CS 6110 Homework 5

Lee Gao (lg342)

April 21, 2013

Excercise 1 Explicit Initialization

Compound data structures such as arrays, tuples, and records often need to be initialized step by step rather than being created all at once. Let us try to model safe, step-by-step initialization of tuples using an extension to the simply type λ:

$$
\begin{aligned}
e &::= x \mid e_1\ e_2 \mid \lambda x : \tau.e \mid b \mid \mathsf{malloc}\,\tau_1 \times \cdots \times \tau_n \mid \#n\ e \mid \#n\ e_1 := e_2 \\
b &::= n \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{null} \\
\tau &::= B \mid \tau_1 \to \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid (\tau_1 \times \cdots \times \tau_n) \backslash \{n_1 \cdots n_k\} \\
B &::= \mathsf{int} \mid \mathsf{bool} \mid 1 \\
v &::= b \mid \lambda x : \tau.e \mid (v_1, \cdots, v_n)
\end{aligned}
$$

And also our source language does not include explicit typles, we augment the grammar in order to define the small-step semantics.

$$e ::= \cdots \mid (v_1, \cdots, v_n)$$

(a) Extend the definition of the evaluation context and small-step operational semantics of the simply-typed λ to include the three new expressions: $\mathsf{malloc}\,\tau_1 * \cdots * \tau_n, \#n\ e$, and $\#n\ e_1 := e_2$.

Let's start with the evaluation context.

$$
\begin{aligned}
E ::= [\cdot] &\mid E\ e_2 \mid v\ E \\
&\mid \#n\ E \mid \#n\ E := e_2 \mid \#n\ v := E
\end{aligned}
$$

so basically we want to reduce all projection arguments into values, and we want to reduce left to right for initializations.

Next, let's consider the small step operational semantics. First, we know that we want malloc to step into a tuple, so we add the rule

$$\frac{}{\mathsf{malloc}\,t_1 * \cdots * t_n \to (\mathsf{null}, \cdots, \mathsf{null})}\ \textsc{Malloc}$$

it might be a bit unsettling at this point to note that we're loosing all type information since tuples are untagged so one might wonder how it would be possible to prove soundness if one can't even prove that one of the intermediate steps of the reduction will have some certain type. Fear not, later on, preservation will save our ass.

Next, let's consider projections.

$$\frac{}{\#k\ (v_1, \cdots, v_k, \cdots, v_n) \to v_k}\ \textsc{Proj}$$

Finally, let's consider initialization

$$\frac{}{\#k\ (v_1, \cdots, v_k, \cdots, v_n) := v_k' \to (v_1, \cdots, v_k', \cdots, v_n)}\ \textsc{Init}$$

Here, one natural question arises as to whether we should enforce the restriction that in the k^{th} initialization above, whether v_k must be null or not. From a soundness perspective this is unnecessary as we can eliminate such violations from the static semantics (we will later see that if we can typecheck this initialization, then we can find a derivation of $\Gamma \vdash v_k : 1$, which by some form of a normal-form lemma means that $v_k = \mathsf{null}$, so if we start off with the static semantic anyways, this requirement will be redundant).

(b) Extend the static semantics for the new expressions.
The easy case is just to extend the semantics for malloc.

$$\frac{}{\Gamma \vdash \mathsf{malloc}\,\tau_1 * \cdots * \tau_n : (\tau_1 * \cdots * \tau_n) \backslash \{1 \cdots n\}}\ \textsc{TMalloc}$$

However, how do we typecheck tuples? Suppose we have a judgment of the form

$$\Gamma \vdash (v_1, \cdots, v_n) : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \cdots, a_k\}$$

we know that for each $j \notin (a_1, \cdots, a_k)$, we must be able to typecheck v_j into type τ_j. But what about the masked indices? Well, we need to have them be type 1 to ensure that they have not been initialized yet, which gives the rule

$$\frac{\forall_{a_i}.\Gamma \vdash v_{a_i} : 1 \qquad \forall_{j \neq a_i}.\Gamma \vdash v_j : \tau_j}{\Gamma \vdash (v_1, \cdots, v_n) : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \cdots, a_k\}}\ \textsc{TTuple}$$

but this raises another interesting questions: what happened to the τ_{a_i}? Well, we didn't need them because we know they are going to be masked, but this results in an interesting observation: the same set of premises could end up giving us multiple type derivations to choose from, so in a sense, typechecking isn't "syntax-directed".
Next, let's consider projections.

$$\frac{\Gamma \vdash e : (\tau_1 * \cdots \tau_k \cdots * \tau_n) \backslash \{a_1, \cdots, a_m\} \qquad k \neq a_i}{\Gamma \vdash \#k\ e : \tau_k}\ \textsc{TProj}$$

and initializations

$$\frac{\Gamma \vdash e_1 : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \ldots, a_m, k\} \qquad \Gamma \vdash e_2 : \tau_k}{\Gamma \vdash \#k\ e_1 := e_2 : (\tau_1 * \cdots \tau_n) \backslash \{a_1, \ldots, a_m\}}\ \textsc{TInit}$$

(c) Now let us try to prove the soundness of the language:
For the sake of clarity, let's write out our operational semantic, static semantics, and evaluation context in full as a recap.

$$\frac{e \to e'}{E[e] \to E[e']}\ \textsc{Context} \qquad \frac{}{(\lambda x : \tau.e)\ v \to e\,\{v/x\}}\ \beta$$

$$\frac{}{\mathsf{malloc}\,t_1 * \cdots * t_n \to (\mathsf{null}, \cdots, \mathsf{null})}\ \textsc{Malloc} \qquad \frac{}{\#k\ (v_1, \cdots, v_k, \cdots, v_n) \to v_k}\ \textsc{Proj}$$

$$\frac{}{\#k\ (v_1, \cdots, v_k, \cdots, v_n) := v_k' \to (v_1, \cdots, v_k', \cdots, v_n)}\ \textsc{Init}$$

$$E ::= [\cdot] \mid E\ e_2 \mid v\ E \mid \#n\ E \mid \#n\ E := e_2 \mid \#n\ v := E$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{TVar} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}}\ \textsc{TInt} \qquad \frac{}{\Gamma \vdash \mathsf{True} : \mathsf{bool}}\ \textsc{TTrue}$$

$$\frac{}{\Gamma \vdash \mathsf{False} : \mathsf{bool}}\ \textsc{TFalse} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{TApp}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x : \tau'.e : \tau' \to \tau}\ \textsc{TLam} \qquad \frac{}{\Gamma \vdash \mathsf{null} : 1}\ \textsc{TNull}$$

$$\frac{}{\Gamma \vdash \mathsf{malloc}\,\tau_1 * \cdots * \tau_n : (\tau_1 * \cdots * \tau_n) \backslash \{1 \cdots n\}}\ \textsc{TMalloc}$$

$$\frac{\forall_{a_i}.\Gamma \vdash v_{a_i} : 1 \qquad \forall_{j \neq a_i}.\Gamma \vdash v_j : \tau_j}{\Gamma \vdash (v_1, \cdots, v_n) : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \cdots, a_k\}}\ \textsc{TTuple}$$

$$\frac{\Gamma \vdash e : (\tau_1 * \cdots \tau_k \cdots * \tau_n) \backslash \{a_1, \cdots, a_m\} \qquad k \neq a_i}{\Gamma \vdash \#k\ e : \tau_k}\ \textsc{TProj}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \ldots, a_m, k\} \qquad \Gamma \vdash e_2 : \tau_k}{\Gamma \vdash \#k\ e_1 := e_2 : (\tau_1 * \cdots \tau_n) \backslash \{a_1, \ldots, a_m\}}\ \textsc{TInit}$$

i. Formulate and prove a preservation lemma for the language, which states that evaluation preserves the type of the expression.

