A Calculus for Dynamic Loading

Michael Hicks
University of Pennsylvania

Stephanie Weirich
Cornell University

February 6, 2001

Abstract

We present the load-calculus, used to model dynamic loading, and prove it sound. The calculus extends the polymorphic λ-calculus with a load primitive that dynamically loads terms that are closed, with respect to values. The calculus is meant to approximate the process of dynamic loading in TAL/Load [4], an version of Typed Assembly Language [7] extending with dynamic linking. To model the key aspects of TAL, the calculus contains references and facilities for named types. Loadable programs may refer to named types defined by the running program, and may export new types to code loaded later. Our approach follows the framework initially outlined by Glew *et. al* [3]. This calculus has been implemented in the TALx86 [6] version of Typed Assembly Language, and is used to implement a full-featured dynamic linking library, DLpop [4].

1 Introduction

The purpose of this report is prove the soundness of a simple calculus for dynamic loading, designed as a theoretical basis for dynamic linking in Typed Assembly Language [7, 6, 3]. Dynamic loading is provided by a simple primitive load, which converts the binary representation of some closed[1] term into the term itself. This calculus is meant to approximate an implementation of dynamic loading in Typed Assembly Language, and so the language provides references and named types, in addition to load and lambda-terms.

Here is a small example of the use of load. Consider the following program (using OCaml-style [5] syntax):

```
let f f2 =
  let x = 1 in
  (3, f2 x)
```

[1]By closed, we mean that there are no free *value* variables; there may be free *type labels*, as we describe shortly.

Assuming `f2` has type `int` $\rightarrow$ `int`, then f has type (`int` $\rightarrow$ `int`) $\rightarrow$ `int` $\times$ `int`. We could load this (closed) program in another program. Assuming that the binary representation of this program is stored in the file "f":

```
let g () =
  let load_succ f = f (function x -> x + 1) in
  let load_fail = (3,4) in
  load [(int -> int]) -> int * int] ("f",load_succ,load_fail)
```

When `g` is executed, it will call load. If load succeeds, it will call `load_succ` with the loaded function of type `int` $\rightarrow$ `int`, in this case the function `f`. If load fails (because of either a type or format error), then `load_fail` is executed instead. The result of evaluating this program, assuming "f" is well-formed, is `(3,2)`.

In order to properly accommodate named types, we define the notion of a *type heap*, which maps *type labels* to types, where the type label corresponds to the named type and the type it maps to is the implementation of that type. A program has two type heaps, the *current* type heap, and the *imported* type heap. Labels within the imported type heap correspond to types not defined by the current program. Additionally, a type label in the imported type heap could be undefined, corresponding to a type whose representation is defined externally. In the ML terminology, an imported type whose representation is known essentially corresponds to *manifest type*, while one whose representation isn't known corresponds to an opaque type, the consequence of which is that values of that type may only be used abstractly. For example, we can (roughly) modify the two programs above as follows:

```
extern namedtype t = int * int
let f f2 =
  let x = 1 in
  roll [t] (3, f2 x)
```

Here `extern` is used to approximate a manifest type defined externally, and `namedtype` indicates that `t` is a unique name for the type.

```
namedtype t = int * int
let g () =
  let load_succ f = f (function x -> x + 1) in
  let load_fail = roll [t] (3,4) in
  let x = load [(int -> int) -> int -> t]
    ("f",load_succ,load_fail) in
  unroll x
```

Here, the running program defines the named type `t`, which is matched up at load time with the imported type `t` defined in the other program. We use `roll` and `unroll` to convert to and from named types.

We could also define opaque named types in both the running program and the loaded code, and this is well-typed as long as the values of those types are used abstractly (that is, they are never part of a `roll` or `unroll` expression, which would reveal their representation). For instance, we could modify the loaded program as:

```
extern namedtype t
let f (f2:t->t) (x:t) =
  (f2 x, f2 x)
```

Now the function `f` has type $(t \to t) \to t \to t \times t$. The running program could become:

```
namedtype t = int * int
let g () =
  let load_succ f = f
    (function x -> let (y,z) = unroll x in roll [t] (y+1,z+1)) in
  let load_fail = (roll [t] (3,4),roll [t] (4,5)) in
  let x = load [(t -> t) -> t -> t]
              ("f",load_succ,load_fail) in
  unroll [t] x
```

The converse is also possible. For instance,

```
namedtype t = int * int
let f x y =
  roll [t] (x,y)
```

Now the function `f` has type `int` $\to$ `int` $\to t$. The running program could become:

```
extern namedtype t
let g () =
  let load_succ f = (f 1 2; ()) in
  let load_fail = () in
  load [int -> int -> t] ("f",load_succ,load_fail)
```

Note that after the value of type t is created, it is discarded. We could easily have defined other functions in the loaded code to manipulate values of type t. We could also combine these two programs to allow recursively defined named types across modules.

The remainder of this report develops the calculus which allows such programs to be written in a well-typed manner. We define the syntax, a static and operational semantics, and finally a type-soundness theorem.

$$\begin{array}{llcl}
\textit{types} & \tau & ::= & \texttt{int} \mid l \mid \tau \to \tau \mid \tau\ \texttt{ref} \\
 & & \mid & \alpha \mid \forall \alpha.\tau \\
\textit{type heaps} & X & ::= & \{l_1 = \chi_1, \ldots, l_n = \chi_n\} \\
\textit{type heap values} & \chi & ::= & \bot \mid \tau \\
\textit{type interfaces} & \Theta & ::= & (X_I, X_H) \\
 & & & \\
\textit{expressions} & e & ::= & i \mid L \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \\
 & & \mid & \Lambda\alpha.e \mid e[\tau] \mid \texttt{unroll}\ e \\
 & & \mid & \texttt{roll}_l\ e \mid \texttt{ref}\ e \\
 & & \mid & \texttt{assign}\ e_1 e_2 \mid !e_1 \\
 & & \mid & \mathsf{load}[\tau]\ e_0\ e_1\ e_2\ e_3 \\
\textit{values} & v & ::= & i \mid L \mid x \mid \lambda x{:}\tau.e \mid \texttt{roll}_l\ v \\
\textit{value heaps} & H & ::= & \{L_1 = v_1, \ldots, L_n = v_n\} \\
 & & & \\
\textit{programs} & P & ::= & (\Theta, H, e) \\
 & & & \\
\textit{value heap types} & \Phi & ::= & \{L_1 : \tau_1, \ldots, L_n : \tau_n\} \\
\textit{type contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha \\
\textit{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}$$

$$\boxed{\begin{array}{l}
i \in \mathcal{Z} \\
l \in \mathsf{TypeLabs} \\
L \in \mathsf{ValueLabs} \\
x \in \mathsf{Vars} \\
\alpha \in \mathsf{TypeVars}
\end{array}}$$

Figure 1: load-calculus Syntax

2 Syntax

The syntax of the load-calculus is shown in Figure 1. A *program* P consists of a *type interface* Θ, a heap H, and an expression to evaluate e. The heap H stores reference cell values, and the expression e represents the program's computation. The interesting part is the type interface, which consists of two *type heaps* X_I and X_H that defined the *named types* of the program. X_I contains named types *imported* by the program (to be resolved later during a linking phase), while X_H contains named types defined by the program, respectively.

A type heap is a finite map; it maps each *type label* l in its domain to a *type heap value* χ, which is either a type τ or $\bot$; the latter indicates an undefined (unresolved) type label. We write $X(l)$ to denote χ in the heap $X = \{\ldots, l = \chi, \ldots\}$. For the heap $X = \{l_1 = \chi_1, \ldots, l_n = \chi_n\}$, $\texttt{dom}(X)$ refers to the set $\{l_1, \ldots, l_n\}$ and $\texttt{rng}(X) = \{\chi_1, \ldots, \chi_n\}$.

Most types τ are standard, particularly base type `int`, function types $\tau \to \tau$, reference types τ `ref` , variable types α and universally-quantified types $\forall\alpha.\tau$. We use the type label l itself to represent a named type defined in the type heap.

Most expressions are standard, particularly integers i, variables x, value abstractions $\lambda x{:}\tau.e$ and applications $e_1 e_2$, type abstractions $\Lambda\alpha.e$ and applications $e[\tau]$, reference construction `ref` e, reference assignment `assign` $e_1 e_2$, and reference deconstruction (dereference) $!e_1$. We assume all λ-bound variables are unique. To coerce an expression e to named type l, we provide a coercion $\texttt{roll}_l\ e$; `unroll` performs the reverse operation. For example, say the type heap X_H has the form $\{\mathit{filehandle} = \texttt{int}\}$, indicating that the named type *filehandle* is defined to have type `int`. To coerce the integer 1 to have type *filehandle*, we would do $\texttt{roll}_{\mathit{filehandle}}\ 1$. Converting it back to an integer would simply require an `unroll`; *i.e.* $\texttt{unroll}\ (\texttt{roll}_{\mathit{filehandle}}\ 1)$. Note that named types are also permitted to be abstract, such that this `unroll` operation is not permitted; this will be clear in the presentation of the static semantics.

