Type Safety and Erasure Proofs for "A Type System for Coordinated Data Structures"

Michael F. Ringenburg Dan Grossman
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{miker,djg}@cs.washington.edu

Abstract

We prove the Type Safety and Erasure Theorems presented in Section 4 of Ringenburg and Grossman's paper "A Type System for Coordinated Data Structures" [1]. We also remind the reader of the syntax, semantics, and typing rules for the coordinated list language described in Section 3 of the same paper. We refer the reader to the original paper for a detailed presentation of the coordinated data structure type system.

1 The Language

Figures 1, 2, and 3 present, respectively, the syntax, semantics, and typing rules for our coordinated list language. We implicitly assume Δ and Γ do not have repeated elements. For example, $\Delta, \alpha{:}\kappa$ is ill-formed if $\alpha \in \mathrm{Dom}(\Delta)$. To avoid conflicts, we can systematically rename constructs with binding occurrences. We therefore treat Δ and Γ as partial functions. All explicit occurrences of α and x in the grammar are binding (except when they constitute the entire type or expression, of course). Substitution is defined as usual.

2 Type Safety

Lemma 1 (Weakening)

1. *If* $\Delta \vdash_{\mathrm{k}} \tau : \kappa$*, then* $\Delta\Delta' \vdash_{\mathrm{k}} \tau : \kappa$*.*
2. *If* $\Delta \vdash_{\mathrm{c}} \Gamma$*, then* $\Delta\Delta' \vdash_{\mathrm{c}} \Gamma$*.*
3. *If* $\Delta;\Gamma \vdash_{\mathrm{t}} e : \tau$ *and* $\Delta\Delta' \vdash_{\mathrm{c}} \Gamma\Gamma'$*, then* $\Delta\Delta';\Gamma\Gamma' \vdash_{\mathrm{t}} e : \tau$*.*

Proof:

1. By induction on the assumed derivation, treating typing contexts as partial functions: If the last rule in the derivation is KBASE, the result is immediate. The other cases follow from induction. (Cases KQUANT and KMU reorder typing contexts and use implicit α-conversion to ensure the bound type variables do not occur in Δ'.)
2. By induction on the assumed derivation: Case CEMPTY is immediate, and case CCONS follows from induction and the previous lemma.
3. By induction on the assumed derivation, treating type and term contexts as partial functions: Case BASE is immediate. Cases PAIR, PROJ, CASE, APP, FIX, TFUN, and UNROLL follow from induction and the previous lemmas. (Cases CASE, FUN, and TFUN reorder contexts and use implicit α-conversion to ensure bound variables or type variables do not occur in Γ' or Δ'.) The remaining cases follow from induction and Weakening Lemma 1, again using reordering and α-conversion as necessary.

VARIABLES	x	$\in$	Var
TYPE VARIABLES	α, β	$\in$	Tyvar
KINDS	κ	::=	$\mathrm{T} \mid \mathrm{L}$
TYPES	σ, τ	::=	$1 \mid \alpha \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \mu(\sigma \leftarrow \beta)\alpha.\tau \mid \tau^* \mid \tau{::}\sigma$
EXPRESSIONS	e	::=	$() \mid x \mid (e, e) \mid \pi_i\ e \mid \mathsf{in}_i\ e \mid \mathsf{case}\ e\ \mathsf{of}\ x.e\ x.e \mid \lambda x{:}\tau.\ e \mid e\ e \mid \mathsf{fix}\ e$
		$\mid$	$\Lambda\alpha{:}\kappa.\ e \mid e\ [\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ e\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e$
		$\mid$	$\mathsf{roll}\ e\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ e \mid \mathsf{peel}\ e\ \mathsf{as}\ \alpha, \alpha, x\ \mathsf{in}\ e$
VALUES	v	::=	$() \mid (v, v) \mid \mathsf{in}_i\ v \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha{:}\kappa.\ e \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{roll}\ v\ \mathsf{as}\ \tau$
TYPE CONTEXTS	Γ	::=	$\cdot \mid \Gamma, x{:}\tau$
KIND CONTEXTS	Δ	::=	$\cdot \mid \Delta, \alpha{:}\kappa$

Figure 1: The syntax for a simple language that includes our extensions and supports coordinated lists.

Lemma 2 (Commuting Substitutions)

If β does not occur free in τ_2 (and $\alpha \neq \beta$), then $\tau_0[\tau_1/\beta][\tau_2/\alpha] = \tau_0[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$

Proof: By induction on the structure of τ_0: If $\tau_0 = 1$, then both types are 1. If $\tau_0 = \alpha$, then both types are τ_2 (using that β is not free in τ_2). If $\tau_0 = \beta$, then both types are $\tau_1[\tau_2/\alpha]$. The remaining cases follow from induction and the definition of substitution.

Lemma 3 (Substitution)

1. *If $\Delta, \alpha{:}\kappa' \vdash_{\mathrm{k}} \tau : \kappa$ and $\Delta \vdash_{\mathrm{k}} \tau' : \kappa'$, then $\Delta \vdash_{\mathrm{k}} \tau[\tau'/\alpha] : \kappa$*
2. *If $\Delta, \alpha{:}\kappa' \vdash_{\mathrm{c}} \Gamma$ and $\Delta \vdash_{\mathrm{k}} \tau' : \kappa'$, then $\Delta \vdash_{\mathrm{c}} \Gamma[\tau'/\alpha]$.*
3. *If $\Delta, \alpha{:}\kappa'; \Gamma \vdash_{\mathrm{t}} e : \tau$ and $\Delta \vdash_{\mathrm{k}} \tau' : \kappa'$, then $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} e[\tau'/\alpha] : \tau[\tau'/\alpha]$.*
4. *If $\Delta; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e : \tau$ and $\Delta; \Gamma \vdash_{\mathrm{t}} e' : \tau'$, then $\Delta; \Gamma \vdash_{\mathrm{t}} e[e'/x] : \tau$.*

Proof:

1. By induction on the derivation of $\Delta, \alpha{:}\kappa' \vdash_{\mathrm{k}} \tau : \kappa$: Case KBASE is trivial if $\tau = 1$, follows from the definition of Δ if τ is a type variable other than α, and follows from $\Delta \vdash_{\mathrm{k}} \tau' : \kappa'$ if $\tau = \alpha$. Cases KPAIR, KSTAR, and KCONS follow from induction and the definition of substitution. Cases KQUANT and KMU follow from induction, the definition of substitution, implicit reordering of typing contexts, and Weakening Lemma 1 (to show that τ' still has kind κ' under the context extended with the type variable(s) bound by μ, $\exists$, or $\forall$).

2. By induction on the derivation of $\Delta, \alpha{:}\kappa' \vdash_{\mathrm{c}} \Gamma$, using the previous lemma

3. By induction on the derivation of $\Delta, \alpha{:}\kappa'; \Gamma \vdash_{\mathrm{t}} e : \tau$: Case BASE follows from the previous lemma. Cases PAIR, PROJ, INJECT, CASE, FUN, APP, FIX, and UNROLL follow from induction. Case TFUN follows from induction and Weakening Lemma 1 (to show τ' has kind κ' under the context extended with the type variable bound by Λ). Cases UNPACK and PEEL follow from induction, Weakening Lemma 1 (as in case TFUN), and Substitution Lemma 1 (for the kind of the result type, $\tau[\tau'/\alpha]$).