Lemma (Preservation)**.**

$$\vdash e : \tau \wedge e \to e' \implies \vdash e' : \tau$$

Proof. Assuming that $\vdash e : \tau$ (because otherwise the lemma holds vacuously), we will show this by induction on the derivation of $e \to e'$[1] with a case analysis on the last rule used in the derivation of $e \to e'$ using the immediate subderivation relation $\prec$ (which is well-founded as seen countless times before). Specifically, our induction hypothesis is

$$P(e \to e') \triangleq \vdash e : \tau \implies \vdash e' : \tau$$

β Here, we have $e = (\lambda x : \tau'.e_0)\ v$ and $e' = e_0\,\{v/x\}$, and $\vdash e : \tau$. By inspection, we find the following derivation of $\vdash e : \tau$:

$$\frac{\dfrac{x : \tau' \vdash e_0 : \tau}{\vdash \lambda x : \tau'.e_0 : \tau' \to \tau}\ \textsc{Lam} \qquad \vdash v : \tau'}{\vdash (\lambda x : \tau.e_0)\ v : \tau}\ \beta$$

[1]The lecture notes decided to induct over the derivation of the typechecking judgment $\vdash e : \tau$ instead, but this quickly becomes problematic in the Context case below when we attempt to show that $\vdash E[e] : \tau \implies \exists \tau'. \vdash e : \tau'$ such that $\vdash e : \tau'$ is furthermore a subderivation of $\vdash E[e] : \tau$ (in order to use the induction hypothesis $P(\vdash e : \tau)$). This is not true if $E = [\cdot]$, so we will need to resort to hacks such as normalizing proofs into cases where if the concluding rule is Context, it cannot use $E = [\cdot]$. It is much cleaner to just do induction over the derivation of the small step SOS.

so that there must exist some derivation of both of the leaves of the above proof, so we have a derivation for $x : \tau' \vdash e_0 : \tau$ and $v : \tau'$.

Now, we know that $e \to e_0\{v/x\}$, so by the substitution lemma, we can conclude that $\vdash e_0\{v/x\} : \tau$, which shows the case.

Malloc Here, we have $e = \mathsf{malloc}\,\tau_1 * \cdots * \tau_n$ and $e' = (\mathsf{null}, \cdots, \mathsf{null})$. We get immediately that $\vdash e : (\tau_1 * \cdots * \tau_n) \backslash \{1, \ldots, n\} = \tau$ We can immediately construct the derivation that

$$\frac{\vdash \mathsf{null} : 1 \qquad \cdots \qquad \vdash \mathsf{null} : 1}{\vdash e' : (\tau_1 * \cdots * \tau_n) \backslash \{1, \ldots, n\}}\ \textsc{TTuple}$$

which shows that $\vdash e' : \tau$ and shows the case.

Proj Here, we have $e = \#k\ (v_1, \cdots, v_n)$ and $e' = v_k$; furthermore, we have some derivation $\vdash e : \tau_k$, which, given any derivation concluding $\vdash e : \tau_k$, must look like

$$\frac{\dfrac{\cdots \qquad \overbrace{\vdash v_k : \tau_k}^{\text{because } k \neq a_i}}{\vdash (v_1, \cdots, v_n) : (\tau_1 * \cdots * \tau_k * \cdots * \tau_n) \backslash \{a_1, \ldots, a_m\}}\ \textsc{TTuple} \qquad k \neq a_i}{\vdash \#k(v_1, \cdots, v_n) : \tau_k}\ \textsc{TProj}$$

but then we have a derivation of $\vdash v_k : \tau_k$, which shows the case.

Init Here, we have $e = \#k\ (v_1, \cdots, v_n) := v'_k$ and $e' = (v_1, \cdots, v'_k, \cdots, v_n)$; furthermore we have some derivation that $\vdash e : \tau$. Now, consider any derivation of $\vdash e : \tau$, it must be the case that its derivation must look like

$$\frac{\dfrac{\forall_{a_i}. \vdash v_{a_i} : 1 \qquad \forall_{j \neq a_i, j \neq k}. \vdash v_j : \tau_j \qquad \cdots}{\vdash (v_1, \cdots, v_n) : (\tau_1 \cdots \tau_n) \backslash \{a_1, \ldots, a_m, k\}}\ \textsc{TTuple} \qquad \vdash v'_k : \tau_k}{\vdash e : (\tau_1 \cdots \tau_n) \backslash \{a_1, \ldots, a_m\}}\ \textsc{TInit}$$

where $k \neq a_i$. Now, using these facts immediately, we can construct another derivation that

$$\frac{\forall_{a_i}. \vdash v_{a_i} : 1 \qquad \forall_{j \neq a_i, j \neq k} \vdash v_j : \tau_j \qquad \vdash v'_k : \tau_k}{\vdash (v_1, \cdots, v'_k, \cdots, v_n) : (\tau_1 \cdots \tau_n) \backslash \{a_1, \ldots, a_m\}}\ \textsc{TTuple}$$

which concludes that $\vdash e' : \tau$ and hence shows the case.

Context Here, $e = E[e_0], e' = E[e'_0]$ and we also get a derivation of $e_0 \to e'_0$ as well as $\vdash e : \tau$.

In order to proceed any further, we need to do a second structural induction on evaluation context on the proposition $P'(E) \triangleq \vdash E[e_0] : \tau \implies \exists \tau'. \vdash e'_0 : \tau'$, using the immediate subcontext relation $\prec'$ which is well-founded by construction. We need this proposition P' in order to use our induction hypothesis P (otherwise we cannot claim that e' is well-formed). Let's do a case analysis on the struct of E.

- $E = [\cdot]$ – Here, $E[e_0] = e_0$, so if we have $\vdash [e_0] : \tau$, then obviously we have $\vdash e_0 : \tau$, which shows the case.
- $E = E_1\ e_2$ – Here, $E[e_0] = E_1[e_0]\ e_2$ and we also have $\vdash E_1[e_0]\ e_2 : \tau$. The only derivations that concludes in this must be of the form

$$\frac{\vdash E_1[e_0] : \tau' \to \tau \qquad \vdash e_2 : \tau'}{\vdash E_1[e_0]\ e_2 : \tau}\ \textsc{TApp}$$

and by inversion, we get that $\vdash E_1[e_0] : \tau' \to \tau$. Now, since $E_1 \prec' E$, we can apply the induction hypothesis $P'(E_1)$ to get that $\vdash E_1[e_0] : \tau' \to \tau \implies \exists \tau''. \vdash e_0 : \tau''$, which shows the case.

- $E = v\ E_2$ – Here, we have $\vdash v\ E_2[e_0] : \tau$; the only derivation that concludes in this must be of the form

$$\frac{\vdash v : \tau' \to \tau \qquad \vdash E_2[e_0] : \tau'}{\vdash v\ E_2[e_0] : \tau}\ \textsc{TApp}$$

and we get a derivation of $\vdash E_2[e_0] : \tau'$. But $E_2 \prec' E$, so by the induction hypothesis $P'(E_2)$, we get that $\exists \tau''. \vdash e_0 : \tau''$, which shows the case.

- $E = \#k\ E'$ – Here, we get $\vdash \#k\ E'[e_0] : \tau_k$, and the only rule that applies is

$$\frac{\vdash E'[e_0] : \tau' \qquad \cdots}{\vdash \#k\ E'[e_0] : \tau_k}\ \textsc{TProj}$$

for some τ' a masked tuple type. Now, since $E' \prec' E$, then from $P'(E')$, we know that $\exists \tau''. \vdash e_0 : \tau''$, which shows the case.