Reference values are stored in the value heap H. Value heaps are finite maps mapping value labels L to values. We write $H(L)$ to denote v in the heap $H = \{\ldots, L = v, \ldots\}$. For the heap $H = \{L_1 = v_1, \ldots, L_n = v_n\}$, $\texttt{dom}(H)$ refers to the set $\{L_1, \ldots, L_n\}$ and $\texttt{rng}(H) = \{v_1, \ldots, v_n\}$. If $H = \{\ldots, L = v, \ldots\}$, then let $H[L = v']$ be the heap $\{\ldots, L = v', \ldots\}$; this operation is undefined if $L \notin H$.

3 Dynamic Semantics

In this section we present the rules for the model of computation in the load-calculus.

3.1 Linking

A running program may dynamically link in other programs as it runs using load. The load expression models the loading of object files in TAL/Load, but varies slightly from its presentation in [4]. Here, load takes 'integers' as its first two arguments (representing the bytes of a program and a *type heap mask*, explained more below) and alternative branches for a successful load and failure.

When a running program dynamically loads another program into it, both the value and type heaps of the two programs must be merged. This process is called *linking*. We first define linking for type heaps, and then for value heaps.

Stated informally, linking two type heaps together yields a resulting heap (1) whose exports are the disjoint union of the source exports, and (2) whose imports are the merge of the imports minus the exports. Type heap linking is well-formed if the imports agree with each other (that is, they don't define any named types whose definitions conflict), and the exports are disjoint. Using the operations and predicates on type heaps and type heap values shown in Figure 2, type heap linking is defined formally as:

Type Heap Values

Operators		
join	$\chi_1 \sqcup \chi_2$	$\bot \sqcup \chi = \chi \qquad \chi \sqcup \bot = \chi \qquad \chi \sqcup \chi = \chi$
approximates	$\chi_1 \leq \chi_2$	$\chi \leq \bot \qquad \chi \leq \chi$
Predicates		
similar	$\chi_1 \sim \chi_2$	$\chi_1 \leq \chi_2$ or $\chi_2 \leq \chi_1$

Type Heaps

Operators		
restriction	$X_1 - X_2$	X_1 restricted to labels not in $\texttt{dom}(X_2)$
disjoint union	$X_1 \uplus X_2$	Union of disjoint maps, defined if $X_1 \mid X_2$
merge	$X_1 \oplus X_2$	Union of similar maps (defined if $X_1 \sim X_2$), maps $l \in \texttt{dom}(X_1) \cap \texttt{dom}(X_2)$ to $X_1(l) \sqcup X_2(l)$
Predicates		
disjoint	$X_1 \mid X_2$	$\texttt{dom}(X_1)$ and $\texttt{dom}(X_2)$ are disjoint
link compatible	$X_1 \precsim X_2$	For l in $\texttt{dom}(X_1) \cap \texttt{dom}(X_2), X_1(l) \leq X_2(l)$
similar	$X_1 \sim X_2$	For l in $\texttt{dom}(X_1) \cap \texttt{dom}(X_2), X_1(l) \sim X_2(l)$
subtype	$X_1 \leq X_2$	$X_1 \precsim X_2$ and $\texttt{dom}(X_2) \subseteq \texttt{dom}(X_1)$

Figure 2: Type heaps and type heap values: operators and predicates

Definition 3.1 (Type Heap Linking)

$$\frac{X_I^1 \sim X_I^2 \qquad X_H^1 \precsim X_I^2 \qquad X_H^2 \precsim X_I^1 \qquad X_H^1 \mid X_H^2}{(X_I^1, X_H^1)\ \texttt{link}\ (X_I^2, X_H^2) \Rightarrow (X_I^3, X_H^3)} \left(\begin{array}{c} X_H^3 = X_H^1 \uplus X_H^2 \\ X_I^3 = ((X_I^1 \oplus X_I^2) - X_H^3) \end{array} \right)$$

Value heap linking is essentially the same as type heap linking, minus the requirements and operations concerning imported values. This is because value heaps are required to be *closed* (*i.e.* self-contained), so there is no import heap and linking them together just becomes disjoint union. This requirement is in contrast to typical definitions of value heap (or term-level) linking (*e.g.* [3, 1, 2]); we make it because we expect value linking to occur in the term language itself.

Definition 3.2 (Value Heap Linking)

$$\frac{H_1 \mid H_2}{H_1\ \texttt{link}\ H_2 \Rightarrow H_3} \ (H_3 = H_1 \uplus H_2)$$

where	$H_1 \mid H_2$	$\texttt{dom}(H_1)$ *and* $\texttt{dom}(H_2)$ *are disjoint*
	$H_1 \uplus H_2$	*Union of disjoint maps, defined if* $H_1 \mid H_2$

3.2 Operational Semantics

We define the operational semantics for the load-calculus using a one-step reduction operator $\mapsto$, following a call-by-value discipline. The most interesting construct is load; its operational rules appear in Figure 3. The first two (term) arguments to load are the integers h, specifying the type heap mask to use during linking, and i, specifying the program to load. We use $\hat{\ }$ as some function that maps integer arguments to programs or type heaps, as appropriate, modeling a filesystem. The argument $\hat{h}$ is a type heap that is more restrictive than the running program's type export heap. It is used by the caller to limit the definitions that may be seen by the loaded code, if desired; as such, we refer to it as a *type heap mask.* The second two arguments to load are the success and failure expressions. If $load - success$ is used, then the success expression is applied to the loaded program expression e; otherwise the expression e_3 is used (*i.e.*, when using $load - failure$). The type argument to τ to load indicates the expected type of the expression in the loaded program.

For load to succeed, three conditions must be met. First, value heap linking $H_1 \,\texttt{link}\, H_2 \Rightarrow H_3$ must succeed, combining the running program's value heap H_1 and the loaded program's value heap H_2 to produce heap H_3. Second, $X_H^1 \vdash \hat{\imath} : \tau$, indicating that the program to be loaded $\hat{\imath}$ is well-formed in the context of the running program's export heap, having type τ, matching the type argument passed to load. Program well-formedness is presented as part of the 'static' semantics in the next subsection. Finally, type heap linking $(X_I^1, X) \,\texttt{link}\, (X_I^2, X_H^2) \Rightarrow (X_I^3, X')$ must succeed. Rather than linking the running program's type interface (X_I^1, X_H^1) with the loaded program's interface, we link (X_I^1, X) instead, thereby replacing the export type heap X_H^1 with X, which is the type heap indicated by $\hat{h}$. This type heap must be the same as or more restrictive than the program's export type heap, as required by the conditions $X_H^1 \leq X$ and $X_I^2 \mid (X_H^1 - X)$. This linking operation produces export heap X', which is merged with the program export heap in the new program: $X_H^3 = X' \oplus X_H^1$.

The remaining operational rules for the calculus are shown in Figure 4. We define $e[e'/x]$ as the capture-avoiding substitution of the term e' for each occurrence of the variable x in the term e. These rules are basically standard. In particular, *beta* performs function application via substitution; *unroll* guarantees that a value that has been coerced to a named type cannot be examined until it has been unrolled; *ref* causes the 'allocation' of a unique value label L in the value heap and stores the value v there; *deref* extracts the value v mapped to by value label L in the value heap; *assign* overwrites the existing mapping for value label L in the value heap with one from L to v (recall that the operation $H[L = v]$ requires that L be defined in H); *tapp* performs type application via substitution. The remaining rules are *congruence* rules. One oddity is that the success and failure expressions in the

$$\boxed{(\Theta, H, \mathsf{load}\ i_1\ i_2\ e_1\ e_2) \mapsto (\Theta', H', e')}$$

$$\frac{\begin{array}{c}(X_I^1, X)\ \mathtt{link}\ (X_I^2, X_H^2) \Rightarrow (X_I^3, X') \\ X_H^1 \leq X \qquad X_I^2 \mid (X_H^1 - X) \\ X_H^1 \vdash \hat{i} : \tau \\ H_1\ \mathtt{link}\ H_2 \Rightarrow H_3\end{array}}{\begin{array}{l}((X_I^1, X_H^1), H_1, \mathsf{load}[\tau]\ h\ i\ e_2\ e_3) \mapsto \\ ((X_I^3, X_H^3), H_3, e_2\ e)\end{array}} \left(\begin{array}{c}\hat{h} = X \\ \hat{i} = ((X_I^2, X_H^2), H_2, e) \\ X_H^3 = X' \oplus X_H^1\end{array}\right) \quad \textit{(load-success)}$$

$$(\Theta, H, \mathsf{load}[\tau]\ h\ i\ e_2\ e_3) \mapsto (\Theta, H, e_3) \quad (\textit{load} - \textit{failure})$$
otherwise

Figure 3: Operational rules for load

congruence rules are *not* call-by-value; they are left unevaluated until the actual loading operation takes place. Then only one of them will evaluate, based on the result. Note that we require a type-passing semantics because the type argument passed to load is used at runtime; we use type-erasure semantics in TAL/Load by introducing λ_R-style term representations for types, as explained in the next section.