   Now consider the case $e = e_1\ [\tau_1]$ (TAPP). By the typing rules, $\Delta, \alpha{:}\kappa'; \Gamma \vdash_{\mathrm{t}} e_1\ [\tau_1] : \tau_2[\tau_1/\alpha_1]$, where $\Delta, \alpha{:}\kappa'; \Gamma \vdash_{\mathrm{t}} e_1 : \forall\alpha_1{:}\kappa_1.\tau_2$ and $\Delta, \alpha{:}\kappa' \vdash_{\mathrm{k}} \tau_1 : \kappa_1$. We also know that $\Delta \vdash_{\mathrm{k}} \tau' : \kappa'$. Thus by Substitution Lemma 1, we have $\Delta \vdash_{\mathrm{k}} \tau_1[\tau'/\alpha] : \kappa_1$. By induction, $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} e_1[\tau'/\alpha] : (\forall\alpha_1{:}\kappa_1.\tau_2)[\tau'/\alpha]$. Because of α-renaming, we can assume $\alpha \neq \alpha_1$. Thus, by the definition of substitution, $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} e_1[\tau'/\alpha] : \forall\alpha_1{:}\kappa_1.(\tau_2[\tau'/\alpha])$. By the TAPP typing rule, we now have $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} e_1[\tau'/\alpha]\ [\tau_1[\tau'/\alpha]] : \tau_2[\tau'/\alpha][\tau_1[\tau'/\alpha]/\alpha_1]$. By the definition of substitution, this is equivalent to $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} (e_1\ [\tau_1])[\tau'/\alpha] : \tau_2[\tau'/\alpha][\tau_1[\tau'/\alpha]/\alpha_1]$. By the Commuting Substitutions Lemma, we have $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} (e_1\ [\tau_1])[\tau'/\alpha] : \tau_2[\tau_1/\alpha_1][\tau'/\alpha]$. Thus the lemma holds for this case. Cases PACK and ROLL follow by the same logic.

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid (E, e) \mid (v, E) \mid \pi_i\ E \mid \mathsf{in}_i\ E \mid \mathsf{case}\ E\ \mathsf{of}\ x.e\ x.e \mid E\ e \mid v\ E \mid \mathsf{fix}\ E \\
& \mid & E\ [\tau] \mid \mathsf{pack}\ \tau, E\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ E\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e \\
& \mid & \mathsf{roll}\ E\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ E \mid \mathsf{peel}\ E\ \mathsf{as}\ \alpha, \beta, x\ \mathsf{in}\ e
\end{array}
$$

$$
\begin{array}{rcll}
\pi_i\ (v_1, v_2) & \xrightarrow{\mathrm{r}} & v_i & \text{where } i \in \{1,2\} \\
\mathsf{case}\ \mathsf{in}_i\ v\ \mathsf{of}\ x.e_1\ x.e_2 & \xrightarrow{\mathrm{r}} & e_i[v/x] & \text{where } i \in \{1,2\} \\
(\lambda x{:}\tau.\ e)\ v & \xrightarrow{\mathrm{r}} & e[v/x] & \\
\mathsf{fix}\ \lambda x{:}\tau.\ e & \xrightarrow{\mathrm{r}} & e[(\mathsf{fix}\ \lambda x{:}\tau.\ e)/x] & \\
(\Lambda\alpha{:}\kappa.\ e)\ [\tau] & \xrightarrow{\mathrm{r}} & e[\tau/\alpha] & \\
\mathsf{unpack}\ (\mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_2)\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e & \xrightarrow{\mathrm{r}} & e[\tau_1/\alpha][v/x] & \\
\mathsf{unroll}\ \mathsf{roll}\ v\ \mathsf{as}\ \tau & \xrightarrow{\mathrm{r}} & v &
\end{array}
$$

$$
\begin{array}{l}
\mathsf{peel}\ (\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e \\
\xrightarrow{\mathrm{r}} \quad e[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}][(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_2)/x] \\
\text{where } \mathrm{peel}(\sigma) = \tau'{::}\sigma'
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{peel}(\tau{::}\sigma) & = & \tau{::}\sigma \\
\mathrm{peel}(\tau^*) & = & \tau{::}\tau^*
\end{array}
\qquad\qquad
\frac{e \xrightarrow{\mathrm{r}} e'}{E[e] \rightarrow E[e']}
$$

Figure 2: The operational semantics for our simple, coordinated list language.

4. By induction on the derivation of $\Delta; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e : \tau$: Case BASE follows trivially if $e = ()$ or $e = y$ and $y \neq x$. If $e = x$, then $\Delta; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e : \tau'$, and $e[e'/x] = e'$. Thus $\Delta; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e[e'/x] : \tau'$ and so the lemma holds for this case. Cases PAIR, PROJ, INJECT, CASE, FUN, APP, FIX, TAPP, PACK, ROLL, and UNROLL follow directly from induction, the typing rules, and the definition of substitution. Note that we can assume that all conflicts between the substitution target variable x and any bound variables α_i are resolved by α-renaming (e.g, in the FUN and CASE cases). Cases TFUN, UNPACK, and PEEL follow from induction, the typing rules, the definition of substitution, and Weakening Lemma 3 (to show that e' still has type τ' under the context extended with the bound type variables).

Lemma 4 (Canonical Forms)

1. *If $\cdot \vdash_{\mathrm{k}} \tau : \mathrm{L}$, then τ has the form $\tau_1{}^*$ or the form $\tau_1{::}\tau_2$.*
2. *If $\cdot; \cdot \vdash_{\mathrm{t}} v : \tau_1 \times \tau_2$, then v has the form (v_1, v_2).*
3. *If $\cdot; \cdot \vdash_{\mathrm{t}} v : \tau_1 + \tau_2$, then v has the form $\mathsf{in}_i\ v_1$ and $i \in \{1,2\}$.*
4. *If $\cdot; \cdot \vdash_{\mathrm{t}} v : \tau_1 \rightarrow \tau_2$, then v has the form $\lambda x{:}\tau_1.\ e$.*
5. *If $\cdot; \cdot \vdash_{\mathrm{t}} v : \forall\alpha{:}\kappa.\tau_1$, then v has the form $\Lambda\alpha{:}\kappa.\ e$.*
6. *If $\cdot; \cdot \vdash_{\mathrm{t}} v : \mu(\sigma \leftarrow \beta)\alpha.\tau_1$, then v has the form $\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1$.*
7. *If $\cdot; \cdot \vdash_{\mathrm{t}} v : \exists\alpha{:}\kappa.\tau_1$, then v has the form $\mathsf{pack}\ \tau_2, v_1\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_1$.*

Proof: The first proof is by inspection of the rules for $\Delta \vdash_{\mathrm{k}} \tau : \kappa$: when $\Delta = \cdot$ and $\kappa = \mathrm{L}$, only KSTAR and KCONS apply. The other proofs are by inspection of the rules for $\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau$: When $\Delta = \cdot$, $\Gamma = \cdot$, e is a value, and τ has the form required by the lemma, only one rule applies. (The rule is one of PAIR, INJECT, FUN, TFUN, ROLL, or PACK.)

Lemma 5 (Subexpression Type Invariance)

If $\cdot; \cdot \vdash_{\mathrm{t}} E[e] : \tau$, $\cdot; \cdot \vdash_{\mathrm{t}} e : \tau'$, and $\cdot; \cdot \vdash_{\mathrm{t}} e' : \tau'$, then $\cdot; \cdot \vdash_{\mathrm{t}} E[e'] : \tau$.

