A PROGRAMMING LANGUAGE DEFINITIONS

The programming languages used throughout the paper mostly follows the presentation in the book Types and Programming Languages by Pierce [2002] with minor changes. In particular, italics are used for metavariables and the axioms in the reduction (evaluation) rules and typing rules are shown with explicitly empty premises.

A.1 UntypedLambda

Figure 21 shows the syntax and evaluation rules for the untyped lambda calculus by Church [1936, 1941], that we have referred to as UntypedLambda. This language is used in Sections 2.1, 2.2, and 3.1 .

Syntax

$$
\begin{array}{rlr}
t ::= & & \text{terms:} \\
 & x & \text{variable} \\
 & \mid \lambda x.t & \text{abstraction} \\
 & \mid t\ t & \text{application} \\
 & & \\
v ::= & & \text{values:} \\
 & \lambda x.t &
\end{array}
$$

Evaluation

$$\frac{}{(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ \text{E-AppAbs}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{E-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{E-App2}$$

Fig. 21. The untyped lambda calculus (UntypedLambda).

A.2 TypedArith

We define here the language TypedArith, used in Section 2.4. Figure 22 shows its syntax and evaluation (reduction) rules and Figure 23 shows its typing rules. The appeal of using this language to present a notional machine focused on the types is its simplicity. Terms don't require type annotations and the typing rules don't require a type environment. In fact, Pierce uses it as the simplest example of a typed language when introducing type safety.

A.3 TypedLambdaRef

In Section 2.3, we showed the language TypedLambdaRef, used to design a notional machine that focuses on references. This language is composed of the simply-typed lambda calculus, the TypedArith language, tuples, the `Unit` type, sequencing, and references. Our goal is again simplicity and this is the simplest language we need for the examples in the book that use the diagram. Figure 24 shows its syntax and evaluation (reduction) rules. We show only the reduction rules for sequencing, references, and tuples because the rules for the rest of the language are similar to what we showed before, except for the store then needs to be threaded through all the rules. Although that is a typed language, we don't present its typing rules because the notional machine in Section 2.3 is focused only on its runtime behavior and not its types.

Syntax

$$
\begin{array}{rcll}
t & ::= & & \text{terms:} \\
& & \texttt{true} & \text{constant true} \\
& | & \texttt{false} & \text{constant false} \\
& | & \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t & \text{conditional} \\
& | & 0 & \text{constant zero} \\
& | & \texttt{succ}\ t & \text{successor} \\
& | & \texttt{pred}\ t & \text{predecessor} \\
& | & \texttt{iszero}\ t & \text{zero test} \\
\\
v & ::= & & \text{values:} \\
& & \texttt{true} & \\
& | & \texttt{false} & \\
& | & nv & \\
\\
nv & ::= & & \text{numeric values:} \\
& & 0 & \\
& | & \texttt{succ}\ nv &
\end{array}
$$

Evaluation

$$\frac{}{\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_2}\ \textsc{E-IfTrue}$$

$$\frac{}{\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_3}\ \textsc{E-IfFalse}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \longrightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}\ \textsc{E-If}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'}\ \textsc{E-Succ}$$

$$\frac{}{\texttt{pred}\ 0 \longrightarrow 0}\ \textsc{E-PredZero}$$

$$\frac{}{\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1}\ \textsc{E-PredSucc}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'}\ \textsc{E-Pred}$$

$$\frac{}{\texttt{iszero}\ 0 \longrightarrow \texttt{true}}\ \textsc{E-IszeroZero}$$

$$\frac{}{\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false}}\ \textsc{E-IszeroSucc}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'}\ \textsc{E-IsZero}$$

Fig. 22. Syntax and reduction rules of the TypedArith language.