4 Static Semantics

As is standard, the static semantics is used to statically check that a program is well-formed. In addition, the operational rule for load requires that well-formedness be checked at *runtime*, before a program can be dynamically loaded into the running program. Informally, program well-formedness is defined as follows. The program type interface components X_I and X_H must be disjoint and well-formed; the value heap H must be well-typed in the context of the type interface; and the program expression e must be well-typed in the context of both the type and value heaps. In the case that the program is being loaded dynamically, its type export heap labels X_H must be disjoint from those of the running program X_P. This is stated formally below.

Definition 4.1 (Program well-formedness)

$$\frac{\begin{array}{cc}\vdash X_I \uplus X_H & X_I \uplus X_H \vdash \Phi \\ X_I \uplus X_H \vdash H : \Phi & X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e : \tau\end{array}}{X_P \vdash ((X_I, X_H), H, e) : \tau} \left(\ X_H \mid X_P\ \right)$$

$\boxed{(\Theta, H, e) \mapsto (\Theta', H', e')}$

$$
\begin{array}{lcll}
(\Theta, H, (\lambda x{:}\tau.e)\ v) & \mapsto & (\Theta, H, e[v/x]) & (\textit{beta}) \\[1em]
(\Theta, H, \texttt{unroll}(\texttt{roll}_l\ v)) & \mapsto & (\Theta, H, v) & (\textit{unroll}) \\[1em]
(\Theta, H, \texttt{ref}\ v) & \mapsto & (\Theta, H \uplus \{L = v\}, L) & (\textit{ref}) \\
\text{where } L \notin \texttt{dom}(H) & & & \\[1em]
(\Theta, H, !L) & \mapsto & (\Theta, H, v) & (\textit{deref}) \\
\text{where } H(L) = v & & & \\[1em]
(\Theta, H, \texttt{assign}\ L\ v) & \mapsto & (\Theta, H[L = v], v) & (\textit{assign}) \\[1em]
(\Theta, H, (\Lambda\alpha.e)[\tau]) & \mapsto & (\Theta, H, e[\tau/\alpha]) & (\textit{tapp})
\end{array}
$$

$$
\frac{(\Theta, H, e) \mapsto (\Theta', H', e')}{\left\{\begin{array}{c}
(\Theta, H, e\ e_2) \mapsto (\Theta', H', e'\ e_2) \\
(\Theta, H, v_1\ e) \mapsto (\Theta', H', v_1\ e') \\
(\Theta, H, \texttt{roll}_l\ e) \mapsto (\Theta', H', \texttt{roll}_l\ e') \\
(\Theta, H, \texttt{unroll}\ e) \mapsto (\Theta', H', \texttt{unroll}\ e') \\
(\Theta, H, \mathsf{load}[\tau]\ e\ e_1\ e_2\ e_3) \mapsto (\Theta', H', \mathsf{load}[\tau]\ e'\ e_1\ e_2\ e_3) \\
(\Theta, H, \mathsf{load}[\tau]\ v\ e\ e_2\ e_3) \mapsto (\Theta', H', \mathsf{load}[\tau]\ v\ e'\ e_2\ e_3) \\
(\Theta, H, \texttt{ref}\ e) \mapsto (\Theta', H', \texttt{ref}\ e') \\
(\Theta, H, !e) \mapsto (\Theta', H', !e') \\
(\Theta, H, \texttt{assign}\ e\ e_2) \mapsto (\Theta', H', \texttt{assign}\ e'\ e_2) \\
(\Theta, H, \texttt{assign}\ v\ e) \mapsto (\Theta', H', \texttt{assign}\ v\ e') \\
(\Theta, H, e[\tau]) \mapsto (\Theta', H', e'[\tau])
\end{array}\right\}} \quad (\textit{congruence})
$$

Figure 4: Operational rules, excluding load

$\boxed{X;\Delta \vdash \tau}$

$$X;\Delta \vdash \mathtt{int} \qquad \frac{\alpha \in \Delta}{X;\Delta \vdash \alpha} \qquad \frac{l \in \mathtt{dom}(X)}{X;\Delta \vdash l}$$

$$\frac{X;\Delta \vdash \tau' \qquad X;\Delta \vdash \tau}{X;\Delta \vdash \tau' \to \tau} \qquad \frac{X;\Delta \vdash \tau}{X;\Delta \vdash \mathtt{ref}\ \tau} \qquad \frac{X;\Delta,\alpha \vdash \tau}{X;\Delta \vdash \forall\alpha.\tau}\ (\alpha \notin \Delta)$$

$\boxed{\vdash X}$

$$\frac{X;\cdot \vdash \tau \quad (\text{for each } \tau \in \mathtt{rng}(X))}{\vdash X}$$

$\boxed{X;\Delta \vdash \Gamma}$

$$X;\Delta \vdash \cdot \qquad \frac{X;\Delta \vdash \Gamma \qquad X;\Delta \vdash \tau}{X;\Delta \vdash \Gamma, x{:}\tau}$$

$\boxed{X \vdash \Phi}$

$$\frac{X;\cdot \vdash \tau \quad (\text{for each } \tau \in \mathtt{rng}(\Phi))}{X \vdash \Phi}$$

$\boxed{X \vdash H : \Phi}$

$$\frac{X;\Phi;\cdot;\cdot \vdash H(L) : \Phi(L) \quad (\text{for each } L \in \mathtt{dom}(H))}{X \vdash H : \Phi}$$

Figure 5: Well-formedness for types, type heaps, contexts, value heap typings, and value heaps

$$\frac{\{\} \vdash (\Theta, H, e) : \tau}{\vdash (\Theta, H, e) : \tau}$$

Figure 5 presents well-formedness conditions for types, type heaps, contexts, value heap typings, and value heaps. Types are checked for well-formedness in relation to a type heap X and a type variable context Δ. The former is used to make sure that a named type l is present in the type heap, and the latter is used to make sure a type variable α is properly quantified. A type heap X is well-formed if all of the types mentioned in its range are well-formed. This rule in combination with the one for named types allows named types to be mutually recursive. Note that a well-formed type heap is *closed*; all of the labels appearing in its range are defined in the type heap itself.

Like type heaps, a value heap typing Φ is well-formed if all of the types mentioned in its range are well-formed. A value heap is well-formed if the values therein may be typed with a given value heap typing. Value typing is checked via the expression typing relation, shown in Figure 6.

Most of expression typing rules are standard. Noteworthy are the rules for load, and `roll` and `unroll`. As mentioned in the operational semantics, the first two term arguments, which are mapped at runtime to a type heap 'context' and a program, respectively, must have type `int`. The type argument τ' indicates the expected type of the loaded program's term component. The third term argument is the 'success-expression' which is applied to the loaded code, so it must take an argument of type τ', returning a result of type τ. The final term argument is the 'failure-expression' which is executed if loading fails; its type must match the return type τ of the success condition so that the overall type of the load expression will be τ.

We use `unroll` to coerce an expression e having some named type l. The result has type τ, where l maps to τ in the type heap X. We use $\texttt{roll}_l$ to coerce an expression e to named type l; if e has type τ then the type heap X must map l to τ. The semantics allows for named types to be opaque (abstract). In particular, the expression $\texttt{unroll} e : \tau$ is only well-typed if $X(l) = \tau$. To make l abstract, we set $X(l)$ to $\perp$, forbidding the coercion to the implementation type. In practice, label l is made abstract to loaded code by mapping it to $\perp$ in the type heap mask X during loading.

