Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.820 Foundations of Program Analysis

Problem Set 3

Out: October 10, 2015
Due: October 21, 2015 at 5:00 PM

The largest part of this problem set involves proving a theorem with Coq, like in the first two problems in Problem Set 2.

Please Remember: The problem set is to be handed in before 5 PM on the day it is due. As with Problem Set 2, please turn in *one* `.tgz` or `.zip` file containing all of the files you produce for this problem set. The questions that aren't about writing Coq code should be answered in a PDF or text file based on your preference (and it's possible we could be talked into permitting other formats).

As usual, we expect you to do your own Coq coding. It is important to *think through your definitions and proof strategies* to some extent, *before* starting to write Coq code, or you can easily paint yourself into a corner where you have little intuition about how to proceed. We recommend starting out by thinking individually at a conceptual level about the design of your type system and outline of your proof strategy. However, *collaborating with other students in the class at a conceptual level is encouraged, so long as the PDF or text file you hand in lists your collaborators.* You'll learn the most if you spend some time on your own first, thinking about these issues; but working with a group to settle on a proof structure is more educational than getting wedged somewhere! **You should still type your Coq proof solutions yourself, without someone else looking over your shoulder guiding you. Group collaboration is only meant to help you understand the *informal* structure of a proof and/or figure out particular general tricks for using tactics to implement a proof step that makes sense on paper.**

We encourage you to start the assignment early and not be shy about posting questions on Piazza as they arise! Use your judgment about whether such questions give away too many solution details, so that they should be marked as private.

Problem 1 Understanding the language (10 points)

Consider the following language:

$$e := x \mid new_\tau \mid e; e \mid let\ x = e\ in\ e \mid e.f \mid e.f := e$$

This is a simple model of an imperative language where you can create references of type τ , where τ can only be a mutable record, and you can mutate the fields of those records. Now, in our simple model language, we can assume that garbage collection takes care of reclaiming objects that are no longer necessary, but garbage collection can sometimes be expensive. In particular, there are many situations where an application needs to allocate a lot of short lived objects that are only necessary within some local context, so it would be great if the garbage collector didn't have to bother with those objects; if we could just store them in some kind of local arena that gets automatically deallocated once the program exits that context. For this exercise we are going to introduce such a mechanism into the language. We do this by introducing two new kinds of expressions:

$$e := newL_\tau \mid context\ e$$

The expression *context* e introduces a new execution context, creates a new arena, executes the expression e and returns the value of evaluating e. Any object allocated in e using the special $newL_\tau$ allocator will be allocated in this new arena, and the arena will be deallocated upon exiting the new context. Contexts can be nested, and in that case, local objects allocated with $newL_\tau$ will be associated with the nearest context to the allocation site.

Example 1 *For example, consider the following code.*

$$\begin{array}{l} let\ x = new_\tau \\ in\ context\,(\ \ let\ y = newL_\tau\ in\ \ y.f = new_\tau; x.f = y.f\ \ );x.f \end{array}$$

In the program above, x is assigned a new object, and then inside a new context, we allocate a local object y . We then allocate a new global object and store it in $y.f$, and then copy that reference from $y.f$ to $x.f$. After exiting the context, the local object will have been reclaimed, but $x.f$ still holds the reference to the global object allocated inside the reference.

The semantics of this scheme can be formalized by the following rules. First, the local reduction rules need to be defined in terms of a heap, as well as some additional program state to keep track of contexts. Local reduction rules act on a triple $(h,\ cid,\ oid,\ e)$ where h is the heap, *cid* is an id of the current context, *oid* is an id for the next object, and e is the local expression to be transformed.

$$\begin{array}{l} (h,\ cid,\ oid,\ new_\tau) \to (h : (g\,(oid) \leftarrow \emptyset),\ cid,\ oid + 1,\ g\,(oid)) \\ (h,\ cid,\ oid,\ newL_\tau) \to (h : (loc\,(oid, cid) \leftarrow \emptyset),\ cid,\ oid + 1,\ loc\,(oid,\ cid)) \\ (h,\ cid,\ oid,\ let\ x = v\ in\ e) \to (h,\ cid,\ oid,\ e\,[v/x]) \\ (h,\ cid,\ oid,\ v; e) \to (h,\ cid,\ oid,\ e) \\ (h,\ cid,\ oid,\ v.f) \to (h,\ cid,\ oid,\ h\,[v, f]) \\ (h,\ cid,\ oid,\ v_1.f := v_2) \to (update(h, v_1, f, v_2),\ cid,\ oid,\ v_2) \\ (h,\ cid,\ oid,\ context\ e) \to (h,\ cid + 1,\ oid,\ inside\ e) \\ (h,\ cid,\ oid,\ inside\ v) \to (cleanup(h,\ cid),\ cid - 1,\ oid,\ v) \end{array}$$

A few things to note about the rules. First, addresses generated by new_τ and $newL_\tau$ are different. Global addresses are created through a constructor g and local addresses are wrapped into a *loc* constructor that includes the id of the current context. Allocation also registers the new addresses into the heap. Reading and writing from fields should fail if the addresses are not already registered in the heap. The local rules also introduce a new syntactic element *inside* which indicates that we are currently evaluating inside a context. Once evaluation inside the context completes, we revert back to the previous context and clean up from the heap any local addresses from context *cid*. At this point the notation is still somewhat informal, but we make this formal in the coq file distributed with the assignment.