$\boxed{\Delta \vdash_{\mathrm{k}} \tau : \kappa}$

KBASE
$$\frac{}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} 1 : \mathrm{T} \\ \Delta \vdash_{\mathrm{k}} \alpha : \Delta(\alpha)\end{array}}$$

KPAIR
$$\frac{\Delta \vdash_{\mathrm{k}} \tau_1 : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \tau_2 : \mathrm{T}}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \tau_1 \times \tau_2 : \mathrm{T} \\ \Delta \vdash_{\mathrm{k}} \tau_1 + \tau_2 : \mathrm{T} \\ \Delta \vdash_{\mathrm{k}} \tau_1 \to \tau_2 : \mathrm{T}\end{array}}$$

KQUANT
$$\frac{\Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \forall \alpha{:}\kappa.\tau : \mathrm{T} \\ \Delta \vdash_{\mathrm{k}} \exists \alpha{:}\kappa.\tau : \mathrm{T}\end{array}}$$

KMU
$$\frac{\Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L} \qquad \Delta, \alpha{:}\mathrm{T}, \beta{:}\mathrm{T} \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \mu(\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}}$$

KSTAR
$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \tau^* : \mathrm{L}}$$

KCONS
$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L}}{\Delta \vdash_{\mathrm{k}} \tau{::}\sigma : \mathrm{L}}$$

$\boxed{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau}$

BASE
$$\frac{}{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{t}} () : 1 \\ \Delta; \Gamma \vdash_{\mathrm{t}} x : \Gamma(x)\end{array}}$$

PAIR
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \tau_1 \qquad \Delta; \Gamma \vdash_{\mathrm{t}} e_2 : \tau_2}{\Delta; \Gamma \vdash_{\mathrm{t}} (e_1, e_2) : \tau_1 \times \tau_2}$$

PROJ
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau_1 \times \tau_2 \qquad i \in \{1, 2\}}{\Delta; \Gamma \vdash_{\mathrm{t}} \pi_i\ e : \tau_i}$$

INJECT
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau \qquad \Delta \vdash_{\mathrm{k}} \tau' : \mathrm{T}}{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{in}_1\ e : \tau + \tau' \\ \Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{in}_2\ e : \tau' + \tau\end{array}}$$

CASE
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau_1 + \tau_2 \qquad \Delta; \Gamma, x{:}\tau_1 \vdash_{\mathrm{t}} e_1 : \tau \qquad \Delta; \Gamma, x{:}\tau_2 \vdash_{\mathrm{t}} e_2 : \tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{case}\ e\ \mathsf{of}\ x.e_1\ x.e_2 : \tau}$$

FUN
$$\frac{\Delta; \Gamma, x{:}\tau \vdash_{\mathrm{t}} e : \tau' \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \lambda x{:}\tau.\ e : \tau \to \tau'}$$

APP
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \tau' \to \tau \qquad \Delta; \Gamma \vdash_{\mathrm{t}} e_2 : \tau'}{\Delta; \Gamma \vdash_{\mathrm{t}} e_1\ e_2 : \tau}$$

FIX
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau \to \tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{fix}\ e : \tau}$$

TFUN
$$\frac{\Delta, \alpha{:}\kappa; \Gamma \vdash_{\mathrm{t}} e : \tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \Lambda\alpha{:}\kappa.\ e : \forall \alpha{:}\kappa.\tau}$$

TAPP
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \forall \alpha{:}\kappa.\tau' \qquad \Delta \vdash_{\mathrm{k}} \tau : \kappa}{\Delta; \Gamma \vdash_{\mathrm{t}} e\ [\tau] : \tau'[\tau/\alpha]}$$

PACK
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau'[\tau/\alpha] \qquad \Delta \vdash_{\mathrm{k}} \tau : \kappa \qquad \Delta \vdash_{\mathrm{k}} \exists \alpha{:}\kappa.\tau' : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau, e\ \mathsf{as}\ \exists \alpha{:}\kappa.\tau' : \exists \alpha{:}\kappa.\tau'}$$

UNPACK
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \exists \alpha{:}\kappa.\tau' \qquad \Delta, \alpha{:}\kappa; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e_2 : \tau \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{unpack}\ e_1\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2 : \tau}$$

ROLL
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau[\tau'/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau/\alpha] \qquad \Delta \vdash_{\mathrm{k}} \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{roll}\ e\ \mathsf{as}\ \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau : \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau}$$

UNROLL
$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{unroll}\ e : \tau[\tau'/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau/\alpha]}$$

PEEL
$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : (\mu(\sigma \leftarrow \beta)\alpha.\tau_1) \times (\mu(\sigma \leftarrow \beta)\alpha.\tau_2) \\ \Delta, \alpha_{hd}{:}\mathrm{T}, \alpha_{tl}{:}\mathrm{L}; \Gamma, x{:}(\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_1) \times (\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_2) \vdash_{\mathrm{t}} e_2 : \tau \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}\end{array}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{peel}\ e_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2 : \tau}$$

Figure 3: The typing rules for our coordinated list language.

Proof: By induction on the structure of E, proceeding by cases. All cases follow directly by the typing rules and the inductive hypothesis. For instance, if $E = (E_1, e_2)$, then $E[e] = (E_1[e], e_2)$ and $E[e'] = (E_1[e'], e_2)$. By inversion, if $\cdot;\cdot \vdash_\tau E[e] : \tau_1 \times \tau_2$ then $\cdot;\cdot \vdash_\tau E_1[e] : \tau_1$. By induction, $\cdot;\cdot \vdash_\tau E_1[e'] : \tau_1$. Thus, $\cdot;\cdot \vdash_\tau E[e'] : \tau_1 \times \tau_2$. The other cases proceed identically.

Lemma 6 (Preservation)

1. *If* $\cdot;\cdot \vdash_\tau e : \tau$ *and* $e \xrightarrow{\mathrm{r}} e'$*, then* $\cdot;\cdot \vdash_\tau e' : \tau$.
2. *If* $\cdot;\cdot \vdash_\tau e : \tau$ *and* $e \to e'$*, then* $\cdot;\cdot \vdash_\tau e' : \tau$.

Proof:

1. We consider each of the $e \xrightarrow{\mathrm{r}} e'$ rules in turn:

   - **Case** $\pi_i\ (v_1, v_2) \xrightarrow{\mathrm{r}} v_i$**:** By the PROJ typing rule, $\cdot;\cdot \vdash_\tau \pi_i\ (v_1, v_2) : \tau_i$ where $\cdot;\cdot \vdash_\tau (v_1, v_2) : \tau_1 \times \tau_2$. Inversion of $\cdot;\cdot \vdash_\tau (v_1, v_2) : \tau_1 \times \tau_2$ ensures $\cdot;\cdot \vdash_\tau v_i : \tau_i$.
   - **Case** $\mathsf{case\ in}_i\ v\ \mathsf{of}\ x.e_1\ x.e_2 \xrightarrow{\mathrm{r}} e_i[v/x]$**:** By the CASE typing rule, $\cdot;\cdot \vdash_\tau \mathsf{case\ in}_i\ v\ \mathsf{of}\ x.e_1\ x.e_2 : \tau$ where $\cdot;\cdot \vdash_\tau \mathsf{in}_i\ v : \tau_1 + \tau_2$, $\cdot;\cdot, x{:}\tau_1 \vdash_\tau e_1 : \tau$, and $\cdot;\cdot, x{:}\tau_2 \vdash_\tau e_2 : \tau$. Inversion of $\cdot;\cdot \vdash_\tau \mathsf{in}_i\ v : \tau_1 + \tau_2$ ensures $\cdot;\cdot \vdash_\tau v : \tau_i$. Thus, by Substitution Lemma 4, $\cdot;\cdot \vdash_\tau e_i[v/x] : \tau$.
   - **Case** $(\lambda x{:}\tau'.\ e)\ v \xrightarrow{\mathrm{r}} e[v/x]$**:** By the APP typing rule, $\cdot;\cdot \vdash_\tau (\lambda x{:}\tau'.\ e)\ v : \tau$ where $\cdot;\cdot \vdash_\tau \lambda x{:}\tau'.\ e : \tau' \to \tau$, and $\cdot;\cdot \vdash_\tau v : \tau'$. Inversion of $\cdot;\cdot \vdash_\tau \lambda x{:}\tau'.\ e : \tau' \to \tau$ ensures $\cdot;\cdot, x{:}\tau' \vdash_\tau e : \tau$. Thus, by Substitution Lemma 4, $\cdot;\cdot \vdash_\tau e[v/x] : \tau$.
   - **Case** $\mathsf{fix}\ \lambda x{:}\tau.\ e \xrightarrow{\mathrm{r}} e[(\mathsf{fix}\ \lambda x{:}\tau.\ e)/x]$**:** By the FIX typing rule, $\cdot;\cdot \vdash_\tau \mathsf{fix}\ \lambda x{:}\tau.\ e : \tau$ where $\cdot;\cdot \vdash_\tau \lambda x{:}\tau.\ e : \tau \to \tau$. Inversion of $\cdot;\cdot \vdash_\tau \lambda x{:}\tau.\ e : \tau \to \tau$ ensures $\cdot;\cdot, x{:}\tau \vdash_\tau e : \tau$. Thus, by Substitution Lemma 4, $\cdot;\cdot \vdash_\tau e[(\mathsf{fix}\ \lambda x{:}\tau.\ e)/x] : \tau$.
   - **Case** $(\Lambda\alpha{:}\kappa.\ e)\ [\tau] \xrightarrow{\mathrm{r}} e[\tau/\alpha]$**:** By the TAPP typing rule, $\cdot;\cdot \vdash_\tau (\Lambda\alpha{:}\kappa.\ e)\ [\tau] : \tau'[\tau/\alpha]$ where $\cdot;\cdot \vdash_\tau \Lambda\alpha{:}\kappa.\ e : \forall\alpha{:}\kappa.\tau'$ and $\cdot \vdash_\kappa \tau : \kappa$. Inversion of $\cdot;\cdot \vdash_\tau \Lambda\alpha{:}\kappa.\ e : \forall\alpha{:}\kappa.\tau'$ ensures $\cdot, \alpha{:}\kappa;\cdot \vdash_\tau e : \tau'$. Therefore, by Substitution Lemma 3, $\cdot;\cdot \vdash_\tau e[\tau/\alpha] : \tau'[\tau/\alpha]$.
   - **Case** $\mathsf{unpack}\ (\mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_2)\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e \xrightarrow{\mathrm{r}} e[\tau_1/\alpha][v/x]$**:** By the UNPACK and PACK typing rules, $\cdot;\cdot \vdash_\tau \mathsf{unpack}\ (\mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_2)\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e : \tau$ where $\cdot;\cdot \vdash_\tau \mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_2 : \exists\alpha{:}\kappa.\tau_2$, $\cdot, \alpha{:}\kappa;\cdot, x{:}\tau_2 \vdash_\tau e : \tau$, and $\cdot \vdash_\kappa \tau : \mathrm{T}$. Inversion of $\cdot;\cdot \vdash_\tau \mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_2 : \exists\alpha{:}\kappa.\tau_2$ ensures $\cdot;\cdot \vdash_\tau v : \tau_2[\tau_1/\alpha]$ and $\cdot \vdash_\kappa \tau_1 : \kappa$. Therefore, by Substitution Lemma 3, $\cdot;\cdot, x{:}\tau_2[\tau_1/\alpha] \vdash_\tau e[\tau_1/\alpha] : \tau[\tau_1/\alpha]$. By Substitution Lemma 4, $\cdot;\cdot \vdash_\tau e[\tau_1/\alpha][v/x] : \tau[\tau_1/\alpha]$. Because $\cdot \vdash_\kappa \tau : \mathrm{T}$, we know that α does not appear free in τ. Therefore $\cdot;\cdot \vdash_\tau e[\tau_1/\alpha][v/x] : \tau$.
   - **Case** $\mathsf{unroll\ roll}\ v\ \mathsf{as}\ \tau \xrightarrow{\mathrm{r}} v$**:** By the UNROLL and ROLL typing rules,

     $$\cdot;\cdot \vdash_\tau \mathsf{unroll\ roll}\ v\ \mathsf{as}\ \tau : \tau_1[\tau_2/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau_1/\alpha]$$

     where $\tau = \mu(\tau_2{::}\sigma \leftarrow \beta)\alpha.\tau_1$ and $\cdot;\cdot \vdash_\tau \mathsf{roll}\ v\ \mathsf{as}\ \tau : \tau$. Inversion of $\cdot;\cdot \vdash_\tau \mathsf{roll}\ v\ \mathsf{as}\ \tau : \tau$ ensures $\cdot;\cdot \vdash_\tau v : \tau_1[\tau_2/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau_1/\alpha]$.
   - **Case** $e = \mathsf{peel}\ (\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_1 \xrightarrow{\mathrm{r}} e'$ **where**

     $$e' = e_1[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}][(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_2)/x]$$

     and $\mathrm{peel}(\sigma) = \tau'{::}\sigma'$**:** By the PEEL typing rule, $\cdot;\cdot \vdash_\tau e : \tau$ where $\cdot \vdash_\kappa \tau : \mathrm{T}$,

     $$\cdot;\cdot \vdash_\tau (\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2) : \mu(\sigma \leftarrow \beta)\alpha.\tau_1 \times \mu(\sigma \leftarrow \beta)\alpha.\tau_2\ ,$$

     and

     $$\cdot, \alpha_{hd}{:}\mathrm{T}, \alpha_{tl}{:}\mathrm{L};\cdot, x{:}\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_1 \times \mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_2 \vdash_\tau e_1 : \tau\ .$$

     By inversion of the KCONS and KSTAR rules, $\cdot \vdash_\kappa \tau' : \mathrm{T}$ and $\cdot \vdash_\kappa \sigma' : \mathrm{L}$. Because α_{hd} and α_{tl} do not appear free in τ, τ_1, or τ_2, we can apply Substitution Lemma 3 twice and Substitution Lemma 4 once to the typing expression for e_1 above, and obtain $\cdot;\cdot \vdash_\tau e' : \tau$

2. Follows directly by Preservation Lemma 1, the operational semantics, and the Subexpression Type Invariance Lemma.

Lemma 7 (Progress)

If $\cdot;\cdot \vdash_\tau e : \tau$ *then* e *is a value or there exists an* e' *such that* $e \to e'$.

Proof: By definition of $e \to e'$, it suffices to prove the following: If $\cdot;\cdot \vdash_\tau e : \tau$ and e is not a value then there exists an E, e_r, and e'_r such that $e = E[e_r]$ and $e_r \xrightarrow{r} e'_r$. The proof is by induction on the structure of e:

- If e is $()$, the result holds vacuously because e is a value.
- If e is x, the result holds vacuously because $\cdot;\cdot \vdash_\tau x : \tau$ is impossible.
- If e is (e_1, e_2), then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_1$ and $\cdot;\cdot \vdash_\tau e_2 : \tau_2$ where $\tau = \tau_1 \times \tau_2$. By induction, if e_1 is not a value, then there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = (E_1[e_r], e_2)$, so letting $E = (E_1, e_2)$ suffices. Similarly, if e_1 is a value and e_2 is not a value, induction ensures there exist E_1 and e_r such that letting $E = (v, E_1)$ suffices. Else e_1 and e_2 are both values, so e is value and the result holds vacuously.
- If e is $\pi_i\ e_1$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_1 \times \tau_2$, $i \in \{1,2\}$ and $\tau = \tau_i$. By induction, if e_1 is not a value, there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = \pi_i\ E_1[e_r]$, so letting $E = \pi_i\ E_1$ suffices. Else e_1 is a value and $\cdot;\cdot \vdash_\tau e_1 : \tau_1 \times \tau_2$ ensures $e_1 = (v_1, v_2)$ for some v_1 and v_2. Therefore, $e \xrightarrow{r} v_i$ so letting $E = [\cdot]$ suffices.
- If e is $\mathsf{in}_i\ e_1$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_i$, $i \in \{1,2\}$ where $\tau = \tau_1 + \tau_2$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = \mathsf{in}_i\ E_1[e_r]$, so letting $E = \mathsf{in}_i\ E_1$ suffices. Otherwise, if e_1 is a value then e is a value, and so the result holds vacuously.
- If e is $\mathsf{case}\ e_1\ \mathsf{of}\ x.e_2\ x.e_3$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_1 + \tau_2$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = \mathsf{case}\ E_1[e_r]\ \mathsf{of}\ x.e_2\ x.e_3$, so letting $E = \mathsf{case}\ E_1\ \mathsf{of}\ x.e_2\ x.e_3$ suffices. Otherwise, if e_1 is a value then the Canonical Forms Lemma ensures that e_1 has the form $\mathsf{in}_i\ v$, $i \in \{1,2\}$ for some value v. Thus $e \xrightarrow{r} e_i[v/x]$, so letting $E = [\cdot]$ suffices.
- If e is $\lambda x{:}\tau_1.\ e_1$, the result holds vacuously because e is a value.
- If e is $e_1\ e_2$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_1 \to \tau_2$ and $\cdot;\cdot \vdash_\tau e_2 : \tau_1$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = E_1[e_r]\ e_2$, so letting $E = E_1\ e_2$ suffices. If e_1 is a value v and e_2 is not a value, then by induction there exists E_2 and e_r such that $e_2 = E_2[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = v\ E_2[e_r]$, so letting $E = v\ E_2$ suffices. Otherwise, if e_1 and e_2 are values then the Canonical Forms Lemma ensures that e_1 has the form $\lambda x{:}\tau_1.\ e_3$. Thus $e \xrightarrow{r} e_3[e_2/x]$, so letting $E = [\cdot]$ suffices.
- If e is $\mathsf{fix}\ e_1$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau \to \tau$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = \mathsf{fix}\ E_1[e_r]$, so letting $E = \mathsf{fix}\ E_1$ suffices. Otherwise, if e_1 is a value then the Canonical Forms Lemma ensures that e_1 has the form $\lambda x{:}\tau.\ e_2$. Thus $e \xrightarrow{r} e_2[\mathsf{fix}\ \lambda x{:}\tau.\ e_2/x]$, so letting $E = [\cdot]$ suffices.
- If e is $\Lambda\alpha{:}\kappa.\ e_1$, the result holds vacuously because e is a value.
- If e is $e_1\ [\tau_1]$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \forall\alpha{:}\kappa.\tau_2$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = E_1[e_r]\ [\tau_1]$, so letting $E = E_1\ [\tau_1]$ suffices. Otherwise, if e_1 is a value then the Canonical Forms Lemma ensures that e_1 has the form $\Lambda\alpha{:}\kappa.\ e_2$. Thus $e \xrightarrow{r} e_2[\tau_1/\alpha]$, so letting $E = [\cdot]$ suffices.
- If e is $\mathsf{pack}\ \tau_1, e_1\ \mathsf{as}\ \tau_2$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_3[\tau_1/\alpha]$, where $\tau_2 = \exists\alpha{:}\kappa.\tau_3$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e'_r$. Then $e = \mathsf{pack}\ \tau_1, E_1[e_r]\ \mathsf{as}\ \tau_2$, so letting $E = \mathsf{pack}\ \tau_1, E_1\ \mathsf{as}\ \tau_2$ suffices. Otherwise, if e_1 is a value then e is a value, and so the result holds vacuously.

- If e is $\mathsf{unpack}\ e_1\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \exists\alpha{:}\kappa.\tau_1$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e_r'$. Then $e = \mathsf{unpack}\ E_1[e_r]\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2$, so letting $E = \mathsf{unpack}\ E_1\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2$ suffices. Otherwise, if e_1 is a value then the Canonical Forms Lemma ensures that e_1 has the form $\mathsf{pack}\ \tau_2, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_1$. Thus $e \xrightarrow{r} e_2[\tau_2/\alpha][v/x]$, so letting $E = [\cdot]$ suffices.

- If e is $\mathsf{roll}\ e_1\ \mathsf{as}\ \tau_1$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \tau_2[\tau_3/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau_2/\alpha]$, where $\tau_1 = \mu(\tau_3{::}\sigma \leftarrow \beta)\alpha.\tau_2$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e_r'$. Then $e = \mathsf{roll}\ E_1[e_r]\ \mathsf{as}\ \tau_1$, so letting $E = \mathsf{roll}\ E_1\ \mathsf{as}\ \tau_1$ suffices. Otherwise, if e_1 is a value then e is a value, and so the result holds vacuously.

- If e is $\mathsf{unroll}\ e_1$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \mu(\tau_2{::}\sigma \leftarrow \beta)\alpha.\tau_1$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e_r'$. Then $e = \mathsf{unroll}\ E_1[e_r]$, so letting $E = \mathsf{unroll}\ E_1$ suffices. Otherwise, if e_1 is a value then the Canonical Forms Lemma ensures that e_1 has the form $\mathsf{roll}\ v\ \mathsf{as}\ \mu(\tau_2{::}\sigma \leftarrow \beta)\alpha.\tau_1$. Thus $e \xrightarrow{r} v$, so letting $E = [\cdot]$ suffices.

- If e is $\mathsf{peel}\ e_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$, then inverting $\cdot;\cdot \vdash_\tau e : \tau$ ensures $\cdot;\cdot \vdash_\tau e_1 : \mu(\sigma \leftarrow \beta)\alpha.\tau_1 \times \mu(\sigma \leftarrow \beta)\alpha.\tau_2$. If e_1 is not a value, then by induction there exists E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{r} e_r'$. Then $e = \mathsf{peel}\ E_1[e_r]\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$, so letting $E = \mathsf{peel}\ E_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$ suffices. Otherwise, if e_1 is a value then the Canonical Forms Lemma (parts 2 and 6) and inversion of the PAIR rule ensures that e_1 has the form $(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)$. Thus

  $$e \xrightarrow{r} e_2[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}][(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_2)/x]\,,$$

  where $\mathrm{peel}(\sigma) = \tau'{::}\sigma'$. Thus letting $E = [\cdot]$ suffices.

Definition 8 (Stuck)

An expression e is stuck if e is not a value and there is no e' such that $e \to e'$.

Theorem 9 (Type Safety)

If $\cdot;\cdot \vdash_\tau e : \tau$ and $e \to^* e'$ (where $\to^*$ is the reflexive, transitive closure of $\to$, then e' is not stuck).

Proof: By induction on the number of steps to reach e', using induction and the Preservation Lemma to conclude $\cdot;\cdot \vdash_\tau e' : \tau$ and the Progress Lemma to conclude e' is not stuck.

3 Type Erasure

We start by listing the erasure rules. The untyped language is the standard, left-to-right, call-by-value λ-calculus with products and sums. We assume the reader is already familiar with its semantics, so we do not bother formalizing them here.