- $E = \#k\ E_1 := e_2$ – Here, we get $\vdash \#k\ E_1[e_0] := e_2 : \tau$, and the only rule that applies is

$$\frac{\vdash E_1[e_0] : \tau' \qquad \cdots}{\vdash \#k\ E_1[e_0] := e_2 : \tau}\ \textsc{TInit}$$

so we get a derivation of $\vdash E_1[e_0] : \tau'$ for some τ'. Since $E_1 \prec' E$, then by IH $P'(E_1)$, we have that $\exists \tau''. \vdash e_0 : \tau''$ which shows the case.

- $E = \#\ v := E_2$ – Here, we get $\vdash \#k\ v := E_2[e_0] : \tau$, and the only rule that applies is

$$\frac{\cdots \qquad \vdash E_2[e_0] : \tau_k}{\vdash \#k\ v := E_2[e_0] : \tau}\ \textsc{TInit}$$

so we get a derivation of $\vdash E_2[e_0] : \tau_k$ for some τ_k. since $E_2 \prec' E$, then $P'(E_2)$ gives use $\exists \tau''. \vdash e_0 : \tau''$ which shows the case and concludes the proof that $\forall E. P'(E)$. □

From this, we can use $P'(E)$ for $e = E[e_0]$ and $\vdash E[e_0] : \tau$ to get that there exists some τ' such that $\vdash e_0 : \tau'$. Now, because $e_0 \to e_0' \prec E[e_0] \to E[e_0']$[2], we can apply the induction hypothesis $P(e_0 \to e_0')$ and $\vdash e_0 : \tau'$ to get $\vdash e_0' : \tau'$; but once again, we're stuck.

We will now need to employ structural induction on E again to show the proposition $P''(E) \triangleq \vdash E[e] : \tau \land \vdash e : \tau' \land \vdash e' : \tau' \implies \vdash E[e'] : \tau$[3] using the same subcontext relation $\prec'$ used above. Let's do a case analysis on the structure of E.

- $E = [\cdot]$ – Here, $\tau = \tau'$, so we immediately get $\vdash e' : \tau$.

[2]It might be a bit confusing sometimes to make sense of how this works for the case $\dfrac{e \to e'}{[e] \to [e']}$, since it is in some semblance reflexive hence it appears to not be "well-founded"; this just means that ordering by the conclusions isn't well-founded (i.e. if we use a $\prec$ relation on the conclusions of derivations rather than derivations themselves). But since all valid derivations must have finite height, then the top part is smaller than the full derivation and is hence different, so we're good.

[3]We would typically bring this out as a full Context lemma, but it is pretty compact so I left it in

- $E = E_1\ e_2$ – Here, $\vdash E_1[e]\ e_2 : \tau$ and $\vdash e, e' : \tau'$. The only derivation to conclude in $\vdash E[e] : \tau$ is

$$\frac{\vdash E_1[e] : \tau'' \to \tau \qquad \vdash e_2 : \tau''}{\vdash E_1[e]\ e_2 : \tau}\ \textsc{TApp}$$

  Since $E_1 \prec' E$, then by $P''(E_1)$, we get that $\vdash E_1[e'] : \tau'' \to \tau$. Furthermore, inverting the above TApp instance, we also get $\vdash e_2 : \tau''$, so we can immediately construct the derivation

$$\frac{\vdash E_1[e'] : \tau'' \to \tau \qquad \vdash e_2 : \tau''}{\vdash E_1[e']\ e_2 : \tau}\ \textsc{TApp}$$

  which shows the case.
- $E = v\ E_2$ – Here, $\vdash v\ E_2[e] : \tau$ and $\vdash e, e' : \tau'$. The only derivation to conclude $\vdash E[e] : \tau$ must be of the form

$$\frac{\vdash v : \tau'' \to \tau \qquad \vdash E_2[e] : \tau''}{\vdash v\ E_1[e] : \tau}\ \textsc{TApp}$$

  which gives the derivations for $\vdash v : \tau'' \to \tau$ and $\vdash E_2[e] : \tau''$. But since $E_2 \prec E$, we can apply the induction hypothesis $P''(E_2)$ to get $\vdash E_2[e'] : \tau''$, from which we can construct the derivation

$$\frac{\vdash v : \tau'' \to \tau \qquad \vdash E_2[e'] : \tau''}{\vdash v\ E_1[e'] : \tau}\ \textsc{TApp}$$

  which shows the case.
- $E = \#k\ E'$ – Here, $\vdash \#k\ E'[e] : \tau_k$ and $\vdash e, e' : \tau'$. The only derivations to conclude in $\vdash E[e] : \tau_k$ must be of the form

$$\frac{\vdash E'[e] : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \cdots, a_m\} \qquad k \neq a_i}{\vdash \#k\ E'[e] : \tau_k}\ \textsc{TProj}$$

  but since $E' \prec' E$, then $P''(E')$ gives $\vdash E'[e'] : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \cdots, a_m\}$, from which we can construct a new derivation

$$\frac{\vdash E'[e'] : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \cdots, a_m\} \qquad k \neq a_i}{\vdash \#k\ E'[e'] : \tau_k}\ \textsc{TProj}$$

  which shows the case.
- $E = \#k\ E_1 := e_2$ – Here, $\vdash \#k\ E_1[e] := e_2 : \tau$ and $\vdash e, e' : \tau'$. The only derivations to conclude in the above must be of the form

$$\frac{\vdash E_1[e] : \tau \backslash \{k\} \qquad \vdash e_2 : \tau_k}{\vdash \#k\ E_1[e] := e_2 : \tau}\ \textsc{TInit}$$

  since $E_1 \prec' E$, then we can apply the induction hypothesis $P''(E_1)$ to get $\vdash E_1[e'] : \tau \backslash \{k\}$, from which we can construct the derivation

$$\frac{\vdash E_1[e'] : \tau \backslash \{k\} \qquad \vdash e_2 : \tau_k}{\vdash \#k\ E_1[e'] := e_2 : \tau}\ \textsc{TInit}$$

  which shows the the case.

- $E = \#k\ v := E_2$ – Here, $\vdash \#k\ v := E_2[e] : \tau$ and $\vdash e, e' : \tau'$. The only derivations to conclude in the above must be of the form

$$\frac{\vdots^1 \qquad \vdash E_2[e] : \tau_k}{\vdash \#k\ v := E_2[e] : \tau}\ \textsc{TInit}$$

but since $E_2 \prec' E$, we can apply the induction hypothesis $P''(E_2)$ to get $\vdash E_2[e'] : \tau_k$, and construct the derivation

$$\frac{\vdots^1 \qquad \vdash E_2[e'] : \tau_k}{\vdash \#k\ v := E_2[e'] : \tau}\ \textsc{TInit}$$

which shows the case and concludes the proof of $\forall E.P''(E)$. □

Finally, because we have $\vdash E[e_0] : \tau, \vdash e_0, e_0' : \tau'$ and $P''(E)$, we get $\vdash E[e_0'] : \tau$, which shows the case and concludes the proof of preservation. □

Of course, we're going to need to give the substitution lemma.

Lemma (Substitution)**.**

$$\Gamma, x : \tau' \vdash e : \tau \wedge \vdash v : \tau' \implies \Gamma \vdash e\{v/x\} : \tau$$

Proof. As per the proof in the lecture notes, we will prove this by induction on the derivation of the typechecking judgment $\Gamma \vdash e : \tau$ (using the immediate subderivation relation which we've already argued to be well-founded). Our induction hypothesis is simply just

$$P(\Gamma, x : \tau' \vdash e : \tau) \triangleq \vdash v : \tau' \implies \Gamma \vdash e\{v/x\} : \tau$$

Let's do a case analysis on the structure of e.

