*, then either* $e = \texttt{true}$ *or* $e = \texttt{false}$*.*

This lemma enumerates the forms that well-typed values can take (the eponymous canonical forms*). You are not required to prove this lemma.*

We can make this language a bit more useful by allowing function definitions to refer recursively to the functions they define. We can then write, for example, the factorial function:

$$\texttt{fun}\ \mathit{fact}\ (x : \texttt{int})\ \{\texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ x * \mathit{fact}(x + (-1))\};$$
$$\texttt{result}\ \mathit{fact}(5)$$

Task 8 *How does this extension change your answers to Tasks 3, 5, and 6? Give the syntax and rules where different. (You should also make a change to the form of the assumption* $u \stackrel{\mathsf{def}}{=} x.e$*.)*

2.2 Functions as Expressions

While we can write programs with a predefined list of functions in this language, there is no way of manipulating functions within expressions. For example, we cannot write a function which takes a function as an argument or one which returns a function. One way of solving this is by adding an operator which makes an expression of a function name, in the style of C's function pointers.

$$e \quad := \quad \cdots \mid \texttt{fn}(f) \mid {}^*e(e)$$

We add a corresponding type of functions which has expressions $\texttt{fn}(f)$ as its canonical forms.

$$\tau \quad := \quad \cdots \mid \tau \Rightarrow \tau$$

Now it is possible to write functions which act on functions. For example, the following program should return 9.

$$\texttt{fun}\ \mathit{fact}\ (x : \texttt{int})\ \{\texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ x * \mathit{fact}(x + (-1))\};$$
$$\texttt{fun}\ \mathit{onetothree}\ (y : \texttt{int} \Rightarrow \texttt{int})\ \{{}^*y(1) + {}^*y(2) + {}^*y(3)\};$$
$$\texttt{result}\ \mathit{onetothree}(\texttt{fn}(\mathit{fact}))$$

Naively, we might extend our language to accommodate these new constructs by adding new statics and dynamics rules for the $\texttt{fn}(f)$ and ${}^*e(e)$. However, simply doing this will break either progress or preservation.

Task 9 *Which property (progress or preservation) fails? Why? Describe two solutions to the problem, one by changing the rules for the judgments* $\Delta \vdash p\ \mathsf{val}_{\mathsf{p}}$ *and* $\Delta \vdash p \mapsto_{\mathsf{p}} p'$ *and one by changing the rules for the judgment* $\Delta \vdash p :_{\mathsf{p}} \tau$*.*

These issues, as well as the ambiguity discussed in Task 4, arise essentially because we have separated functions and expressions into distinct sorts. Because of this, we cannot substitute function definitions for function names until we evaluate a particular function call. As such, we have to maintain an environment of function definitions to substitute at time-of-call. If we instead have first-class functions, constructible within the expression language, the situation is much simpler.

Task 10 *Return to the core expression language. We add a new, distinct function type which contains recursive function expressions.*

$$\tau ::= \cdots \mid \tau \to \tau$$

Define abstract syntax for operators for constructing and applying recursive function terms. (In other words, define a version of λ-abstraction and application which allows for self-reference.) Give statics and dynamics rules for these operators, assuming that we have only the expression language and no outer layer of function definitions.

3 Structs

In this section, we will add C-style structs to our language. As an example, take the following C definition of a type of binary trees, which are intended to be labeled with integers at the leaves. (If you are not familiar with C syntax, you can find simple tutorials online or ask Evan to clarify.)

```
typedef struct tree_data tree;

struct tree_data {
  int leaf_value;
  tree *left;
  tree *right;
};
```

By convention, either both `left` and `right` are null, in which case the tree is a leaf tagged with the value in `leaf_value`, or neither is null, in which case the tree is a node and `leaf_value` is ignored.

To add these sort of data structures to our language, we'll decompose the problem into three parts. First, we need a way to build types with multiple fields. Second, we need types that have a "null" value. Finally, we need types that can refer to themselves recursively.