$\boxed{X;\Phi;\Delta;\Gamma \vdash e : \tau}$

$$\frac{\begin{array}{c}X;\Phi;\Delta;\Gamma \vdash e_0 : \texttt{int}\\ X;\Phi;\Delta;\Gamma \vdash e_1 : \texttt{int}\\ X;\Phi;\Delta;\Gamma \vdash e_2 : \tau' \to \tau\\ X;\Phi;\Delta;\Gamma \vdash e_3 : \tau\end{array}}{X;\Phi;\Delta;\Gamma \vdash \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3 : \tau}$$

$$X;\Phi;\Delta;\Gamma \vdash i : \texttt{int} \qquad X;\Phi;\Delta;\Gamma \vdash x : \Gamma(x) \qquad X;\Phi;\Delta;\Gamma \vdash L : \Phi(L)\ \texttt{ref}$$

$$\frac{X;\Phi;\Delta;\Gamma \vdash e : l}{X;\Phi;\Delta;\Gamma \vdash \texttt{unroll}\ e : \tau}\ (X(l) = \tau) \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e : \tau}{X;\Phi;\Delta;\Gamma \vdash \texttt{roll}_l\ e : l}\ (X(l) = \tau)$$

$$\frac{X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash e : \tau \qquad X;\Delta \vdash \tau'}{X;\Phi;\Delta;\Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau}$$

$$\frac{X;\Phi;\Delta;\Gamma \vdash e_1 : \tau' \to \tau \qquad X;\Phi;\Delta;\Gamma \vdash e_2 : \tau'}{X;\Phi;\Delta;\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{X;\Phi;\Delta,\alpha;\Gamma \vdash e : \tau}{X;\Phi;\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e : \forall\alpha.\tau \qquad X;\Delta \vdash \tau'}{X;\Phi;\Delta;\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}$$

$$\frac{X;\Phi;\Delta;\Gamma \vdash e : \tau}{X;\Phi;\Delta;\Gamma \vdash \texttt{ref}\ e : \tau\ \texttt{ref}} \qquad \frac{X;\Phi;\Delta;\Gamma \vdash e : \tau\ \texttt{ref}}{X;\Phi;\Delta;\Gamma \vdash\ !e : \tau}$$

$$\frac{\begin{array}{c}X;\Phi;\Delta;\Gamma \vdash e_1 : \tau\ \texttt{ref}\\ X;\Phi;\Delta;\Gamma \vdash e_2 : \tau\end{array}}{X;\Phi;\Delta;\Gamma \vdash \texttt{assign}\ e_1\ e_2 : \tau}$$

Figure 6: Expression typing

5 Properties of the formal system

The important formal property of this system is that it is *type-safe* (this property is also called *type-soundness*). In particular, if a program is well-typed, it will execute in a well-defined fashion indefinitely, or until it completes with a particular value. Formally stated:

Theorem 5.1 (Type Safety) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto^* (\Theta', H', e')$ *then* (Θ', H', e') *then either* e' *is a value or may be further reduced by some rule of the operational semantics.*

Note that $\mapsto^*$ is the multi-step reduction relation, indicating one or more applications of the single-step relation $\mapsto$. Type safety is proven using the standard technique of showing *subject reduction* and *progress*:

Lemma 5.2 (Subject Reduction) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto (\Theta', H', e')$ *then* $\vdash (\Theta', H', e') : \tau$

Lemma 5.3 (Progress) *If* $\vdash (\Theta, H, e) : \tau$ *and* e *is not a value, then there exists a* (Θ', H', e') *such that* $(\Theta, H, e) \mapsto (\Theta', H', e')$.

Stated informally, subject reduction indicates that if a given program has a type τ, and it may take (at least) one reduction step, then the resulting program, after applying the reduction rule, still has type τ. Progress indicates that if a well-typed program cannot take an evaluation step, then it must be a value having type τ. The following chapter presents proofs for these properties.

6 Proofs

Our presentation of the proofs of soundness is bottom-up, starting with properties of the system needed for the final proof. We start with properties of type heaps, then properties of value heaps, then properties of type derivations, and finally the proof of type-safety.

6.1 Properties of Type Heaps

All of the Lemmas (and their corollaries) developed in this subsection are for the purpose of proving the load case of the subject reduction, in Section 6.4.

Lemma 6.1 (Type Heap Equalities) *Suppose* A, B, C, D *are type heaps, then*

1. $(A \oplus B) \oplus C = A \oplus (B \oplus C)$

2. $(A \oplus B) = (B \oplus A)$

3. *if* $A \mid B$ *then* $A \oplus B = A \uplus B$

4. *if* $A \precsim C$, $B \precsim D$, $A \sim B$ *and* $C \sim D$, $A \mid D$, $B \mid C$, *then* $(A \oplus B) \precsim (C \oplus D)$

5. *if* $B \precsim A$ *then* $(A - B) \uplus B = A \oplus B$.

6. *if* $A \leq B$ *then* $A \precsim B$

7. *if* $A \leq B$ *then* $A \oplus B = A$

8. *if* $A \precsim B$ *then* $A \sim B$

9. *if* $B \leq A$ *and* $C \mid (B - A)$ *then* $C - B = C - A$

Proof of 4 This fails if for some l, $(A \oplus B)(l) = \bot$ and $(C \oplus D)(l) = \tau$. Assume $A(l) = \bot$. Then $C(l) = \bot$, if $l \in \texttt{dom}(C)$ as $A \precsim C$. Furthermore $l \notin \texttt{dom}(D)$ as $A \mid D$. So $(C \oplus D)(l) = \bot$ if anything. Analogous reasoning if $B(l) = \bot$.

Proof of 5 If $l \in A$ and not in B then trivially $(A \oplus B)(l) = ((A - B) \uplus B)(l)$, likewise if $l \in B$ and not A, and if $A(l) = B(l)$. Suppose $A(l) = \bot$ and $B(l) = \tau$, then $(A \oplus B)(l) = \tau$ and $(A - B)(l)$ is undefined so $(A - B) \uplus B(l) = \tau$. The reverse case, where $A(l) = \tau$ and $B(l) = \bot$ cannot happen by assumption.

Proof of 9 As C does not include labels in B that are not in A, then removing the labels from C that are in B is the same as removing only the ones from A.

Lemma 6.2 (Type Heap Merge) *If* $\vdash X_A$ *and* $\vdash X_B$ *then* $\vdash X_A \oplus X_B$.

Lemma 6.3 (Type Heap Weakening) *Suppose* $X \oplus X'$ *is well-defined.*

1. *If* $X; \Delta \vdash \tau$ *then* $(X \oplus X'); \Delta \vdash \tau$

2. *If* $X \vdash \Phi$ *then* $X \oplus X' \vdash \Phi$

3. *If* $X; \Phi; \Delta; \Gamma \vdash e : \tau$ *then* $X \oplus X'; \Phi; \Delta; \Gamma \vdash e : \tau$.

4. *If* $X \vdash H : \Phi$ *then* $X \oplus X' \vdash H : \Phi$

Proof

1. Proof is by induction on $X; \Delta \vdash \tau$. If $\tau = \alpha$ then $(X \oplus X'); \Delta \vdash \alpha$. If $\tau = \texttt{int}$ then $(X \oplus X'); \Delta \vdash \texttt{int}$. If $\tau = l$ then l is still in the domain of $X \oplus X'$ (though its range might change), so $(X \oplus X'); \Delta \vdash l$. The remaining cases follow by induction.

2. We are given that for each $\tau \in \texttt{rng}(\Phi)$, $X \vdash \tau$. It follows by 1 that $X \oplus X' \vdash \tau$.
3. Proof is by induction on $X; \Phi; \Delta; \Gamma \vdash e : \tau$. This follows trivially or by induction for every rule except: If e is an abstraction or a type application, then Part 1 is also needed to verify the type added to the context. If e is $\texttt{unroll}\ e'$ or $\texttt{roll}_l\ e'$ then we note that because $X(l) = \tau$, then by the definition of $X \oplus X'$, $(X \oplus X')(l) = \tau$ as well, and the rest follows by induction.
4. We are given that for each $L \in \texttt{dom}(H)$ that $X; \Phi; \cdot; \cdot \vdash H(L) : \Phi(L)$. It follows by 3 that $X \oplus X'; \Phi; \cdot \vdash H(L) : \Phi(L)$.

Corollary 6.4 *Suppose $X \uplus X'$ is well-defined.*

1. *If $X; \Delta \vdash \tau$ then $X \uplus X'; \Delta \vdash \tau$*
2. *If $X \vdash \Phi$ then $X \uplus X' \vdash \Phi$*
3. *If $X; \Phi; \Delta; \Gamma \vdash e : \tau$ then $X \uplus X'; \Phi; \Delta; \Gamma \vdash e : \tau$.*
4. *If $X \vdash H : \Phi$ then $X \uplus X' \vdash H : \Phi$*

Lemma 6.5 (Type Heap Redundancy Elimination) *If $X \leq X'$ and $X \oplus X'' \vdash \tau$ then $X \oplus (X'' - X') \vdash \tau$*

Proof

(Sketch) Any label in X'' that is also in X' will also be in X as that type heap contains all of the labels of X'. Therefore subtracting out the redundant labels will not interefere with type well-formedness.

6.2 Properties of Value Heaps

The Lemmas (and their corollaries) developed in this subsection are used in the proof of subject reduction for the load and `ref` cases.