- $e = b$ – Here, $e\{v/x\} = e$, so $\Gamma \vdash b\{v/x\} : \tau$ holds trivially.
- $e = x$ – Here, e is the variable we want to replace, so $e\{v/x\} = v$; furthermore, $\Gamma, x : \tau' \vdash x : \tau \implies \tau = \tau'$. Therefore, we easily get $\Gamma \vdash x\{v/x\} = v : \tau' = \tau$, which shows the case.
- $e = y \neq x$ – Here, $y\{v/x\} = y$, so if $\Gamma, x : \tau' \vdash y : \tau$, then from TVar, then $\Gamma(y) = \tau$. From this, we can immediately construct $\Gamma \vdash y\{v/x\} = y : \Gamma(y) = \tau$, which shows the case.
- $e = \lambda x : \tau''.e'$ – Here, $e\{v/x\} = e$, so we can get a derivation $\Gamma, x : \tau' \vdash e : \tau'' \to \tau''' = \tau$, which looks like

$$\frac{\Gamma, x : \tau', x : \tau'' \vdash e' : \tau'''}{\Gamma, x : \tau' \vdash e : \tau'' \to \tau'''}\ \textsc{TLam}$$

so we get for free the derivation up top: $\Gamma, x : \tau', x : \tau'' \vdash e' : \tau''' \iff \Gamma, x : \tau'' \vdash e' : \tau'''$. From this, we can construct the derivation

$$\frac{\Gamma, x : \tau'' \vdash e' : \tau'''}{\Gamma \vdash e : \tau'' \to \tau'''}\ \textsc{TLam}$$

but since $e = e\{v/x\}$, this shows the case.

- $e = \lambda y : \tau''.e'$ where $x \neq y$ – Now, because we have $\vdash v : \tau'$ (so that v is well-formed), then v cannot contain free variables, so we fall into the substitution case where $y \notin \textsc{Fvs}(v)$, and $e\{v/x\} = \lambda y : \tau''.e'\{v/x\}$.
Now, the only possible derivation of $\Gamma, x : \tau' \vdash \lambda y : \tau''.e' : \tau'' \to \tau'' = \tau'$ must look like
$$\frac{\Gamma, x : \tau', y : \tau'' \vdash e' : \tau'''}{\Gamma, x : \tau' \vdash \lambda y : \tau''.e' : \tau'' \to \tau'''}\ \textsc{TLam}$$
from this, we get the derivation $\Gamma, x : \tau', y : \tau'' \vdash e' : \tau'''$ which $\prec \Gamma, x : \tau' \vdash e : \tau$, so we can use $P((\Gamma, y : \tau''), x : \tau' \vdash e' : \tau''')$ and $\vdash v : \tau'$ to get $\Gamma, y : \tau'' \vdash e'\{v/x\} : \tau'''$. But then, we can construct the derivation
$$\frac{\Gamma, y : \tau'' \vdash e'\{v/x\} : \tau'''}{\Gamma \vdash \lambda y : \tau''.e'\{v/x\} : \tau'' \to \tau''' = \tau}\ \textsc{TLam}$$
hence giving $\Gamma \vdash e\{v/x\} : \tau$ which shows the case.
- $e = e_1\ e_2$ – Here, $e\{v/x\} = e_1\{v/x\}\ e_2\{v/x\}$, and we have $\Gamma, x : \tau' \vdash e_1\ e_2 : \tau$, which must have a derivation of the form
$$\frac{\Gamma, x : \tau' \vdash e_1 : \tau'' \to \tau \qquad \Gamma, x : \tau' \vdash e_2 : \tau''}{\Gamma, x : \tau' \vdash e_1\ e_2 : \tau}\ \textsc{TApp}$$
from this we get the derivations $\Gamma, x : \tau' \vdash e_1 : \tau'' \to \tau$ and $\Gamma, x : \tau' \vdash e_2 : \tau''$, both of which are subderivations of $\Gamma, x : \tau' \vdash e : \tau$, so we can apply the induction hypothesis on both $P(\Gamma, x : \tau' \vdash e_1 : \tau'' \to \tau)$ to get $\Gamma \vdash e_1\{v/x\} : \tau'' \to \tau$ and $\Gamma \vdash e_2\{v/x\} : \tau''$. Now, we can just construct the derivation
$$\frac{\Gamma \vdash e_1\{v/x\} : \tau'' \to \tau \qquad \Gamma \vdash e_2\{v/x\} : \tau''}{\Gamma \vdash e_1\{v/x\}\ e_2\{v/x\} : \tau}\ \textsc{TApp}$$
which shows the case.
- $e = (v_1, \cdots, v_n)$ – Here, $e\{v/x\} = (v_i\{v/x\})$ and we have $\Gamma, x : \tau' \vdash (v_1, \cdots, v_n) : \tau = (\tau_i)\backslash\{a_i\}$, which must only have derivations of the form
$$\frac{\forall_{a_i}.\Gamma, x : \tau' \vdash v_{a_i} : 1 \qquad \forall_{j \neq a_i}.\Gamma, x : \tau' \vdash v_j : \tau_j}{\Gamma, x : \tau' \vdash (v_1, \cdots, v_n) : (\tau_j)\backslash\{a_i\}}\ \textsc{TTuple}$$
but for each such derivation, we can use the induction hypothesis (or some kind of a normal-form lemma on the 1-type) on each of $P(\Gamma, x : \tau' \vdash v_{a_i} : 1)$ to get $\Gamma \vdash v_{a_i}\{v/x\} : 1$ and on $P(\Gamma, x : \tau' \vdash v_j : \tau_j)$ for $j \neq a_i$ to get $\Gamma \vdash v_j\{v/x\} : \tau_j$, from which we can construct the derivation
$$\frac{\forall_{a_i}.\Gamma \vdash v_{a_i}\{v/x\} : 1 \qquad \forall_{j \neq a_i}.\Gamma \vdash v_j\{v/x\} : \tau_j}{\Gamma, \vdash (v_1\{v/x\}, \cdots, v_n\{v/x\}) : (\tau_j)\backslash\{a_i\} = \tau}\ \textsc{TTuple}$$
for all such possible derivation, which shows the case.
- $e = \mathsf{malloc}\,\tau_1 \times \cdots \times \tau_n$ – Here, $e\{v/x\} = e$, $\tau = \tau_1 \times \cdots \times \tau_n$. We can immediately construct the derivation
$$\frac{}{\Gamma \vdash \mathsf{malloc}\,\tau_1 \times \cdots \times \tau_n : \tau_1 \times \cdots \times \tau_n = \tau}\ \textsc{TMalloc}$$
which shows the case.

- $e = \#k\ e'$ – Here, $e\{v/x\} = \#k\ e'\{v/x\}$. Now, $\vdash e : \tau_k$ must only have derivations of the form

$$\frac{\Gamma, x : \tau' \vdash e' : (\tau_j) \setminus \{a_i\} \qquad k \neq a_i}{\Gamma, x : \tau' \vdash \#k\ e' : \tau_k}\ \textsc{TProj}$$

for each such derivation, by the induction hypothesis on $P(\Gamma, x : \tau' \vdash e' : (\tau_j) \setminus \{a_i\})$, we get $\Gamma \vdash e'\{v/x\} : (\tau_j) \setminus \{a_i\}$, which we can use immediately to construct the derivation

$$\frac{\Gamma \vdash e'\{v/x\} : (\tau_j) \setminus \{a_i\} \qquad k \neq a_i}{\Gamma \vdash \#k\ e'\{v/x\} : \tau_k}\ \textsc{TProj}$$

which shows the case.