The first two are fairly easily solved by adding product and sum types to the language.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \tau \times \tau \mid \texttt{unit} \mid \tau + \tau \\ e & ::= & \cdots \mid \langle e, e \rangle \mid \langle\rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{in}[l](e) \mid \texttt{in}[r](e) \mid \texttt{case}\ e\ \{x.e; x.e\} \end{array}$$

A canonical element of the product type $\tau_1 \times \tau_2$ is a pair $\langle e_1, e_2 \rangle$ where $e_1 :_\texttt{e} \tau_1$ and $e_2 :_\texttt{e} \tau_2$; we can represent a struct type with multiple fields by chaining products. The unit type `unit` has a single canonical element $\langle\rangle$, and can be used to represent an empty struct. We think of `unit` as a nullary product, a tuple with zero fields. The canonical elements of the sum type $\tau_1 + \tau_2$ are either of the form $\texttt{in}[l](e_1)$ with $e_1 :_\texttt{e} \tau_1$ or $\texttt{in}[r](e_2)$ with $e_2 :_\texttt{e} \tau_2$. We may therefore represent a possibly-null field of type τ with the type $\tau + \texttt{unit}$.

Adding recursively defined types is more complicated, but we can approach it in much the same way that we approached recursive function blocks in the previous section. We add type variables to the type grammar, and type declarations to the program grammar:

$$\begin{array}{lcl} \tau & ::= & \cdots \mid t \\ p & ::= & \cdots \mid \texttt{type}\ t = \tau; p \end{array}$$

In the type declaration above, t may appear in τ. Much as we track an environment of function definitions for evaluation, we can use a context of type definitions for typechecking in the presence of these declarations. We won't go into the details in this homework, as we'll be covering recursively-defined types later in the course.

Task 11 *Give a type declaration in this language which corresponds to the C type* `tree` *defined above. This definition should match the C struct's behavior exactly.*

Task 12 *The C struct definition of* `tree` *assumes that the programmer follows the conventions described above, and it is possible to construct a* `tree` *which is well-typed but corresponds to no legal tree. Give an alternate definition of* `tree` *in our language which statically enforces the conventions.*

A Base Expression Language

A.1 Statics

$$\frac{}{\Gamma, x :_\mathsf{e} \tau \vdash x :_\mathsf{e} \tau}\ (\textsc{Hyp}) \qquad \frac{}{\Gamma \vdash \texttt{num}[n] :_\mathsf{e} \texttt{int}}\ (\textsc{Num}) \qquad \frac{}{\Gamma \vdash \texttt{true} :_\mathsf{e} \texttt{bool}}\ (\textsc{True})$$

$$\frac{}{\Gamma \vdash \texttt{false} :_\mathsf{e} \texttt{bool}}\ (\textsc{False}) \qquad \frac{\Gamma \vdash e_1 :_\mathsf{e} \texttt{int} \quad \Gamma \vdash e_2 :_\mathsf{e} \texttt{int}}{\Gamma \vdash \texttt{plus}(e_1; e_2) :_\mathsf{e} \texttt{int}}\ (\textsc{Plus})$$

$$\frac{\Gamma \vdash e_1 :_\mathsf{e} \texttt{int} \quad \Gamma \vdash e_2 :_\mathsf{e} \texttt{int}}{\Gamma \vdash \texttt{times}(e_1; e_2) :_\mathsf{e} \texttt{int}}\ (\textsc{Times}) \qquad \frac{\Gamma \vdash e_1 :_\mathsf{e} \texttt{int} \quad \Gamma \vdash e_2 :_\mathsf{e} \texttt{int}}{\Gamma \vdash \texttt{leq}(e_1; e_2) :_\mathsf{e} \texttt{bool}}\ (\textsc{Leq})$$

$$\frac{\Gamma \vdash e_1 :_\mathsf{e} \texttt{bool} \quad \Gamma \vdash e_2 :_\mathsf{e} \tau \quad \Gamma \vdash e_3 :_\mathsf{e} \tau}{\Gamma \vdash \texttt{if}(e_1; e_2; e_3) :_\mathsf{e} \tau}\ (\textsc{If})$$