Lemma 6.6 (Value Heap Weakening) *If $X \vdash \Phi$, $X \vdash H : \Phi$, $X; \Phi; \cdot; \cdot \vdash e : \tau$ and given some $L' \notin \texttt{dom}(\Phi)$ and some type τ' such that $X; \cdot \vdash \tau'$, then*

1. *$X \vdash (\Phi \uplus \{L' : \tau'\})$*
2. *$X \vdash H : (\Phi \uplus \{L' : \tau'\})$*
3. *$X; (\Phi \uplus \{L' : \tau'\}); \cdot; \cdot \vdash e : \tau$*

Proof

1. We must show that for all $\tau \in \texttt{rng}(\Phi \uplus \{L' : \tau'\})$, $X \vdash \tau$. If $\tau \in \texttt{rng}(\Phi)$, then this is true by inversion of $X \vdash \Phi$. If $tau \notin \texttt{rng}(\Phi)$ then $\tau = \tau'$, and we are given that $X; \cdot \vdash \tau'$.
2. This follows trivially by assumption, since we have not changed the domain of H.
3. Proof by induction on $X; \Phi; \cdot; \cdot \vdash e : \tau$. Follows trivially or by induction. In the abstraction and type application cases, we need to use 1 for the introduction of the new type, and for $e = L$, we have $\Phi(L) = \tau = (\Phi \uplus \{L' : \tau'\})(L)$.

Corollary 6.7 *If $X \vdash \Phi$ and $X \vdash H : \Phi$, and given some Φ' such that $X \vdash \Phi'$ and $\Phi \mid \Phi'$, then*

1. $X \vdash (\Phi \uplus \Phi')$

2. $X \vdash H : (\Phi \uplus \Phi')$

3. $X; (\Phi \uplus \Phi'); \cdot; \cdot \vdash e : \tau$

Lemma 6.8 (Value Heap Redundancy Elimination) *If $X \vdash H : \Phi$, and there exists some $L' \in \texttt{dom}(\Phi)$ s.t. $L' \notin \texttt{dom}(H)$, then $X \vdash H : \{L : \tau | L : \tau \in \Phi, L \neq L'\}$.*

Proof

We must show that for all $L \in \texttt{dom}(H), H(L) : (\{L : \tau | L : \tau \in \Phi, L \neq L'\})(L)$. But this is obvious, since we have only removed a label from Φ that was not in H.

6.3 Properties of Type Derivations

Lemma 6.9 (Type in Type Substitution) *If $X; \Delta, \alpha \vdash \tau$ and $X; \Delta \vdash \tau'$ then $X; \Delta \vdash \tau[\tau'/\alpha]$*

Proof

Proof by induction on $X; \Delta, \alpha \vdash \tau$

Case 1: $X; \Delta, \alpha \vdash l$ or $X; \Delta, \alpha \vdash \texttt{int}$ follows trivially.

Case 2: $X; \Delta, \alpha \vdash \alpha$ since by assumption $X; \Delta \vdash \alpha[\tau'/\alpha]$.

Remaining cases follow by simple induction.

The following lemma is used in the `ref` case of subject reduction.

Lemma 6.10 (Regularity)

If $X; \Phi; \Delta; \Gamma \vdash v : \tau$, $\vdash X$, $X; \Delta \vdash \Gamma$, and $X \vdash \Phi$, then $X; \Delta \vdash \tau$.

Proof

The proof proceeds by induction on the derivation $X; \Phi; \Delta; \Gamma \vdash e : \tau$.

Case 1: $X; \Phi; \Delta; \Gamma \vdash i : \texttt{int}$. Follows directly that $X; \Delta \vdash \texttt{int}$.

Case 2: $X; \Phi; \Delta; \Gamma \vdash L : \Phi(L)$. By assumption $X \vdash \Phi$, and by inversion $X; \cdot \vdash \Phi(L)$.

Case 3: $X; \Phi; \Delta; \Gamma \vdash y : \Gamma(y)$. By assumption $X; \Delta \vdash \Gamma$, so $X; \Delta \vdash \Gamma(y)$.

Case 4: $X; \Phi; \Delta; \Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau$. By inversion $X; \Delta \vdash \tau'$. As a result, $X; \Delta \vdash \Gamma, x{:}\tau'$, as τ' is well-formed. Therefore, by induction $X; \Delta \vdash \tau$. Thus, $X; \Delta \vdash \tau' \to \tau$.

Case 5: $X; \Phi; \Delta; \Gamma \vdash e_1\ e_2 : \tau$. By induction $X; \Delta \vdash \tau' \to \tau$, and by inversion $X; \Delta \vdash \tau$.

Case 6: $X; \Phi; \Delta; \Gamma \vdash \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3 : \tau$. By induction.

Case 7: $X; \Phi; \Delta; \Gamma \vdash \texttt{roll}_l\ e : l$. By the rule side-condition $X(l) = \tau$, thus $X; \Delta \vdash l$.

Case 8: $X; \Phi; \Delta; \Gamma \vdash \texttt{unroll}\ e : \tau$. By the rule side-condition $X(l) = \tau$, and by assumption $\vdash X$, so $X; \cdot \vdash \tau$. By weakening, $X; \Delta \vdash \tau$.

Case 9: $X; \Phi; \Delta; \Gamma \vdash \texttt{ref}\ e : \tau\ \texttt{ref}$. By induction $X; \Delta \vdash \tau$, so $X; \Delta \vdash \tau\ \texttt{ref}$ follows directly.

Case 10: $X; \Phi; \Delta; \Gamma \vdash !e : \tau$. By induction $X; \Delta \vdash \tau\ \texttt{ref}$, and by induction again $X; \Delta \vdash \tau$.

Case 11: $X; \Phi; \Delta; \Gamma \vdash \texttt{assign}\ e_1 e_2 : \tau$. By induction.

Case 12: $X; \Phi; \Delta; \Gamma \vdash e : \forall\alpha.\tau$. By induction.

Case 13: $X; \Phi; \Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]$, so by inversion $X; \Phi; \Delta, \alpha; \Gamma \vdash e : \tau$ and $X; \Delta \vdash \tau'$. By induction $X; \Delta, \alpha \vdash \tau$ and by type in type substitution, $X; \Delta \vdash \tau[\tau'/\alpha]$.

The following two lemmas are used in the proof of substitution, also below.

Lemma 6.11 (Weakening) *If $X; \Phi; \Delta; \Gamma \vdash e : \tau$ and $x \notin \texttt{dom}(\Gamma)$ and $\alpha \notin \texttt{dom}(\Delta)$, then $X; \Phi; \Delta; \Gamma, x{:}\tau' \vdash e : \tau$, and $X; \Phi; \Delta, \alpha; \Gamma \vdash e : \tau$. Moreover, the latter derivations have the same depth as the former.*

Lemma 6.12 (Permutation) *If $X; \Phi; \Delta; \Gamma \vdash e : \tau$ with Γ' is a permutation of Γ and Δ' is a permutation of Δ, then $X; \Phi; \Delta; \Gamma' \vdash e : \tau$, and $X; \Phi; \Delta'; \Gamma \vdash e : \tau$. Moreover, the latter derivations have the same depth as the former.*

Lemma 6.13 (Substitution) *If* $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash e : \tau$ *and* $X;\Phi;\Delta;\Gamma \vdash e' : \tau'$ *then* $X;\Phi;\Delta;\Gamma \vdash e[e'/x] : \tau$.

Proof

Proof is by induction on $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash e : \tau$.

Case 1: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash i : \mathtt{int}$

Therefore $e[e'/x] = i$, and $X;\Phi;\Delta;\Gamma \vdash i : \mathtt{int}$.

Case 2: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash L : \Phi(L)$.

Therefore $e[e'/x] = L$, and $X;\Phi;\Delta;\Gamma \vdash L : \Phi(L)$.

Case 3: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash y : (\Gamma,x{:}\tau')(y)$.

If $y = x$ then $y[e'/x] = e'$. By assumption, $X;\Phi;\Delta;\Gamma \vdash e' : \tau'$, and the result follows from $\tau = \tau'$. Otherwise, $y[e'/x] = y$ and $X;\Phi;\Delta;\Gamma \vdash y : \Gamma(y)$.

Case 4: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash \lambda y{:}\tau''.e : \tau'' \to \tau$

Follows by induction (with Weakening and Permutation): $X;\Phi;\Delta;\Gamma,y{:}\tau'' \vdash e[e'/x] : \tau$. Therefore, $X;\Phi;\Delta;\Gamma \vdash (\lambda y{:}\tau''.e)[e'/x] : \tau'' \to \tau$.