- $e = \#k\ e_1 = e_2$ – Here, $e\{v/x\} = \#k\ e_1\{v/x\} = e_2\{v/x\}$ and $\Gamma, x : \tau' \vdash e : \tau$ can only have derivations of the form

$$\frac{\Gamma, x : \tau' \vdash e_1 : (\tau_j) \setminus \{a_i, k\} \qquad \Gamma, x : \tau' \vdash e_2 : \tau_k}{\Gamma, x : \tau' \vdash \#k\ e_1 := e_2 : (\tau_j) \setminus \{a_i\}}\ \textsc{TInit}$$

applying the induction hypothesis on both of the subderivations above gives us $\Gamma \vdash e_1\{v/x\} : (\tau_j)\{a_i, k\}$ and $\Gamma \vdash e_2\{v/x\} : \tau_k$, which can be used immediately to build the derivation

$$\frac{\Gamma \vdash e_1\{v/x\} : (\tau_j)\{a_i, k\} \qquad \Gamma \vdash e_2\{v/x\} : \tau_k}{\Gamma \vdash e\{v/x\} = \#k\ e_1\{v/x\} := e_2\{v/x\} : (\tau_j) \setminus \{a_i\}}\ \textsc{TInit}$$

which shows the case and concludes the proof. □

ii. Formulate and prove a progress lemma, which states that a well-typed program is either in a normal form, or can be stepped into another program.

Lemma (Progress)**.**

$$\vdash e : \tau \implies e \textit{ is a value} \vee \exists e'.e \to e'$$

Proof. We will prove this by induction on the typing derivation of e (with $\prec$ being the subderivation relation which we have already argued to be well-founded) on the proposition

$$P(\vdash e : \tau) \triangleq e \text{ is a value} \vee \exists e'.e \to e'$$

with a case analysis on on the structure of e. Note that we exclude the cases that are vacuously false, like $e = x$.

- $e = b$ – b is already a value, so this case is trivial.
- $e = \lambda x : \tau'.e'$ – this is also already a value
- $e = (v_1, \cdots, v_n)$ – this is also already a value
- $e = e_0\ e_1$ – Here, we have a derivation of the form

$$\frac{\vdash e_0 : \tau' \to \tau \qquad \vdash e_1 : \tau'}{\vdash e_0\ e_1 : \tau}\ \textsc{TApp}$$

applying the induction hypothesis to both of the subderivations on top, we get that e_0 is either a value or steps to e_0', and e_1 is either a value or steps to e_1'. Let's do a case-by-case analysis here based on excluded middle:

– if e_0, e_1 are both values, then by inspection, the only value that type checks to a function type is a lambda, so $e_0 = \lambda x : \tau'.e'$ and $e_1 = v_1$, at which point we can apply the β reduction rule and take a step.
– if e_0 is not a value, then we can construct the evaluation context that $e = [e_0]\ e_1$, and since $e_0 \to e_0'$, we can construct the derivation $\dfrac{e_0 \to e_0'}{[e_0]\ e_1 \to [e_0']\ e_1}$, hence taking a step.
– e_0 is a value, but e_1 is not, then we can construct the evaluation context that $e = E[e_1] = e_0\ [e_1]$ and since $e_1 \to e_1'$, we can construct the derivation $\dfrac{e_1 \to e_1'}{e_0\ [e_1] \to e_0\ [e_1']}$, hence taking a step.

which shows the case.

- $e = \mathsf{malloc}\,\tau_1 \times \cdots \times \tau_n$ – Easy, $\dfrac{}{e \to (\mathsf{null}, \cdots, \mathsf{null})}$, hence taking a step.
- $e = \#k\ e'$ – Here, we have must have some derivation of the form

$$\frac{\vdash e' : (\tau_j) \backslash \{a_i\} \qquad k \neq a_i}{\vdash \#k\ e' : \tau_k}\ \textsc{TProj}$$

so by applying the induction hypothesis on the only premise (a subderivation) on top, we get that either e' is a value or it can take a step to e''. Let's do a case analysis here based on excluded middle:
– e' is a value, in which case by inspection, the only way that a value can typecheck to a masked-tuple type shown above is if $e' = (v_1, \cdots, v_k, \cdots v_n)$. But then, we can just construct the derivation $\dfrac{}{\#k\ (v_1, \cdots, v_k, \cdots v_n) \to v_k}$, hence taking a step.
– $e' \to e''$, in which case we can construct an evaluation context $e = \#k\ [e']$ and construct the derivation $\dfrac{e' \to e''}{\#k\ [e'] \to \#k\ [e'']}$, hence taking a step.

which shows the case.

- $e = \#k\ e_1 := e_2$ – Here, we must have some derivation of the form

$$\frac{\vdash e_1 : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \ldots, a_m, k\} \qquad \vdash e_2 : \tau_k}{\vdash \#k\ e_1 := e_2 : (\tau_1 * \cdots \tau_n) \backslash \{a_1, \ldots, a_m\}}\ \textsc{TInit}$$

applying the induction hypothesis to both of the suderivations on top tells us that either e_1 is a value or it can step to e_1', and the same for e_2. Here, let's do a case-by-case analysis based on excluded middle:
– Both e_1 and e_2 are values. Since $\vdash e_1 : (\tau_1 * \cdots * \tau_n) \backslash \{a_1, \ldots, a_m, k\}$, then by inspection of the typing judgments, the only value that can be given that type must be of the form $(v_1, \cdots v_k \cdots, v_n)$ where $\vdash v_k : 1$ from the inspection also tells us that $v_k = \mathsf{null}$. Let $e_2 = v_k'$, then we can immediately apply the derivation $\dfrac{}{\#k\ (v_1, \cdots v_k \cdots, v_n) := v_k' \to (v_1, \cdots v_k' \cdots, v_n)}$, hence taking a step.
– e_1 isn't a value. Here, we can construct an evaluation context $e = \#k\ [e_1] := e_2$ so that we can construct the derivation $\dfrac{e_1 \to e_1'}{\#k\ [e_1] := e_2 \to \#k\ [e_1'] := e_2}$, hence taking a step.

– e_1 is a value but e_2 isn't. Here, we can construct an evaluation context $e = \#k\ e_1 := [e_2]$ so that we can construct the derivation $\dfrac{e_2 \to e_2'}{\#k\ e_1 := [e_2] \to \#k\ e_1 := [e_2']}$, hence taking a step.

this shows the case and concludes the proof. □

iii. Formulate the soundness theorem.

Theorem (Soundness)**.**

$$\vdash e : \tau \wedge e \to^* e' \implies e' \textit{ is a value} \vee \exists e''.e' \to e''$$

(d) The language, as described above, ensures that every element in a tuple is initialized exactly once. However, it is sometimes desirable to enforce reinitialization, for example, to disallow further accesses to some sensitive data, or when the computation is staged, to give the next stage a fresh start.

Update the operational and static semantics to support enforcement of reinitializing an already initialized element of a tuple, while still ensuring the soundness. You do not have to redo the soundness proof.

Let's consider the subtyping relation between just τ and $\tau\backslash 1$. Now, we can use τ whenever $\tau\backslash 1$ is required, including reinitialization, because when we are working with

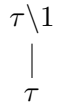
$$\#k\ x : \tau := e$$

we are in effect promoting x to

$$\#k\ x : \tau\backslash 1 := e$$

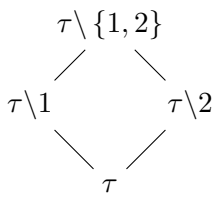
We can draw this out as a Hasse diagram:

Figure 1: Hasse diagram of a singleton tuple.