A.2 Dynamics

$$\frac{}{\Delta \vdash \texttt{num}[n]\ \mathsf{val_e}}\ (\textsc{Num-V}) \qquad \frac{}{\Delta \vdash \texttt{true}\ \mathsf{val_e}}\ (\textsc{True-V}) \qquad \frac{}{\Delta \vdash \texttt{false}\ \mathsf{val_e}}\ (\textsc{False-V})$$

$$\frac{\Delta \vdash e_1 \mapsto_\mathsf{e} e_1'}{\Delta \vdash \texttt{plus}(e_1; e_2) \mapsto_\mathsf{e} \texttt{plus}(e_1'; e_2)}\ (\textsc{Plus-S1}) \qquad \frac{\Delta \vdash e_1\ \mathsf{val_e} \quad \Delta \vdash e_2 \mapsto_\mathsf{e} e_2'}{\Delta \vdash \texttt{plus}(e_1; e_2) \mapsto_\mathsf{e} \texttt{plus}(e_1; e_2')}\ (\textsc{Plus-S2})$$

$$\frac{}{\Delta \vdash \texttt{plus}(\texttt{num}[n]; \texttt{num}[m]) \mapsto_\mathsf{e} \texttt{num}[n+m]}\ (\textsc{Plus-I}) \qquad \frac{\Delta \vdash e_1 \mapsto_\mathsf{e} e_1'}{\Delta \vdash \texttt{times}(e_1; e_2) \mapsto_\mathsf{e} \texttt{times}(e_1'; e_2)}\ (\textsc{Times-S1})$$

$$\frac{\Delta \vdash e_1\ \mathsf{val_e} \quad \Delta \vdash e_2 \mapsto_\mathsf{e} e_2'}{\Delta \vdash \texttt{times}(e_1; e_2) \mapsto_\mathsf{e} \texttt{times}(e_1; e_2')}\ (\textsc{Times-S2}) \qquad \frac{}{\Delta \vdash \texttt{times}(\texttt{num}[n]; \texttt{num}[m]) \mapsto_\mathsf{e} \texttt{num}[nm]}\ (\textsc{Times-I})$$

$$\frac{\Delta \vdash e_1 \mapsto_\mathsf{e} e_1'}{\Delta \vdash \texttt{leq}(e_1; e_2) \mapsto_\mathsf{e} \texttt{leq}(e_1'; e_2)}\ (\textsc{Leq-S1}) \qquad \frac{\Delta \vdash e_1\ \mathsf{val_e} \quad \Delta \vdash e_2 \mapsto_\mathsf{e} e_2'}{\Delta \vdash \texttt{leq}(e_1; e_2) \mapsto_\mathsf{e} \texttt{leq}(e_1; e_2')}\ (\textsc{Leq-S2})$$

$$\frac{n \leq m}{\Delta \vdash \texttt{leq}(\texttt{num}[n]; \texttt{num}[m]) \mapsto_{\texttt{e}} \texttt{true}} \ (\textsc{Leq}\text{-I1}) \qquad \frac{n > m}{\Delta \vdash \texttt{leq}(\texttt{num}[n]; \texttt{num}[m]) \mapsto_{\texttt{e}} \texttt{false}} \ (\textsc{Leq}\text{-I2})$$

$$\frac{\Delta \vdash e_1 \mapsto_{\texttt{e}} e_1'}{\Delta \vdash \texttt{if}(e_1; e_2; e_3) \mapsto_{\texttt{e}} \texttt{if}(e_1'; e_2; e_3)} \ (\textsc{If}\text{-S}) \qquad \frac{}{\Delta \vdash \texttt{if}(\texttt{true}; e_2; e_3) \mapsto_{\texttt{e}} e_2} \ (\textsc{If}\text{-I1})$$

$$\frac{}{\Delta \vdash \texttt{if}(\texttt{false}; e_2; e_3) \mapsto_{\texttt{e}} e_3} \ (\textsc{If}\text{-I2})$$