Case 5: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash e_1\ e_2 : \tau$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e_1[e'/x] : \tau'' \to \tau$ and $X;\Phi;\Delta;\Gamma \vdash e_2[e'/x] : \tau''$. Therefore, $X;\Phi;\Delta;\Gamma \vdash (e_1 e_2)[e'/x] : \tau$.

Case 6: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash \mathsf{load}[\tau'']\ e_0\ e_1\ e_2\ e_3 : \tau$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e_0[e'/x] : \mathtt{int}$,$X;\Phi;\Delta;\Gamma \vdash e_1[e'/x] : \mathtt{int}$, $X;\Phi;\Delta;\Gamma \vdash e_2[e'/x] : \tau'' \to \tau$, and $X;\Phi;\Delta;\Gamma \vdash e_3[e'/x] : \tau$, so therefore $X;\Phi;\Delta;\Gamma \vdash (\mathsf{load}[\tau'']\ e_1\ e_2\ e_3)[e'/x] : \tau$.

Case 7: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash \mathtt{roll}_l\ e : l$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e[e'/x] : \tau''$, so $X;\Phi;\Delta;\Gamma \vdash (\mathtt{roll}_l\ e)[e'/x] : l$

Case 8: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash \mathtt{unroll}\ e : \tau$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e[e'/x] : l$, so $X;\Phi;\Delta;\Gamma \vdash (\mathtt{unroll}\ e)[e'/x] : \tau$.

Case 9: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash \mathtt{ref}\ e : \tau\ \mathtt{ref}$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e[e'/x] : \tau$, so $X;\Phi;\Delta;\Gamma \vdash (\mathtt{ref}\ e)[e'/x] : \tau\ \mathtt{ref}$.

Case 10: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash\ !e : \tau$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e[e'/x] : \tau\ \mathtt{ref}$, so $X;\Phi;\Delta;\Gamma \vdash (!e)[e'/x] : \tau$.

Case 11: $X;\Phi;\Delta;\Gamma,x{:}\tau' \vdash \mathtt{assign}\ e_1 e_2 : \tau$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e_1[e'/x] : \tau\ \texttt{ref}$ and $X;\Phi;\Delta;\Gamma \vdash e_2[e'/x] : \tau$. Therefore, $X;\Phi;\Delta;\Gamma \vdash (\texttt{assign}\ e_1e_2)[e'/x] : \tau$.

Case 12: $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash \Lambda\alpha.e : \forall\alpha.\tau$. Follows by induction (with Weakening): $X;\Phi;\Delta,\alpha;\Gamma \vdash e[e'/x] : \tau$, so $X;\Phi;\Delta,\alpha;\Gamma \vdash \Lambda\alpha.e[e'/x] : \forall\alpha.\tau$.

Case 13: $X;\Phi;\Delta;\Gamma, x{:}\tau' \vdash e[\tau'] : \tau[\tau'/\alpha]$. Follows by induction: $X;\Phi;\Delta,\alpha;\Gamma \vdash e[e'/x] : \forall\alpha.\tau$, so $X;\Phi;\Delta;\Gamma \vdash (e[e'/x])[\tau'] : \tau[\tau'/\alpha]$.

Lemma 6.14 (Type Substitution) *If* $\vdash X$ *and* $X;\Phi;\Delta,\alpha;\Gamma \vdash e : \tau$ *and* $X;\Delta \vdash \tau'$ *then* $X;\Phi;\Delta;\Gamma \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$.

Proof
Proof is by induction on $X;\Phi;\Delta,\alpha;\Gamma \vdash e : \tau$. Most cases are trivial or by induction. Selected cases:

Case 1: $X;\Phi;\Delta,\alpha;\Gamma \vdash \mathsf{load}[\tau'']\ e_1\ e_2\ e_3 : \tau$.

By induction: $X;\Phi;\Delta;\Gamma \vdash e_0[\tau'/\alpha] : \texttt{int}$, $X;\Phi;\Delta;\Gamma \vdash e_1[\tau'/\alpha] : \texttt{int}$, $X;\Phi;\Delta;\Gamma \vdash e_2[\tau'/\alpha] : (\tau'' \to \tau)[\tau'/\alpha]$, and $X;\Phi;\Delta;\Gamma \vdash e_3[\tau'/\alpha] : \tau[\tau'/\alpha]$. By type in type substitution $X;\Delta \vdash \tau''[\tau'/\alpha]$ so therefore $X;\Phi;\Delta;\Gamma \vdash (\mathsf{load}[\tau'']\ e_0\ e_1\ e_2\ e_3)[\tau'/\alpha] : \tau[\tau'/\alpha]$.

Case 2: $X;\Phi;\Delta,\alpha;\Gamma \vdash \texttt{roll}_l\ e : l$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e[\tau'/\alpha] : \tau''[\tau'/\alpha]$. As $X(l) = \tau''$ and $\vdash X$ then $X;\cdot \vdash \tau''$. Therefore $X;\Phi;\Delta;\Gamma \vdash (\texttt{roll}_l\ e)[\tau'/\alpha] : l[\tau'/\alpha]$ since τ'' must be closed.

Case 3: $X;\Phi;\Delta,\alpha;\Gamma \vdash \texttt{unroll}\ e : \tau[\tau'/\alpha]$.

Follows by induction: $X;\Phi;\Delta;\Gamma \vdash e[\tau'/\alpha] : l[\tau'/\alpha]$, so $X;\Phi;\Delta;\Gamma \vdash (\texttt{unroll}\ e)[\tau'/\alpha] : \tau[\tau'/\alpha]$.

The following lemma is used in the proof of progress, to develop type soundness.

Lemma 6.15 (Canonical Forms) *If* $X;\Phi;\cdot \vdash v : \tau$ *and*

- $\tau = \texttt{int}$ *then* $v = i$.
- $\tau = \tau_1 \to \tau_2$ *then* $v = \lambda x{:}\tau.e$.
- $\tau = l$ *then* $v = \texttt{roll}_l(v')$ *for some* v'.
- $\tau = \tau\ \texttt{ref}$ *then* $v = L$.
- $\tau = \forall\alpha.\tau$ *then* $v = \Lambda\alpha.e$.

Proof
Proof is by examination of the last step of the typing derivation $X;\Phi;\cdot;\cdot \vdash v : \tau$. Most rules either require the expression to be a non-value or require a non-empty context. The remaining rules produce each of the types at the correct values.

6.4 Type Soundness

We establish type soundness in the standard manner, by proof of subject reduction and progress.

Lemma 6.16 (Subject Reduction) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto (\Theta', H', e')$ *then* $\vdash (\Theta', H', e') : \tau$

Proof

$\vdash (\Theta', H', e') : \tau$ is proven by showing that, for some Φ'

- (Type-heap well-formedness) $\vdash X'_I \uplus X'_H$
- (Value-heap typing well-formedness) $X'_I \uplus X'_H \vdash \Phi'$
- (Value-heap well-formedness) $X'_I \uplus X'_H \vdash H' : \Phi'$
- (Expression well-formedness) $X'_I \uplus X'_H; \Phi'; \cdot; \cdot \vdash e' : \tau$

For brevity, we refer to these points in the proof as *THWF*, *VHTWF*, *VHWF*, and *EWF*, respectively, and except when otherwise noted, we assume that $\Phi' = \Phi$ and that *THWF*, *VHTWF*, *VHWF* hold by assumption. The proof is by induction on the typing derivation $\vdash (\Theta, H, e)$ (for some Φ), and on $(\Theta, H, e) \mapsto (\Theta', H', e')$.

Case 1: (beta) $(\Theta, H, (\lambda x{:}\tau.e'')v) \mapsto (\Theta, H, e''[v/x])$.

As e is an application, by inversion $X_I \uplus X_H; \Phi; x{:}\tau' \vdash e'' : \tau$, and $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau'$. *EWF* follows by substitution: $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e''[v/x] : \tau$.