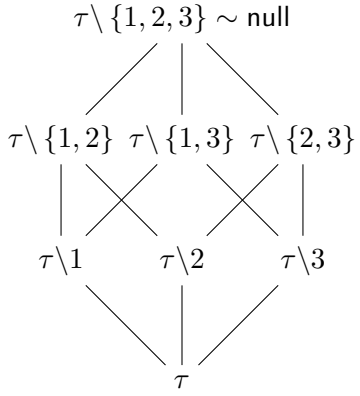
Similarly, suppose we have a tuple type $\tau = (\tau_1, \tau_2)$, we can use τ. Furthermore, anytime we have a $\tau\backslash 1$, we can use it when $\tau\backslash\{1, 2\}$ is expected, and so on. This can be drawn as

Figure 2: Hasse diagram of a pair.



the triple case is the most illuminating

Figure 3: Hasse diagram of a triple. We're allowed to substitute a type for any other as long as the other type can be reached using only upward edges from the source type.



Each time you do an update, you're effectively moving down a level (or staying the same if you're reinitializing a non-masked field) because everything on the lower levels can be used as if they. Therefore, this subtyping relation concisely captures the complete lattice of $(P_\omega, \subseteq)$! Therefore, when we want to reinitialize the second field of $\tau\setminus 1$, we can just use the fact that $\tau\setminus 1 \leq \tau\setminus\{1,2\}$ to use it as if it was masked!

Now, because we originally made the SOS behave as if there were no constraints on the initialization (beyond count mismatching), we don't need to do anything extra for the operational semantics. Furthermore, we can bake the subsumption semantics into the typing judgments themselves. To do this, note that we don't have to change anything on the $\Gamma \vdash \mathsf{malloc}(\tau_i) : (\tau_i)\,\{i\}$ because $(\tau_i)\,\{i\}$ is already $\top$. So the only place we need to change is the TTuple rule. We can instead rewrite it as

$$\frac{\exists B.\forall i \in B.\Gamma \vdash v_i : 1 \qquad \forall j \notin B.\Gamma \vdash v_j : \tau_j \qquad B \subseteq A}{\Gamma \vdash (v_i) : (\tau_i)\backslash A}\ \textsc{TTuple-Subsumption}$$

This makes sense: the only situations relevant to tuples during which we would expect to get stuck occurs only when we attempt to use an uninitialized field of a tuple. This subsumption semantic only allows us to use a tuple when it's higher up on the lattice (aka when the program expects less fields to be filled) so it can never attempt to use a noninitialized field, only treat possibly filled fields as if they are uninitialized, which is already well-defined in the structural operational semantic. So in effect, we're just making the type system more complete :)

Excercise 2 Continuation Passing Style

In this problem you will implement a continuation passing style translation.

The source and target languages for this question are similar respectively to the source language and the second intermediate language of the lecture notes. The source language is described below

$$e ::= n \mid x \mid \lambda x.e \mid e_0\ e_1 \mid (e_i) \mid \#n\ e \mid e_0 + e_1 \mid \mathsf{let}\, x = e_0\, \mathsf{in}\, e_1 \mid \mathsf{ifp}(e_0)\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \mid \mathsf{cwcc}\ e$$

(a) Extend the CPS translation given in Lecture 16 with a translation for the cwcc construct.

Translation into Il1

At first glance, we would want to translate cwcc immediately as

$$[\![\mathsf{cwcc}\ e]\!]k = [\![e]\!](\lambda f.f\ k\ k)$$

but there's a slight problem with this method of translation where we allow continuations to be administrative lambdas: let's just illustrate this with a really simple example:

$$\begin{aligned}
[\![\mathsf{cwcc}\ (\lambda f.f\ 3)]\!]k &= [\![\lambda f.f\ 3]\!](\lambda g.g\ k\ k) \\
&= \mathsf{let}\, k' = \lambda g.g\ k\ k\, \mathsf{in}\, k'\ (\lambda f k''.[\![f\ 3]\!]k'') \\
&=_{\eta} \mathsf{let}\, k' = \lambda g.g\ k\ k\, \mathsf{in}\, k'\ (\lambda f k''.f\ 3\ k'') \\
&=_{\beta\eta} (\lambda f k''.f\ 3\ k'')\ k\ k \\
&=_{\beta} k\ 3\ k
\end{aligned}$$

now, suppose this was part of an addition operation, then we would've have translated that operation into a continuation as an administrative lambda:

$$[\![3 + e]\!]\mathsf{halt} = [\![3]\!](\lambda n.[\![e]\!](\lambda m.\mathsf{halt}\ n + m))$$

so if $[\![3]\!]k = k\ 3$ and $e = \mathsf{cwcc}(\lambda f.f\ 3)$ as above, the entire thing reduces down to

$$(\lambda n.\, \mathsf{let}\, k = \lambda m.\mathsf{halt}\ n + m\, \mathsf{in}\, k\ 3\ k)\ 3$$

see the problem yet? Let's do one more round of β reduction:

$$\mathsf{let}\, k = \lambda m.\mathsf{halt}\ 3 + m\, \mathsf{in}\, k\ 3\ k$$

and one last reduction gets us to

$$(\lambda m.\mathsf{halt}\ 3 + m)\ 3\ (\lambda m.\mathsf{halt}\ 3 + m)$$

here, we're applying a 1-adic function to 2 arguments! Recall that we're translating all abstractions as 2-adic functions, but since we're allowed to call continuations, which were translated as 1-adic administrative lambdas, we're going to have an argument mismatch problem.

In order to support cwcc, not only do we need the above translation, we also need to make all administrative lambdas into 2-adic functions as well, where the first argument is the value given to the continuation.

But this raises a good question: what should we stick into that second parameter for continuations? Coupled to this question is the question of how to handle a cwcc into a function that returns? How should $\mathsf{cwcc}(\lambda x.x)$ behave? Well, here, we pass it the current continuation k, but it never calls k, instead, it returns... and that's the question:

- Where does it return to?
- What does it return with?

Here, I'm going to return back to the most recently bound continuation: since this is treated as a function, this is just that second argument. Furthermore, it should just return what ever it evaluates to as the argument. That is, cwcc f is pretty much calling f with the current continuation!

the argument to f

$$[\![\mathsf{cwcc}\ \mathsf{e}]\!]\underbrace{k}_{\text{current}} = [\![e]\!](\lambda f\kappa.f\ k\ \ k\ )$$

where to go in case of returning

so we will evaluate e, drop its value and the most recent continuation off into $\lambda f\kappa.f\ k\ k$, and we will call f with the current continuation as the argument, but in case f returns, it will drop it off into k as the point of return.

Let's first describe IL1, which is the intermediate language that we're going to translate into first.

$$\begin{aligned} v &::= n \mid x \mid \lambda xk.c \mid \mathsf{halt} \\ e &::= v \mid v_0 + v_1 \mid (v_i) \mid \pi_n v \mid \mathsf{ifp}(v_0)\,\mathsf{then}\,v_1\,\mathsf{else}\,v_2 \\ c &::= \mathsf{let}\,x = e\,\mathsf{in}\,c \mid v_0\ v_1\ v_2 \end{aligned}$$