Contexts are defined recursively by the following grammar:

$$H \ := \ \square \mid H; e \mid \ let \ x = H \ in \ e \mid H.f \mid \ H.f \ := e \mid v.f \ := H \mid \ inside \ H$$

The top level small-step evaluation rule is

$$\frac{(h, cid, \ oid, \ e) \rightarrow (h^{'}, cid^{'}, oid^{'}, e^{'})}{(h, \ cid, \ oid, \ H\,[e]) \rightarrow (h^{'}, \ cid^{'}, oid^{'}, \ H\,[e^{'}])}$$

Part a: (10 points) The file `ps3-semantics.v` contains all the semantics definitions described so far. In the file `ps3-yourcode.v` use the semantics in order to prove that the program in Example 1 can be evaluated to a value of the form $g(n)$ for some n.

Problem 2 — Defining a simple type system (60 points)

One problem with the language as described so far is that one could write programs that end up with dangling pointers. For example, consider the following program.

Example 2

$$\begin{array}{l} let \ x = new_\tau \\ in \ context \ (\ \ let \ y = newL_\tau \ in \ \ x.f = y \ \ ); x.f.f := x.f \end{array}$$

This code will lead to a problem, because the object y will be deallocated, so when we try to modify the object outside the context, evaluation will fail.

Our goal for this part of the problem set is going to be to implement a type system that will help avoid the problem in the previous example. A strawman solution to the problem is to have two types: *Local* and *Global* . The idea is that new objects created with *newL* will be of type *Local* and objects created with *new* are going to be of type *Global* . In order to avoid any dangling references, we could disallow references from *Global* objects to *Local* objects. A problem with this approach is that the notion of locality is relative; in one context an object may be local, but from a nested context it is global, but it still needs to be distinguished from the truly global objects. We will achieve this by defining a type $Local_i$ where i is an offset from the current local context. The typing rules are as follows.

$$\frac{\Gamma(x)=\tau}{\Gamma,c\vdash x\ :\tau} \qquad \frac{}{\Gamma,c\vdash new{:}Global} \qquad \frac{\Gamma,c\vdash e_1{:}\tau_1 \qquad \Gamma;x{:}\tau_1,c\vdash e_2}{\Gamma,c\vdash\ let\ x{:}\tau_1{=}e_1\ in\ e_2\ :\tau_2}$$

$$\frac{\Gamma,c\vdash e{:}\tau}{\Gamma,c\vdash e.f{:}\tau} \qquad \frac{\Gamma\vdash e\ :\tau \quad \Gamma,c\vdash e'{:}\tau}{\Gamma\vdash e.f:=e'\ :\tau} \qquad \frac{}{\Gamma,c\vdash newL{:}Local_0}$$

$$\frac{updateTypes(\Gamma),(c{+}1)\vdash e{:}Global}{\Gamma,c\vdash\ context\ e\ {:}Global} \qquad \frac{\Gamma,c\vdash\ e_1{:}\tau' \qquad \Gamma,c\vdash\ e_2{:}\tau}{\Gamma,c\vdash\ e_1;e_2{:}\tau}$$

$$\frac{updateTypes(\Gamma),(c{+}1)\vdash e{:}Global}{\Gamma,c\vdash\ inside\ e{:}Global} \qquad \frac{}{\Gamma,c\vdash g(n){:}Global} \qquad \frac{c{=}t{+}cid}{\Gamma,c\vdash loc(n,cid){:}Local_t}$$

updateTypes finds any type $Local_i$ in Γ and replaces it with $Local_{i+1}$. The last three constructs correspond to constructs that do not appear in the original programs but are introduced by the small step semantics.

Part a: (10 points) The file `ps3-types.v` contains a skeleton of the type definitions listed above. Your goal is to complete this type system to match the above definitions.

Part b: (10 points) In the same file create 3 small programs in the language and prove that they type check according to your typing rules. Collectively your 3 programs should exercise all the typing rules.

Part c: (40 points) In the same file, there is a preservation theorem which you need to prove for this type system.

Problem 3 — Extending the type system (30 points)

One problem with the type system as defined so far is that it is too restrictive. Local objects can only point to other local objects in the same context. From the semantics, though, it should be clear that references from local objects to global objects would not pose any problems. From this intuition, it would seem that the following sub-typing relationship makes sense for the type system.

$$Global <: Local_i$$

$$\frac{i > j}{Local_i <: Local_j}$$

Together with the standard rule for subtyping

$$\frac{\Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash e : \tau}$$

This means that I can always use a global object in place of a local object, and I can always use an object from an enclosing context in place of an object local to the current context.

Part a: (5 points) Is this claim true? (hint, the answer is no) If no, show an example of a program that would typecheck thanks to this subtyping relation but would not execute correctly according to the semantics.

Part b: (15 points) One way to fix the problem is to have two kinds of fields

- **sametype** These fields always point to objects of the exact same type as the object to which they belong, so if e is $Local_2$ $e.fs$ is also of type $Local_2$ if field fs is a sametype field. Just like in the original type system.
- **supertype** These fields can point to objects in the same context or to objects in enclosing contexts or the global context.

In the file `ps3-yourcode.v`, define a formalization of the new type system using the `isSameType` predicate provided in ps3-types.v. Your typing rules should be defined under `typeCheckBetter` to distinguish them from the ones from Problem 2.

Part c: (10 points) In the same file create 3 small programs in the language and prove that they type check according to your typing rules. Collectively your 3 programs should exercise the new typing rules.

Part d: (20 points) **BONUS:** For an additional 20 points, prove a *preservation* theorem for your new type system.

6.820 Fundamentals of Program Analysis

Fall 2015