Case 2: (load-success) $((X^1_I, X^1_H), H, \mathsf{load}[\tau']h\ i\ e_2\ e_3) \mapsto ((X^3_I, X^3_H), H_3, e_2\ e)$

We must establish each of *THWF*, *VHTWF*, *VHWF*, and *EWF*. We know the following facts. From this evaluation rule:

1. $\hat{h} = X$
2. $\hat{i} = ((X^2_I, X^2_H), H_2, e)$
3. $X^1_H \vdash \hat{i} : \tau'$
4. $(X^1_I, X)\ \texttt{link}\ (X^2_I, X^2_H) \Rightarrow (X^3_I, X')$
5. $X^3_I = X' \oplus X^1_H$
6. $H_1\ \texttt{link}\ H_2 \Rightarrow H_3$
7. $X^1_H \leq X$
8. $X^2_I \mid (X^1_H - X)$

   As linking is well-formed for both value heaps (by 6) and type heaps (by 4):

9. $X_I^1 \sim X_I^2$

10. $X \mid X_H^2$

11. $X_H^2 \precsim X_I^1$

12. $X \precsim X_I^2$

13. $H_1 \mid H_2$

As the loaded program is well-formed (by 3), for some Φ_2:

14. $\vdash X_I^2 \uplus X_H^2$

15. $X_I^2 \uplus X_H^2 \vdash \Phi_2$

16. $X_I^2 \uplus X_H^2 \vdash H_2 : \Phi_2$

17. $X_I^2 \uplus X_H^2; \Phi_2; \cdot; \cdot \vdash e : \tau'$

18. $X_H^2 \mid X_H^1$

Since the running program is well-formed, for some Φ_1:

19. $\vdash X_I^1 \uplus X_H^1$

20. $X_I^1 \uplus X_H^1 \vdash \Phi_1$

21. $X_I^1 \uplus X_H^1 \vdash H_1 : \Phi_1$

22. $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash \mathsf{load}[\tau'] i e_2 e_3 : \tau$

By inversion of this last expression's typing judgement:

23. $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash i : \texttt{int}$

24. $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash e_2 : \tau' \rightarrow \tau$

25. $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash e_3 : \tau$

We define Φ_1 as the least Φ_1 that satisfies 21, and Φ_2 as the least Φ_2 that satisfies 16. Finally, we define Φ_3 as $\Phi_1 \uplus \Phi_2$, which is well-defined as $\texttt{dom}(H_1) = \texttt{dom}(\Phi_1)$, $\texttt{dom}(H_2) = \texttt{dom}(\Phi_2)$, and $\vdash H_1 \mid H_2$ by 19.

To prove well-formedness of the new program we must establish:

- *(THWF):* $\vdash X_I^3 \uplus X_H^3$
  By definition, this is $((X_I^1 \oplus X_I^2) - (X \uplus X_H^2)) \uplus ((X \uplus X_H^2) \oplus X_H^1)$. The expression $(X_I^1 \oplus X_I^2)$ is well-defined by 9, $(X \uplus X_H^2)$ is well-defined by 10. As $X_H^1 \leq X$ and $X \mid X_H^2$ then $((X \uplus X_H^2) \oplus X_H^1)$ is well-defined, and equal to $X_H^2 \oplus X_H^1$. Finally, the whole thing is well-defined if there is no label defined in $X_I^1 \oplus X_I^2$, not defined in X but defined in X_H^1. However, 8 and 19 guarantee that fact.
  Therefore, $X_I^3 \uplus X_H^3$

$= ((X_I^1 \oplus X_I^2) - (X \uplus X_H^2)) \uplus (X_H^2 \oplus X_H^1)$	by reasoning above
$= ((X_I^1 \oplus X_I^2) - (X_H^1 \oplus X_H^2)) \uplus (X_H^2 \oplus X_H^1)$	as $X_I^2 \mid (X_H^1 - X)$ and $X_I^1 \mid X_H^1$ the restriction from type mask X can't remove more labels
$= (X_I^1 \oplus X_I^2) \oplus (X_H^1 \oplus X_H^2)$	by Lemma 6.1 (5), since $(X_H^1 \oplus X_H^2) \precsim (X_I^2 \oplus X_I^1)$ by 11, 12, and Lemma 6.1 (4) $= (X_H^1 \oplus X_H^2) \precsim (X_I^1 \oplus X_I^2)$ by commutativity
$= (X_I^1 \oplus X_H^1) \oplus (X_I^2 \oplus X_H^2)$	by associativity and commutivity

By 19, $\vdash (X_I^1 \oplus X_H^1)$, and by 14, $\vdash (X_I^2 \oplus X_H^2)$. So by lemma 6.2, the whole thing is well formed.

- *(VHTWF):* $X_I^3 \uplus X_H^3 \vdash \Phi_3$
  This is equivalent to $X_I^3 \uplus X_H^3 \vdash (\Phi_1 \uplus \Phi_2)$. Consider some $L : \tau \in \Phi_3$; there are two possibilities:
  1. $L : \tau \in \Phi_1$. By 20, $X_I^1 \uplus X_H^1 \vdash \Phi_1$, so $X_I^1 \uplus X_H^1 \vdash \tau$. As $X_I^1 \mid X_H^1$, this is equivalently $X_I^1 \oplus X_H^1 \vdash \tau$. By 6.3 (type heap weakening), $(X_I^1 \oplus X_H^1) \oplus (X_I^2 \oplus X_H^2) \vdash \tau$, which we have shown in the proof of *THWF* is equivalent to $X_I^3 \uplus X_H^3 \vdash \tau$.
  2. $L : \tau \in \Phi_2$. By 15, $X_I^2 \uplus X_H^2 \vdash \Phi_2$, so $X_I^2 \uplus X_H^2 \vdash \tau$. By similar weakening as above we may conclude $X_I^3 \uplus X_H^3 \vdash \tau$.
- *(VHWF):* $X_I^3 \uplus X_H^3 \vdash H_3 : \Phi_3$ This is equivalent to $X_I^3 \uplus X_H^3 \vdash (H_1 \uplus H_2) : (\Phi_1 \uplus \Phi_2)$. Consider some $L \in H_3$; there are two possibilities:
  1. $(L = v) \in H_1$. By 21, $X_I^1 \uplus X_H^1 \vdash H_1 : \Phi_1$, so $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash v : \tau$, where $\Phi_1(L) = \tau$. By 6.3 as in VHTWF, $(X_I^1 \oplus X_H^1) \oplus (X_I^2 \oplus X_H^2); \Phi_1; \cdot; \cdot \vdash v : \tau$.
  2. $(L = v) \in H_2$. $X_I^3 \uplus X_H^3; \Phi_3; \cdot; \cdot \vdash v : \Phi_3(L)$ follows by similar reasoning.
- *(EWF):* $X_I^3 \uplus X_H^3; \Phi_3; \cdot; \cdot \vdash e_2\ e : \tau$
  We know $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash e_2 : \tau' \to \tau$ and $X_I^1 \uplus X_H^1; \Phi_1; \cdot; \cdot \vdash e_3 : \tau$. By the same weakening argument as above, $X_I^3 \uplus X_H^3; \Phi_3; \cdot; \cdot \vdash e_2 : \tau' \to \tau$, and $X_I^3 \uplus X_H^3; \Phi_3; \cdot; \cdot \vdash e_3 : \tau$. Therefore, we may conclude our well-formedness result.

Case 3: (load-failure) $((X_I, X_H), H_1, \mathsf{load}[\tau']\ h\ i\ e_2\ e_3) \mapsto ((X_I, X_H), H_1, e_3)$

EWF follows directly as $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e_3 : \tau$.

Case 4: (unroll) $((X_I, X_H), H, \texttt{unroll}(\texttt{roll}_l\ v)) \mapsto ((X_I, X_H), H, v)$

We must have concluded (during the typing derivation of $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \texttt{unroll}(\texttt{roll}_l\ v) : \tau$) that $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \texttt{roll}_l\ v : l$ (where $(X_I \uplus X_H)(l) = \tau$), and again $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau$, which proves *EWF*.

Case 5: (ref) $((X_I, X_H), H, \texttt{ref}\ v) \mapsto ((X_I, X_H), H \uplus \{L = v\}, L)$

THWF follows by assumption. We show *VHTWF* and *VHWF* as follows. Consider the typing derivation of $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \texttt{ref}\ v : \tau$: by inversion $\tau = \tau'\ \texttt{ref}$ and $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau'$, for some Φ. We may assume that $L \notin \texttt{dom}(\Phi)$ by value-heap redundancy elimination since $L \notin H$ (by the side-condition on the evaluation rule). Therefore we choose $\Phi' = \Phi \uplus \{L : \tau'\}$.

To show *VHTWF*, we must show that $X_I \uplus X_H \vdash \Phi'$. Consider an arbitrary $L' \in \texttt{dom}(\Phi')$:

- if $L' \in \texttt{dom}(\Phi)$ then $X_I \uplus X_H \vdash (\Phi \uplus \{L : \tau'\})(L')$ by assumption and value-heap weakening.
- if $L' = L$ then to show $X_I \uplus X_H \vdash (\Phi \uplus \{L : \tau'\})(L')$, we must show that $X_I \uplus X_H \vdash \tau'$. This follows because by the typing derivation of $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash \texttt{ref}\ v : \tau$ we must have previously concluded that $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau'$. By Lemma 6.10, $X_I \uplus X_H \vdash \tau'$.