Continuations will be modeled as $\lambda x\kappa.c$ where the second κ will be guaranteed to be discarded.[4] We will define this translation from source to IL1 as

$$\begin{aligned} [\![n]\!]k &= k\ n\ k \\ [\![x]\!]k &= k\ x\ k \\ [\![\lambda x.e]\!]k &= k\ (\lambda xk'.[\![e]\!]\ k')\ k \\ [\![e_0\ e_1]\!]k &= [\![e_0]\!](\lambda f\kappa.[\![e_1]\!](\lambda v\kappa'.f\ v\ k)) \\ [\![(e_1,\cdots,e_n)]\!]k &= [\![e_1]\!](\lambda x_1\kappa_1.\cdots[\![e_n]\!](\lambda x_n\kappa_n.k\ (x_1,\cdots,x_n)\ k)\cdots) \\ [\![\pi_n\ e]\!]k &= [\![e]\!](\lambda p\kappa.\,\mathsf{let}\,v = \#n\ p\,\mathsf{in}\,k\ v\ k) \\ [\![e_0 + e_1]\!]k &= [\![e_0]\!](\lambda x_0\kappa_0.[\![e_1]\!](\lambda x_1\kappa_1.\,\mathsf{let}\,n = x_0 + x_1\,\mathsf{in}\,k\ n\ k)) \\ [\![\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2]\!]k &= [\![e_1]\!](\lambda x\kappa.[\![e_2]\!]\ k) \\ [\![\mathsf{ifp}(e_0)\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2]\!]k &= [\![e_0]\!](\lambda b\kappa_0.[\![e_1]\!](\lambda v_1\kappa_1.[\![e_2]\!](\lambda v_2\kappa_2.k\ (\mathsf{ifp}(b)\,\mathsf{then}\,v_1\,\mathsf{else}\,v_2)\ k))) \\ [\![\mathsf{cwcc}\ e]\!]k &= [\![e]\!](\lambda f\kappa.f\ k\ k) \end{aligned}$$

but of course, we can always stick whatever we want in that second argument of the continuation calls, and we can in fact optimize this translation a bit by actually doing this inline eta-reduction of the k (rather than translating the terms as continuation terms): Here, $k \in$ IL1.val.

$$\begin{aligned} [\![n]\!]k &= k\ n\ \mathsf{halt} \\ [\![x]\!]k &= k\ x\ \mathsf{halt} \\ [\![\lambda x.e]\!]k &= k\ (\lambda xk'.[\![e]\!]\ k')\ \mathsf{halt} \\ [\![e_0\ e_1]\!]k &= [\![e_0]\!](\lambda f\kappa.[\![e_1]\!](\lambda v\kappa'.f\ v\ k)) \\ [\![(e_1,\cdots,e_n)]\!]k &= [\![e_1]\!](\lambda x_1\kappa_1.\cdots[\![e_n]\!](\lambda x_n\kappa_n.k\ (x_1,\cdots,x_n)\ \mathsf{halt})\cdots) \\ [\![\pi_n\ e]\!]k &= [\![e]\!](\lambda p\kappa.\,\mathsf{let}\,v = \#n\ p\,\mathsf{in}\,k\ v\ \mathsf{halt}) \\ [\![e_0 + e_1]\!]k &= [\![e_0]\!](\lambda x_0\kappa_0.[\![e_1]\!](\lambda x_1\kappa_1.\,\mathsf{let}\,n = x_0 + x_1\,\mathsf{in}\,k\ n\ \mathsf{halt})) \\ [\![\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2]\!]k &= [\![e_1]\!](\lambda x\kappa.[\![e_2]\!]\ k) \\ [\![\mathsf{ifp}(e_0)\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2]\!]k &= [\![e_0]\!](\lambda b\kappa_0.[\![e_1]\!](\lambda v_1\kappa_1.[\![e_2]\!](\lambda v_2\kappa_2.k\ (\mathsf{ifp}(b)\,\mathsf{then}\,v_1\,\mathsf{else}\,v_2)\ \mathsf{halt}))) \\ [\![\mathsf{cwcc}\ e]\!]k &= \mathsf{let}\,k' = k\,\mathsf{in}\,[\![e]\!](\lambda f\kappa.f\ k'\ k') \end{aligned}$$

this translation ensures that it's impossible for the value k to be exponentially "expanded".

[4] Here, we discard the κ because we're doing under the assumption that a continuation will never return, but returning is symbolized by calling that second continuation argument (as with the functions). Therefore, we never use it, and we can pass into it anything we want.

Closure Conversion

One crucial aspect of of the translation between IL1 and IL2 is the process of lifting all of the abstractions to the top of the program, but we cannot do this until we're guaranteed that your programs are closed. There are a few ways of doing this. One can do another CPS translation from IL1 to IL1 again to model environment lookups and extensions as their own continuations, but this seemed a bit too much work.

Another way is to use static analysis (similar to the Fvs function) to determine all of the variables of a program, call it $\text{VS}(c)$. Let $\langle\cdot\rangle$ be an enumeration of $\text{VS}(c)$: for example, suppose $\text{VS}(c) = \{x, y, z\}$, then $\langle\cdot\rangle$ could be the function

$$\langle\cdot\rangle = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$$

we can then use a tuple ρ as an environment and do a translation triple

$$\begin{aligned}
&\mathcal{V}[\![v]\!]\rho : e \\
&\mathcal{E}[\![e]\!]\rho : (\mathsf{var} \times e)\ \mathsf{list} \times e \\
&\mathcal{C}[\![e]\!]\rho : c
\end{aligned}$$

given as

$$\begin{aligned}
\mathcal{V}[\![n]\!]\rho &= n \\
\mathcal{V}[\![x]\!]\rho &= \pi_{\langle x\rangle}\rho \\
\mathcal{V}[\![\lambda x k.c]\!]\rho &= (p, \lambda y_{\langle x\rangle}\rho'.\, \mathsf{let}\, y_i = \pi_i\rho'\ (\forall y_i \neq y_{\langle x\rangle} \in \text{VS}(c))\, \mathsf{in}\, \mathsf{let}\, \rho'' = (y_i)\, \mathsf{in}\, \mathcal{C}[\![c]\!]\rho'' \\
&\vdots \\
\mathcal{C}[\![v_0\ v_1]\!]\rho &= \mathsf{let}\, fn = \mathcal{V}[\![v_0]\!]\rho, \rho' = \pi_1 fn, f = \pi_2 fn, v' = \mathcal{V}[\![v_1]\!]\rho\, \mathsf{in}\, f\ v'\ \rho' \\
&\vdots
\end{aligned}$$

However this generates an excessive amount of boilerplate code to update the environment. Therefore, an easier alternative is to use an idea of explicitly abstracting over all of the non-closed variables.

To do this, we will first define the target language of this closure conversion as IL1 with a slight modification. Let $\bar{\rho}$ be a sequence of variables $\rho_1, \cdots, \rho_n = \text{VS}(c)$, we can define the target language as

$$\begin{aligned}
v &::= n \mid x \mid \lambda x k \bar{\rho}.c \mid \mathsf{halt} \\
e &::= v \mid v_0 + v_1 \mid (v_i) \mid \pi_n v \mid \mathsf{ifp}(v_0)\, \mathsf{then}\, v_1\, \mathsf{else}\, v_2 \\
c &::= \mathsf{let}\, x = e\, \mathsf{in}\, c \mid v_0\ v_1\ v_2\ \bar{\rho}
\end{aligned}$$

with natural domains for the translations

$$\begin{aligned}
&\mathcal{V}[\![v]\!]\rho : v \\
&\mathcal{E}[\![e]\!]\rho : e \\
&\mathcal{C}[\![e]\!]\rho : c
\end{aligned}$$

given by

$$\mathcal{V}[\![n]\!] = n$$

$$
\begin{aligned}
\mathcal{V}[\![x]\!] &= \rho_x \\
\mathcal{V}[\![\mathsf{halt}]\!] &= \mathsf{halt} \\
\mathcal{V}[\![\lambda xk.c]\!] &= \lambda xk\bar{\rho}.\,\mathsf{let}\,\rho_{\langle x\rangle} = x, \rho_{\langle k\rangle} = k\,\mathsf{in}\,\mathcal{C}[\![c]\!] \\
\mathcal{E}[\![v]\!] &= \mathcal{V}[\![v]\!] \\
\mathcal{E}[\![v_0 + v_1]\!] &= \mathcal{V}[\![v_0]\!] + \mathcal{V}[\![v_1]\!] \\
\mathcal{E}[\![(v_i)]\!] &= (\mathcal{V}[\![v_i]\!]) \\
\mathcal{E}[\![\pi_n v]\!] &= \pi_n \mathcal{V}[\![v]\!] \\
\mathcal{E}[\![\mathsf{ifp}(v_0)\,\mathsf{then}\,v_1\,\mathsf{else}\,v_2]\!] &= \mathsf{ifp}(\mathcal{V}[\![v_0]\!])\,\mathsf{then}\,\mathcal{V}[\![v_1]\!]\,\mathsf{else}\,\mathcal{V}[\![v_2]\!] \\
\mathcal{C}[\![v_0\ v_1\ v_2]\!] &= \mathcal{V}[\![v_0]\!]\ \mathcal{V}[\![v_1]\!]\ \mathcal{V}[\![v_2]\!] \\
\mathcal{C}[\![\mathsf{let}\,x = e\,\mathsf{in}\,c]\!] &= \mathsf{let}\,\rho_{\langle x\rangle} = \mathcal{E}[\![e]\!]\,\mathsf{in}\,\mathcal{C}[\![c]\!]
\end{aligned}
$$