Syntax

T	::=		types:
		Bool	type of booleans
	|	Nat	type of natural numbers

Typing rules

$$\frac{}{\texttt{true} : \texttt{Bool}}\ \text{T-True}$$

$$\frac{}{\texttt{false} : \texttt{Bool}}\ \text{T-False}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}\ \text{T-If}$$

$$\frac{}{\texttt{0} : \texttt{Nat}}\ \text{T-Zero}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ}\ t_1 : \texttt{Nat}}\ \text{T-Succ}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred}\ t_1 : \texttt{Nat}}\ \text{T-Pred}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero}\ t_1 : \texttt{Bool}}\ \text{T-IsZero}$$

Fig. 23. Syntax of types and typing rules of the TypedArith language.

Syntax

t ::=		terms:
	x	variable
	| $\lambda x : T . t$	abstraction
	| $t\ t$	application
	| `true`	boolean true
	| `false`	boolean false
	| `0`	zero
	| `succ` t	successor
	| `pred` t	predecessor
	| `iszero` t	zero test
	| `unit`	unit constant
	| $t;\ t$	sequence
	| `ref` t	reference creation
	| $!t$	dereference
	| $t\ := t$	assignment
	| l	location
	| $\{t_i{}^{i \in 1..n}\}$	tuple
	| $t.i$	projection
v ::=		values:
	$\lambda x : T . t$	
	| `true`	
	| `false`	
	| `0`	
	| `succ` v	
	| `unit`	
	| l	
	| $\{v_i{}^{i \in 1..n}\}$	
T ::=		types:
	$T \rightarrow T$	function type
	| `Bool`	boolean type
	| `Nat`	natural number type
	| `Unit`	unit type
	| `Ref` T	reference type
	| $\{T_i{}^{i \in 1..n}\}$	tuple type
μ ::=		store:
	$\emptyset$	empty store
	| $\mu, l \mapsto v$	location binding

Evaluation

$$\frac{t_1 | \mu \longrightarrow t_1' | \mu'}{t_1;\ t_2 | \mu \longrightarrow t_1';\ t_2 | \mu'} \ \textsc{E-Seq}$$

$$\frac{}{\texttt{unit};\ t_2 | \mu \longrightarrow t_2 | \mu} \ \textsc{E-SeqNext}$$

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ v_1 | \mu \longrightarrow l | (\mu, l \mapsto v_1)} \ \textsc{E-RefV}$$

$$\frac{t_1 | \mu \longrightarrow t_1' | \mu'}{\texttt{ref}\ t_1 | \mu \longrightarrow \texttt{ref}\ t_1' | \mu'} \ \textsc{E-Ref}$$

$$\frac{\mu(l) = v}{!l | \mu \longrightarrow v | \mu} \ \textsc{E-DerefLoc}$$

$$\frac{t_1 | \mu \longrightarrow t_1' | \mu'}{!t_1 | \mu \longrightarrow !t_1' | \mu'} \ \textsc{E-Deref}$$

$$\frac{}{l\ := v_2 | \mu \longrightarrow \texttt{unit} | [l \mapsto v_2]\mu} \ \textsc{E-Assign}$$

$$\frac{t_1 | \mu \longrightarrow t_1' | \mu'}{t_1\ := t_2 | \mu \longrightarrow t_1'\ := t_2 | \mu'} \ \textsc{E-Assign1}$$

$$\frac{t_2 | \mu \longrightarrow t_2' | \mu'}{v_1\ := t_2 | \mu \longrightarrow v_1\ := t_2' | \mu'} \ \textsc{E-Assign2}$$

$$\frac{}{\{v_i{}^{i \in 1..n}\}.j | \mu \longrightarrow v_j | \mu} \ \textsc{E-ProjTuple}$$

$$\frac{t_1 | \mu \longrightarrow t_1' | \mu'}{t_1.i | \mu \longrightarrow t_1'.i | \mu'} \ \textsc{E-Proj}$$

$$\frac{t_j | \mu \longrightarrow t_j' | \mu'}{\begin{array}{l}\{v_i{}^{i \in 1..j-1}, t_j, t_k{}^{k \in j+1..n}\} | \mu \\ \longrightarrow \{v_i{}^{i \in 1..j-1}, t_j', t_k{}^{k \in j+1..n}\} | \mu'\end{array}} \ \textsc{E-Tuple}$$

Fig. 24. TypedLambdaRef: Syntax and Evaluation