To show *VHWF*, we must show that $X_I \uplus X_H \vdash (H \uplus \{L = v\}) : \Phi'$. Consider an arbitrary $L' \in \texttt{dom}(H \uplus \{L = v\})$:

- if $L' \in \texttt{dom}(H)$ then $X_I \uplus X_H \vdash H(L') : (\Phi \uplus \{L : \tau'\})(L')$ by assumption and value-heap weakening.
- if $L' = L$ then to show $X_I \uplus X_H \vdash H(L) : (\Phi \uplus \{L : \tau'\})(L')$, we must show that $X_I \uplus X_H \vdash v : \tau'$. But this follows by assumption, as noted above.

Finally, to show *EWF*, we note that $X_I \uplus X_H; (\Phi \uplus \{L : \tau'\}); H \uplus \{L = v\} \vdash L : \tau'\ \texttt{ref}$.

Case 6: (deref) $((X_I, X_H), H, !L) \mapsto ((X_I, X_H), H, H(L))$.

By inversion, $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash L : \tau\ \texttt{ref}$, and furthermore that $\Phi(L) = \tau$. By program well-formedness, $X_I \uplus X_H \vdash H : \Phi$, which implies that $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash H(L) : \Phi(L) = \tau$, which is the desired result.

Case 7: (assign) $((X_I, X_H), H, \texttt{assign}\ L\ v) \mapsto ((X_I, X_H), H[L = v], v)$.

THWF and *VHTWF* follow by assumption for $\Phi' = \Phi$. To show *VHWF*, we must show that $X \uplus X_H \vdash H[L = v] : \Phi'$. Consider some $L' \in H[L = v]$:

- if $L' \neq L$, then $X_I \uplus X_H \vdash (H[L = v])(L') : \Phi'(L')$ follows by assumption (since H has not changed at these labels).
- if $L' = L$, then we must show that $X_I \uplus X_H \vdash v : \Phi'(L)$. But by inversion $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash L : \tau\ \texttt{ref}$ which implies (again by inversion) that $\Phi'(L) = \tau$. Also by inversion $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash v : \tau$, which gives the desired result.

Finally, for EWF we must show that $X_I \uplus X_H; \Phi'; \cdot; \cdot \vdash v : \tau$. This follows trivially by inversion.

Case 8: (tapp) $(\Theta, H, (\Lambda\alpha.e)[\tau]) \mapsto (\Theta, H, e[\tau/\alpha])$, of type $\tau'[\tau/\alpha]$.

By inversion, $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash (\Lambda\alpha.e) : \forall\alpha.\tau$ and $X_I \uplus X_H; \cdot; \cdot \vdash \tau'$. Doing this again we get $X_I \uplus X_H; \Phi; \alpha; \cdot; \cdot \vdash e : \tau$. We may now apply type substitution to conclude $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$

Case 9: (congruence rules) Follow by induction of $(\Theta, H, e) \mapsto (\Theta', H', e')$.

Lemma 6.17 *If $X_I \uplus X_H$, $X_I \uplus X_H \vdash \Phi$, $X_I \uplus X_H \vdash H : \Phi$, $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e : \tau$, and e is not a value, then there exists an $((X'_I, X'_H), H', e')$ such that $((X_I, X_H), H, e) \mapsto ((X'_I, X'_H), H', e')$.*

Proof

Proof is by induction on $X_I \uplus X_H; \Phi; \cdot; \cdot \vdash e : \tau$ and on $((X_I, X_H), H, e) \mapsto ((X'_I, X'_H), H', e')$. We will only consider the expression typing rules in which e is not a value:

Case 1: (app) $e = e_1\ e_2$

Three cases:

- e_1 is not a value
  By induction, there exists an $((X'_I, X'_H), H', e'_1)$ such that $((X_I, X_H), H, e_1) \mapsto ((X'_I, X'_H), H', e'_1)$. By congruence, $((X_I, X_H), H, e_1\ e_2) \mapsto ((X'_I, X'_H), H', e'_1\ e_2)$.
- e_2 is not a value
  By induction, there exists an $((X'_I, X'_H), H', e'_2)$ such that $((X_I, X_H), H, e_2) \mapsto ((X'_I, X'_H), H', e'_2)$. By congruence, $((X_I, X_H), H, e_1\ e_2) \mapsto ((X'_I, X'_H), H', e_1\ e'_2)$.
- e_1 and e_2 are values.
  By canonical forms, $e_1 = \lambda x : \tau'.e$. Therefore, by beta reduction, $((X_I, X_H), H, e_1\ e_2)$ steps to $((X_I, X_H), H', e[e_2/x])$.

Case 2: (load) $e = \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3$

If either of the first two arguments is not a value, then by induction there exists a $((X'_I, X'_H), H', e'_i)$ such that $((X_I, X_H), H, e_1) \mapsto ((X'_I, X'_H), H', e'_i)$. Therefore, by congruence, load can take a step.

Otherwise, e_0 and e_1 are values and by canonical forms, some integers h, i. If the conditions for load-success hold (i.e. h is the representation of a type heap and i is the representation of well-typed program that is link-compatible with $\hat{h}$ and the current type heap) then $((X_I, X_H), H, \mathsf{load}[\tau']\ e_1\ e_2\ e_3) \mapsto ((X'_I, X'_H), H', e_2\ e)$.

If not, the load-fail step rule applies and $((X_I, X_H), H, \mathsf{load}[\tau']\ e_0\ e_1\ e_2\ e_3) \mapsto ((X_I, X_H), H, e_3)$.

Case 3: (unroll) $e = \texttt{unroll}\ e_1$.

If e_1 is not a value, congruence rule applies. Otherwise e_1 must be a value of type l, so by canonical forms, $e_1 = \texttt{roll}_l(v)$. By the unroll reduction, $((X_I, X_H), H, \texttt{unroll}(\texttt{roll}_l v)) \mapsto ((X_I, X_H), H, v)$.

Case 4: (ref) $e = \texttt{ref}\ e_1$.

If e_1 is not a value, congruence rule applies. Otherwise, by the ref reduction, $((X_I, X_H), H, \texttt{ref}\ v) \mapsto ((X_I, X_H), H \uplus \{L = v\}, L)$.

Case 5: (deref) $e = !e_1$.

If e_1 is not a value, congruence rule applies. Otherwise e_1 must be a value of type $\tau'\texttt{ref}$, so by canonical forms, $e_1 = L$. By inversion, $L \in \texttt{dom}(H)$, and by the deref reduction, $((X_I, X_H), H, !L) \mapsto ((X_I, X_H), H, H(L))$.

Case 6: (assign) $e = \texttt{assign}\ \ e_1\ e_2$.

If e_1 and/or e_2 are not values, congruence rule applies. Otherwise, e_1 is a value of type $\tau'\texttt{ref}$, so by canonical forms, $e_1 = L$. By inversion, $L \in \texttt{dom}(H)$, and by the assign reduction, $(\Theta, H, \texttt{assign}\ Lv) \mapsto (\Theta, H[L = v], v)$.

Case 7: (tapp) $e = e_1[\tau]$.

If e_1 is not a value, congruence rule applies. Otherwise e_1 is of type $\forall\alpha.\tau$, so by canonical forms, $e_1 = \Lambda\alpha.e'$, so by tapp reduction $(\Theta, H, e_1[\tau]) \mapsto (\Theta, H.e'[\tau/\alpha])$.

Corollary 6.18 (Progress) *If* $\vdash (\Theta, H, e) : \tau$ *and* e *is not a value, then there exists a* (Θ', H', e') *such that* $(\Theta, H, e) \mapsto (\Theta', H', e')$.

We say that a term is *stuck* if it is not a value and if no rule of the operational semantics applies to it. Type safety requires that no well-typed term can become stuck:

Theorem 6.19 (Type Safety) *If* $\vdash (\Theta, H, e) : \tau$ *and* $(\Theta, H, e) \mapsto^* (\Theta', H', e')$ *then* (Θ', H', e') *is not stuck.*

Proof

Proof is by induction on the number of steps of execution $((\Theta, H, e) \mapsto^* (\Theta', H', e'))$ using Progress to show there is a new state and Subject Reduction to show that that new state is well typed.

References

[1] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the ACM SIGPLAN Symposium on the Principles of Programming Languages*, pages 266–277. ACM, January 1997.

[2] Dominic Duggan. Sharing in Typed Module Assembly Language. In *Preliminary Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, Technical Report CMU-CS-00-161. Carnegie Mellon University, September 2000.

[3] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the ACM SIGPLAN Symposium on the Principles of Programming Languages*, pages 250–261. ACM, January 1999.

[4] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In *Preliminary Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, Technical Report CMU-CS-00-161. Carnegie Mellon University, September 2000.

[5] Xavier Leroy. *The Objective Caml System, Release 3.00.* Institut National de Recherche en Informatique et Automatique (INRIA), 2000. Available at `http://caml.inria.fr`.

[6] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[7] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.