For example, suppose I have the command

$$c \triangleq \mathsf{let}\,f = \lambda xk.k\ (x+1)\ k\,\mathsf{in}\,f\ 4\ \mathsf{halt}$$

there are three variables here, x, k, f, so we can construct the injection:

$$\langle\cdot\rangle = \{x \mapsto 1, k \mapsto 2, f \mapsto 3\}$$

so

$$\bar{\rho} = \rho_1, \rho_2, \rho_3$$

then the above would translate into

$$\mathcal{C}[\![c]\!] = \mathsf{let}\,\rho_{\langle f\rangle} = \mathcal{E}[\![\lambda xk.k(x+1)k]\!]\,\mathsf{in}\,\mathcal{C}[\![f\ 4\ \mathsf{halt}]\!]$$

expand the $\mathcal{E}$ first

$$
\begin{aligned}
&= \mathsf{let}\,\rho_{\langle f\rangle} = \mathcal{V}[\![\lambda xk.k(x+1)k]\!]\,\mathsf{in}\,\mathcal{C}[\![f\ 4\ \mathsf{halt}]\!] \\
&= \mathsf{let}\,\rho_{\langle f\rangle} = \lambda xk\rho_1\rho_2\rho_3.\,\mathsf{let}\,\rho_{\langle x\rangle} = x, \rho_{\langle k\rangle} = k\,\mathsf{in}\,\mathcal{C}[\![k\ (x+1)\ k]\!]\,\mathsf{in}\,\mathcal{C}[\![f\ 4\ \mathsf{halt}]\!] \\
&\vdots \\
&= \mathsf{let}\,\rho_{\langle f\rangle} = \lambda xk\rho_1\rho_2\rho_3.\,\mathsf{let}\,\rho_{\langle x\rangle} = x, \rho_{\langle k\rangle} = k\,\mathsf{in}\,\rho_{\langle k\rangle}(\rho_{\langle x\rangle}+1)\rho_{\langle k\rangle}\rho_1\rho_2\rho_3\,\mathsf{in}\,\rho_{\langle f\rangle}\ 4\ \mathsf{halt}\ \rho_1\rho_2\rho_3 \\
&= \mathsf{let}\,\rho_3 = \lambda xk\rho_1\rho_2\rho_3.\,\mathsf{let}\,\rho_1 = x, \rho_2 = k\,\mathsf{in}\,\rho_2(\rho_1+1)\rho_2\rho_1\rho_2\rho_3\,\mathsf{in}\,\rho_3\ 4\ \mathsf{halt}\ \rho_1\rho_2\rho_3
\end{aligned}
$$

Lambda Hoisting

This is trivial, go through everything, and for each lambda encountered, take it out, name it, and replace the lambda with the new variable.

Value Lowering

Note that in the final IL2, expressions and applications only work on variables. In effect, it is the closed, lambda lifted IL1 with the restriction that the values are only variables. So we can just translate the hoisted IL1 into the following language

$$v ::= x$$

$$e ::= v \mid \mathsf{val}(n) \mid \mathsf{val}(\mathsf{halt}) \mid v_0 + v_1 \mid (v_i) \mid \pi_n v \mid \mathsf{ifp}(v_0)\,\mathsf{then}\,v_1\,\mathsf{else}\,v_2$$
$$c ::= \mathsf{let}\,x = e\,\mathsf{in}\,c \mid v_0\ v_1\ v_2\ \bar{\rho}$$

We want to define a set of "lowering" translation $\mathcal{LV}[\![v]\!], \mathcal{LE}[\![e]\!], \mathcal{LC}[\![c]\!]$ that binds all non-variable values (integers and halts) into their own variables. Therefore, we need to have both the value and the expressions translation be able to be abstracted as bindings.

$$\mathcal{LV}[\![v]\!] : (\mathsf{var} \times e)\mathsf{list} \times \mathsf{var}$$
$$\mathcal{LE}[\![e]\!] : (\mathsf{var} \times e)\mathsf{list} \times \mathsf{e}$$
$$\mathcal{LC}[\![c]\!] : c$$

We will use the notation

$$\mathsf{let}\,x_i = e_i\,\mathsf{in}\,x$$

to mean

$$([(x_i, e_i); \cdots (x_n, e_n)], x)$$

and respective syntax sugary for expressions. Our translation is given as

$$\begin{aligned}
\mathcal{LV}[\![n]\!] &= \mathsf{let}\,x = n\,\mathsf{in}\,x \\
\mathcal{LV}[\![x]\!] &= ([], x) \\
\mathcal{LV}[\![\mathsf{halt}]\!] &= \mathsf{let}\,x = \mathsf{halt}\,\mathsf{in}\,x \\
\mathcal{LE}[\![v_0 + v_1]\!] &= \mathsf{let}(l_0, x_0) = \mathcal{LV}[\![v_0]\!]\,\mathsf{in}\,\mathsf{let}(l_1, x_1) = \mathcal{LV}[\![v_1]\!]\,\mathsf{in}\,(l_0 \cup l_1, x_0 + x_1) \\
\mathcal{LE}[\![(v_0, \cdots, v_n)]\!] &= \mathsf{let}(l_0, x_0) = \mathcal{LV}[\![v_0]\!]\,\mathsf{in} \cdots \mathsf{let}(l_n, x_n) = \mathcal{LV}[\![v_n]\!]\,\mathsf{in}\,(\bigcup l_k, (x_1, \cdots, x_n)) \\
\mathcal{LE}[\![\pi_n v]\!] &= \mathsf{let}(l, x) = \mathcal{LV}[\![v]\!]\,\mathsf{in}\,(l, \pi_n x) \\
\mathcal{LC}[\![\mathsf{let}\,x = e\,\mathsf{in}\,c]\!] &= \mathsf{let}(x_i = e_i, e') = \mathcal{LE}[\![e]\!]\,\mathsf{in} \qquad \underbrace{\mathsf{let}\,x_i = e_i, x = e'\,\mathsf{in}\,\mathcal{LC}[\![c]\!]}_{\text{this is the actual command returned}} \\
\mathcal{LC}[\![v_0\ v_1\ v_2]\!] &= \mathsf{let}(x_i = e_i, x) = \mathcal{LV}[\![v_0]\!], (y_j = e'_j, y) = \mathcal{LV}[\![v_1]\!], (z_k = e''_k, z) = \mathcal{LV}[\![v_2]\!]\,\mathsf{in} \\
&\quad \mathsf{let}\,x_i = e_i, y_j = e'_j, z_k = e''_k\,\mathsf{in}\,x\ y\ z
\end{aligned}$$

After this, the translation to IL2 is trivial.