$\mathtt{erase}(()) = ()$
$\mathtt{erase}(x) = x$
$\mathtt{erase}((e_1, e_2)) = (\mathtt{erase}(e_1), \mathtt{erase}(e_2))$
$\mathtt{erase}(\pi_i\ e) = \pi_i\ \mathtt{erase}(e)$
$\mathtt{erase}(\mathtt{in}_i\ e) = \mathtt{in}_i\ \mathtt{erase}(e)$
$\mathtt{erase}(\mathtt{case}\ e\ \mathtt{of}\ x.e_1\ x.e_2) = \mathtt{case}\ \mathtt{erase}(e)\ \mathtt{of}\ x.\mathtt{erase}(e_1)\ x.\mathtt{erase}(e_2)$
$\mathtt{erase}(\mathtt{pack}\ \tau_1, e\ \mathtt{as}\ \tau_2) = \mathtt{erase}(e)$
$\mathtt{erase}(\mathtt{unpack}\ e_1\ \mathtt{as}\ \alpha, x\ \mathtt{in}\ e_2) = (\lambda x.\mathtt{erase}(e_2))\ \mathtt{erase}(e_1)$
$\mathtt{erase}(\mathtt{roll}\ e\ \mathtt{as}\ \tau) = \mathtt{erase}(e)$
$\mathtt{erase}(\mathtt{unroll}\ e) = \mathtt{erase}(e)$
$\mathtt{erase}(\mathtt{peel}\ e_1\ \mathtt{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathtt{in}\ e_2) = (\lambda x.\mathtt{erase}(e_2))\ \mathtt{erase}(e_1)$
$\mathtt{erase}(\Lambda\alpha : \kappa.e) = \lambda\_.\mathtt{erase}(e)$, where $\_$ not in e.
$\mathtt{erase}(e[\tau]) = \mathtt{erase}(e)\ ()$
$\mathtt{erase}(\lambda x : \tau.e) = \lambda x.\mathtt{erase}(e)$
$\mathtt{erase}(e_1\ e_2) = \mathtt{erase}(e_1)\ \mathtt{erase}(e_2)$
$\mathtt{erase}(\mathtt{fix}\ e) = \mathtt{fix}\ \mathtt{erase}(e)$

We now prove the erasure theorem. We begin with a useful lemma that shows substitutions for type variables disappear under our erasure rules. Intuitively, this holds because all types are erased by the erasure rules, and because substitution never changes the shape of the terms.

Lemma 10 (Type Substitution Invariance) *If e is an expression in the typed language, τ is a type and α is a type variable, then*

$$\mathtt{erase}(e[\tau/\alpha]) = \mathtt{erase}(e)\,. \tag{1}$$

Proof: All cases follow directly from structural induction on the expression e and from the definition of type substitution.

We next prove another lemma that shows that the erasure function distributes over term-variable substitution (i.e., the distributive property holds for this pair of operations).

Lemma 11 (Distributive Rule) *If e and e' are expressions in the typed language, and x is a variable, then*

$$\mathtt{erase}(e[e'/x]) = \mathtt{erase}(e)[\mathtt{erase}(e')/x]\,. \tag{2}$$

Proof: First, consider the case where $e = x$. Then we have

$$\mathtt{erase}(e[e'/x]) = \mathtt{erase}(x[e'/x]) = \mathtt{erase}(e')\,.$$

Also,

$$\mathtt{erase}(e)[\mathtt{erase}(e')/x] = \mathtt{erase}(x)[\mathtt{erase}(e')/x] = x[\mathtt{erase}(e')/x] = \mathtt{erase}(e')\,.$$

Thus both sides of equation 2 are equivalent, and so the lemma holds. All other cases follow directly by structural induction and by the definition of term-substitution.

We now prove a lemma that states that the erasure of a value is a value.

Lemma 12 (Value Erasure) *Let e be an expression in our typed language.*

1. *If e is a value, then $\mathtt{erase}(e)$ is a value in the untyped language.*
2. *If $\mathtt{erase}(e)$ is a value and $\cdot;\cdot \vdash e : \tau$ for some type τ, then there exists a value v such that $e \to^\star v$ and $\mathtt{erase}(v) = \mathtt{erase}(e)$.*

Proof: We have the following value forms in our typed language:

$$v ::= () \mid (v, v) \mid \mathtt{in}_i\ v \mid \lambda x : \tau.e \mid \Lambda\alpha : \kappa.e \mid \mathtt{pack}\ \tau, v\ \mathtt{as}\ \tau \mid \mathtt{roll}\ v\ \mathtt{as}\ \tau\,.$$

By structural induction and the given erasure rules, these reduce, respectively, to the following forms:

$$v ::= () \mid (v, v) \mid \mathtt{in}_i\ v \mid \lambda x.e' \mid \lambda\_.e' \mid v \mid v\,,$$

all of which are values in the untyped language. Thus part 1 holds.

We prove the second part by induction on the structure of e, proceeding by cases:

- **Case $e = x$:** Holds vacuously, because x does not typecheck under the context $\cdot;\cdot$.
- **Case $e = () \mid \lambda x : \tau.e_1 \mid \Lambda\alpha : \kappa.e_1$:** Holds trivially, because e is a value. Thus, we can let $v = e$ and so $e \to^\star v$ in zero steps and $\mathtt{erase}(v) = \mathtt{erase}(e)$.
- **Case $e = \pi_i e_1 \mid \mathtt{case}\ e_1\ x.e_2\ x.e_3 \mid e_1\ e_2 \mid \mathtt{fix}\ e_1 \mid e_1[t] \mid \mathtt{unpack}\ e_1\ \mathtt{as}\ \alpha, x\ \mathtt{in}\ e_2 \mid \mathtt{peel}\ e_1\ \mathtt{as}\ \alpha_h, \alpha_t, x\ \mathtt{in}\ e_2$:** Holds vacuously, because $\mathtt{erase}(e)$ is not a value.

- **Case** $e = (e_1, e_2) \mid \texttt{in}_i\ e_1 \mid \texttt{pack}\ \tau_1, e_1\ \texttt{as}\ \tau_2 \mid \texttt{roll}\ e_1\ \texttt{as}\ \tau$**:** By inspection of the erasure rules and the value forms, if $\texttt{erase}(e)$ is a value then $\texttt{erase}(e_i)$ must be a value for all i ($i = 1, 2$ for the pair case, and $i = 1$ for the rest). By induction and inspection of the typing rules (which show that if e typechecks under $\cdot;\cdot$, then so does e_i), $e_i \rightarrow^\star v_i$ and $\texttt{erase}(e_i) = \texttt{erase}(v_i)$. Thus, by the typed operational semantics, $e \rightarrow^\star v$ (where $v = (v_1, v_2) \mid \texttt{in}_i\ v_1 \mid \texttt{pack}\ \tau_1, v_1\ \texttt{as}\ \tau_2 \mid \texttt{roll}\ v_1\ \texttt{as}\ \tau$), and by the erasure rules, $\texttt{erase}(e) = \texttt{erase}(v)$.

- **Case** $e = \texttt{unroll}\ e_1$**:** By inspection of the erasure rules, if $\texttt{erase}(\texttt{unroll}\ e_1)$ is a value, then $\texttt{erase}(e_1)$ is a value. By induction and inspection of the typing rules (which show that if e typechecks under $\cdot;\cdot$, then so does e_1), $e_1 \rightarrow^\star v_1$ and $\texttt{erase}(v_1) = \texttt{erase}(e_1)$. Thus, by the typed operational semantics, $e \rightarrow^\star \texttt{unroll}\ v_1$. Since $\cdot;\cdot \vdash e : \tau$, by preservation and canonical forms v_1 must have the form $\texttt{roll}\ v_2\ \texttt{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau$. Thus, by the typed operational semantics, $\texttt{unroll}\ v_1 \rightarrow v_2$. So, we have $e \rightarrow^\star v_2$. Also, by the erasure rules $\texttt{erase}(e) = \texttt{erase}(e_1)$ and $\texttt{erase}(v_1) = \texttt{erase}(v_2)$. We already know that $\texttt{erase}(e_1) = \texttt{erase}(v_1)$, thus $\texttt{erase}(e) = \texttt{erase}(v_2)$ and so the lemma holds.

We next prove the key lemma, which states that erasure and evaluation commute in our enriched language. We will make use of the above lemmas, the Canonical Forms Lemma, and the Preservation Lemma. The Canonical Forms and Preservation Lemmas were proved in Section 2 as part of the type safety proof.

Lemma 13 (Erasure And Evaluation Commute) *Let e and e' be expressions in the typed language, and let $\cdot;\cdot \vdash e : \tau$ for some τ.*

1. *If $e \rightarrow e'$, then $\texttt{erase}(e) \rightarrow \texttt{erase}(e')$ or $\texttt{erase}(e) = \texttt{erase}(e')$.*
2. *If $\texttt{erase}(e) = t$ and $t \rightarrow u$, then there exists an e' such that $e \rightarrow^\star e'$ and $\texttt{erase}(e') = u$.*

Proof: We prove this lemma by structural induction on the expression e.

- **Case** $e = x$**:** If $e = x$, then $\texttt{erase}(e) = x$. The lemma thus holds vacuously, because neither e nor $\texttt{erase}(e)$ take a step. This case also holds vacuously because $e = x$ is not well typed under the empty context.

- **Case** $e = () \mid \lambda x : \tau.e_1 \mid \Lambda\alpha : \kappa.e_1$**:** In these cases, e is a value. By Lemma 12 (value erasure), $\texttt{erase}(e)$ is also a value. The lemma thus holds vacuously, because neither e nor $\texttt{erase}(e)$ take a step.

- **Case** $e = (e_1, e_2)$**:** By the given erasure rules, $\texttt{erase}(e) = (\texttt{erase}(e_1), \texttt{erase}(e_2))$.

  If e_1 is not a value, then the typed operational semantics ensures $e' = (e_1', e_2)$ for some e_1'. By induction, $\texttt{erase}(e_1) \rightarrow \texttt{erase}(e_1')$ or $\texttt{erase}(e_1) = \texttt{erase}(e_1')$. So by the untyped operational semantics, $(\texttt{erase}(e_1), \texttt{erase}(e_2)) \rightarrow (\texttt{erase}(e_1'), \texttt{erase}(e_2))$ or $(\texttt{erase}(e_1), \texttt{erase}(e_2)) = (\texttt{erase}(e_1'), \texttt{erase}(e_2))$. Thus, since $\texttt{erase}(e') = (\texttt{erase}(e_1'), \texttt{erase}(e_2))$, part 1 of the lemma holds.

  On the other hand, if e_1 is a value and e_2 is not a value then the typed operational semantics ensures $e' = (e_1, e_2')$ for some e_2'. By induction, $\texttt{erase}(e_2) \rightarrow \texttt{erase}(e_2')$ or $\texttt{erase}(e_2) = \texttt{erase}(e_2')$. By Lemma 12 (value erasure), $\texttt{erase}(e_1)$ is a value. So by the untyped operational semantics, $(\texttt{erase}(e_1), \texttt{erase}(e_2)) \rightarrow (\texttt{erase}(e_1), \texttt{erase}(e_2'))$ or $(\texttt{erase}(e_1), \texttt{erase}(e_2)) = (\texttt{erase}(e_1), \texttt{erase}(e_2'))$. Thus, since $\texttt{erase}(e') = (\texttt{erase}(e_1), \texttt{erase}(e_2'))$, part 1 of the lemma holds.

  Otherwise, if e_1 and e_2 are both values, then e is a value. Thus part 1 holds vacuously, because e does not take a step.

  If $\texttt{erase}(e_1)$ is not a value, then by the untyped operational semantics

  $$\texttt{erase}(e) = (\texttt{erase}(e_1), \texttt{erase}(e_2)) \rightarrow (e_1'', \texttt{erase}(e_2))$$

  for some e_1''. By induction, $e_1 \rightarrow^\star e_1'$ and $\texttt{erase}(e_1') = e_1''$. Thus by the typed operational semantics, $e \rightarrow^\star e' = (e_1', e_2)$. Since $\texttt{erase}(e') = (e_1'', \texttt{erase}(e_2))$, part 2 of the lemma holds.

  If $\texttt{erase}(e_1)$ is a value and $\texttt{erase}(e_2)$ is not a value, then by the untyped operational semantics

  $$\texttt{erase}(e) = (\texttt{erase}(e_1), \texttt{erase}(e_2)) \rightarrow (\texttt{erase}(e_1), e_2'')$$

for some e_2''. By induction, $e_2 \rightarrow^\star e_2'$ and $\texttt{erase}(e_2') = e_2''$. By Lemma 12, $e_1 \rightarrow^\star e_1'$, where e_1' is a value and $\texttt{erase}(e_1') = \texttt{erase}(e_1)$. Thus by the typed operational semantics, $e \rightarrow^\star e' = (e_1', e_2')$. Since $\texttt{erase}(e') = (\texttt{erase}(e_1), e_2'')$, part 2 of the lemma holds.

If $\texttt{erase}(e_1)$ and $\texttt{erase}(e_2)$ are both values, part 2 of the lemma holds vacuously because $\texttt{erase}(e)$ does not take a step.

- **Case** $e = \pi_i\ e_1$**:** By the given erasure rules, $\texttt{erase}(e) = \pi_i\ \texttt{erase}(e_1)$.

  If e_1 is not a value, then the typed operational semantics ensures that $e' = \pi_i\ e_1'$ for some e_1'. By induction, $\texttt{erase}(e_1) \rightarrow \texttt{erase}(e_1')$ or $\texttt{erase}(e_1) = \texttt{erase}(e_1')$. Thus by the untyped operational semantics, $\pi_i\ \texttt{erase}(e_1) \rightarrow \pi_i\ \texttt{erase}(e_1')$ or $\pi_i\ \texttt{erase}(e_1) = \pi_i\ \texttt{erase}(e_1')$. By the erasure rules, $\texttt{erase}(e') = \pi_i\ \texttt{erase}(e_1')$. Thus part 1 of the lemma holds.

  If e_1 is a value, then because e is well typed and because of the Canonical Forms Lemma, we know that e_1 has the from (v_1, v_2), where v_1 and v_2 are values. Thus by the typed operational semantics, $e' = v_i$. By the erasure rules, $\texttt{erase}(e_1) = (\texttt{erase}(v_1), \texttt{erase}(v_2))$. By Lemma 12 (value erasure), $(\texttt{erase}(v_1), \texttt{erase}(v_2))$ is a pair of values. By the untyped operational semantics, $\texttt{erase}(e) = \pi_i\ \texttt{erase}(e_1) \rightarrow \texttt{erase}(v_i) = \texttt{erase}(e')$. Thus part 1 of the lemma holds.

  If $\texttt{erase}(e_1)$ is not a value, then by the untyped operational semantics $\texttt{erase}(e) \rightarrow \pi_i\ e_1''$ for some e_1''. By induction, $e_1 \rightarrow^\star e_1'$ and $\texttt{erase}(e_1') = e_1''$. Thus by the typed operational semantics, $e \rightarrow^\star e' = \pi_i\ e_1'$. Since $\texttt{erase}(e') = \pi_i\ e_1''$, part 2 of the lemma holds.

  If $\texttt{erase}(e_1)$ is a value, then by Lemma 12 (value erasure) $e_1 \rightarrow^\star e_1' = v$ where v is a value. By the Canonical Forms Lemma, the Preservation Lemma, and the assumption that e is well typed, v must have the form (v_1, v_2) where v_1 and v_2 are values. Thus $e \rightarrow^\star v_i$. Since Lemma 12 also tells us that $\texttt{erase}(e_1) = \texttt{erase}(v)$, we know that $\texttt{erase}(e_1) = (\texttt{erase}(v_1), \texttt{erase}(v_2))$. This is a pair of values (Lemma 12), thus $\texttt{erase}(e) = \pi_i\ \texttt{erase}(e_1) \rightarrow \texttt{erase}(v_i)$. Thus part 2 of the lemma holds.

- **Case** $e = \texttt{in}_i\ e_1 \mid \texttt{case}\ e_1\ \texttt{of}\ x.e_2\ x.e_3 \mid \texttt{roll}\ e_1\ \texttt{as}\ \tau \mid \texttt{pack}\ \tau_1, e_1\ \texttt{as}\ \tau_2 \mid \texttt{fix}\ e_1 \mid e_1\ e_2$**:** These cases all follow directly by the same logic as the pair and projection cases. The fix and function application cases also require Lemma 11.

- **Case** $e = \texttt{unpack}\ e_1\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2$**:** We have $\texttt{erase}(e) = (\lambda x.\texttt{erase}(e_2))\ \texttt{erase}(e_1)$.

  If e_1 is not a value, then part 1 of the lemma follows directly by the same logic as the pair and projection cases.

  If e_1 is a value, then because e is well typed and because of the Canonical Forms Lemma, e_1 must have the form $\texttt{pack}\ \tau_1, v\ \texttt{as}\ \exists\alpha : \kappa.\tau_2$, where v is a value. Thus by the typed operational semantics, $e' = e_2[\tau_1/\alpha][v/x]$. By Lemmas 10 and 11, $\texttt{erase}(e') = \texttt{erase}(e_2)[\texttt{erase}(v)/x]$. By the erasure rules, $\texttt{erase}(e) = (\lambda x.\texttt{erase}(e_2))\ \texttt{erase}(v)$. The untyped operational semantics ensures that $\texttt{erase}(e) \rightarrow \texttt{erase}(e_2)[\texttt{erase}(v)/x]$ because the erasure of a value is a value (Lemma 12). Thus, part 1 of the lemma holds.

  If $\texttt{erase}(e_1)$ is not a value, then part 2 of the lemma follows directly by the same logic as the pair and projection cases.

  If $\texttt{erase}(e_1)$ is a value, then by the untyped operational semantics,

  $$\texttt{erase}(e) = (\lambda x.\texttt{erase}(e_2))\ \texttt{erase}(e_1) \rightarrow \texttt{erase}(e_2)[\texttt{erase}(e_1)/x]\,.$$

  By Lemma 12, the typed operational semantics, the Canonical Forms Lemma, the Preservation Lemma, and the assumption that e is well-typed, $e \rightarrow^\star e'' = \texttt{unpack}\ (\texttt{pack}\ \tau_1, v\ \texttt{as}\ \exists\alpha : \kappa.\tau_2)\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2$, where v is a value and $\texttt{erase}(v) = \texttt{erase}(e_1)$. Thus by the typed operational semantics, $e'' \rightarrow e' = e_2[\tau_1/\alpha][v/x]$. By Lemmas 10 and 11, $\texttt{erase}(e') = \texttt{erase}(e_2)[\texttt{erase}(v)/x] = \texttt{erase}(e_2)[\texttt{erase}(e_1)/x]$. Thus, since $e \rightarrow^\star e'$, part 2 of the lemma holds.

- **Case** $e = \texttt{peel}\ e_1\ \texttt{as}\ \alpha_1, \alpha_2, x\ \texttt{in}\ e_2$**:** This case follows directly by the same logic as the `unpack` case.

- **Case** $e = e_1[\tau]$**:** By the erasure rules, $\texttt{erase}(e) = \texttt{erase}(e_1)\ ()$.

  If e_1 is not a value, then part 1 of the lemma follows directly by the same logic as the pair and projection cases.

  If e_1 is a value, then because e is well typed and because of the Canonical Forms Lemma, e_1 must have the form $\Lambda\alpha : \kappa.e_2$. The typed operational semantics thus ensures that $e' = e_2[\tau/\alpha]$. By Lemma 10, $\texttt{erase}(e') = \texttt{erase}(e_2)$. Also, because we know the form of e_1, the erasure rules tell us that $\texttt{erase}(e) = \lambda\_.\texttt{erase}(e_2)\ ()$, where $\_$ is not in e_2. By the untyped operational semantics, $\texttt{erase}(e) \rightarrow \texttt{erase}(e_2) = \texttt{erase}(e')$. Thus, part 1 of the lemma holds.

  If $\texttt{erase}(e_1)$ is not a value, then part 2 of the lemma follows directly by the same logic as the pair and projection cases.

  If $\texttt{erase}(e_1)$ is a value, then by Lemma 12, the typed operational semantics, the Canonical Forms Lemma, the Preservation Lemma, and the assumption that e is well-typed, we know that $e \rightarrow^\star e'' = (\Lambda\alpha : \kappa.e_2)[\tau]$ and $\texttt{erase}(\Lambda\alpha : \kappa.e_2) = \texttt{erase}(e_1)$. By the typed operational semantics, $e'' \rightarrow e' = e_2[\tau/\alpha]$. By Lemma 10, $\texttt{erase}(e') = \texttt{erase}(e_2)$. We also have

  $$\texttt{erase}(e) = \texttt{erase}(e_1)\ () = \texttt{erase}(\Lambda\alpha : \kappa.e_2)\ () = (\lambda\_.\texttt{erase}(e_2))\ () \rightarrow \texttt{erase}(e_2)\ ,$$

  by the erasure rules and the untyped operational semantics. Thus part 2 of the lemma holds.

- **Case** $e = \texttt{unroll}\ e_1$**:** By the erasure rules, $\texttt{erase}(e) = \texttt{erase}(e_1)$.

  If e_1 is not a value, then part 1 of the lemma follows directly by the same logic as the pair and projection cases.

  If e_1 is a value, then because e is well typed and because of the Canonical Forms Lemma, e_1 must have the form $\texttt{roll}\ v\ \texttt{as}\ \tau$ where v is a value. Thus, by the typed operational semantics, $e \rightarrow v$. By the erasure rules, $\texttt{erase}(e) = \texttt{erase}(v)$. Thus, part 1 of the lemma holds.

  If $\texttt{erase}(e_1)$ is not a value, then part 2 of the lemma follows directly by the same logic as the pair and projection cases.

  If $\texttt{erase}(e_1)$ is a value, then $\texttt{erase}(e) = \texttt{erase}(e_1)$ is a value. Thus $\texttt{erase}(e)$ takes no steps, and so part 2 of the lemma holds vacuously.

Finally, we prove the erasure theorem.

Theorem 14 (Erasure Theorem) *If e is an expression in the typed language, v is a value in the typed language, and $e \rightarrow^\star v$, then $\texttt{erase}(e) \rightarrow^\star \texttt{erase}(v)$ in the untyped language. (Also, e and $\texttt{erase}(e)$ have the same termination behavior.)*

Proof: The first part follows from Lemma 13 and induction on the derivation $e \rightarrow^\star v$. To prove that the termination behaviors are identical, we need to show that e terminates if and only if $\texttt{erase}(e)$ terminates. The forward direction follows directly from the first part of this Lemma and part 1 of Lemma 12. The reverse direction follows from part 2 of Lemma 12, Lemma 13, and induction on the derivation $\texttt{erase}(e) \rightarrow^\star v$, where v is a value in the untyped language.

References

[1] Michael F. Ringenburg and Dan Grossman. A type system for coordinated data structures, July 2004. Submitted for publication.