1 Language $\mathcal{L}$

In this section we define an imperative language $\mathcal{L}$ with memory allocation and deallocation primitives, and for simplification we only use pointers as values.

The syntax of the language $\mathcal{L}$ is as follows.

$$
\begin{array}{rcl}
x, y, z, \ldots \text{ (variables)} & \in & \mathbf{Var} \\
s \text{ (statements)} & ::= & \mathbf{skip} \mid s_1; s_2 \mid *x \leftarrow y \mid \mathbf{free}(x) \\
 & \mid & \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \mid \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s \\
 & \mid & \mathbf{let}\ x = y\ \mathbf{in}\ s \mid \mathbf{let}\ x = *y\ \mathbf{in}\ s \\
 & \mid & \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid f(\vec{x}) \\
 & \mid & \mathbf{const}(*x)s \mid \mathbf{endconst}(*x) \\
d \text{ (proc. defs.)} & ::= & \{f \mapsto (x_1, \ldots, x_n)s\} \\
D \text{ (definitions)} & ::= & \langle d_1 \cup \cdots \cup d_n \rangle \\
P \text{ (programs)} & ::= & \langle D, s \rangle \\
E \text{ (context)} & ::= & E; s \mid \mathbf{const}(*x)E \mid []
\end{array}
$$

Notation $\vec{x}$ is for a finite sequence $\{x_1, ..., x_n\}$, where we assume that each element is distinct; $[\vec{x'}/\vec{x}]s$ is for a term obtained by replacing each free occurrence of $\vec{x}$ in s with variables $\vec{x'}$; the $\mathbf{Dom}(f)$ is a mapping from function name f to its domain; for a map f, the $f\{x \mapsto v\}$ and $f \backslash x$ are defined as follows:

$$
\begin{array}{rcl}
f\{x \mapsto v\}(w) & = & \begin{cases} v & \text{if } x = w \\ f(w) & \text{otherwise.} \end{cases} \\
(f \backslash x)(w) & = & \begin{cases} \text{undefined} & \text{if } x = w \\ f(w) & \text{otherwise.} \end{cases}
\end{array}
$$

and $filter\_C(C, *x)$ is defined by a pseudcode as follows:

$$
\begin{array}{rcl}
filter\_C(C, *x) & = & let\ C' = C - \mathbf{const}(*x)\ in \\
 & & if\ \mathbf{const}(*x) \in C'\ then\ return\ C' \\
 & & else\ return\ C' \backslash \{\mathbf{assume}(*x = \mathbf{null}), \mathbf{assume}(*x \neq \mathbf{null})\}
\end{array}
$$

The **Var** is a countably infinite set of *variables* and each variable is a pointer. The statement **skip** means "does nothing". The statement $s_1; s_2$ is a sequential execution of s_1 and s_2. The statement $*x \leftarrow y$ updates the content of cell which is pointed to by x with the value y. The statement $\mathbf{free}(x)$ deallocates a memory cell which is pointed to by pointer x. The statement $\mathbf{let}\ x = e\ \mathbf{in}\ s$ evaluates the expression e, binds x to the result, and executes s. The expression $\mathbf{malloc}()$ allocates a new memory cell. The expression **null** evaluates to the null pointer. The expression $*y$ means dereferencing a memory cell pointed to by y. The statement $\mathbf{ifnull}\ (*x)\mathbf{then}\ s_1\mathbf{else}\ s_2$ executes s_1 if $*x$ is **null** and executes s_2 otherwise. The statement $f(\vec{x})$ expresses a procedure f with arguments $\vec{x}$. The statement $\mathbf{const}(*x)s$ means $(*x)$ is a constant in statement s; the statement $\mathbf{endconst}(*x)$ means from this point $(*x)$ maybe not constant.

The d represents a procedure definition which maps a procedure name f to its procedure body $(\vec{x})s$; The D represents a set of procedure definitions $\langle d_1 \cup \ldots d_n \rangle$, and each definition is distinct;

The pair $\langle D, s\rangle$ represents a program, where D is a set of definitions and s is a main statement; the E represents evaluation context.

1.1 Operational semantics

In this section we introduce operational semantics of language $\mathcal{L}$. We assume there is a countable infinite set $\mathcal{H}$ of *heap addresses* ranged over by l.

We use a quadruple configuration $\langle H, R, s, n\rangle$ to express a run-time state. Each elements in the configuration is as follows.

- H, a *heap*, is a finite mapping from $\mathcal{H}$ to $\mathcal{H} \cup \{\mathbf{null}\}$;
- R, an *environment*, is a finite mapping from **Var** to $\mathcal{H} \cup \{\mathbf{null}\}$;
- s is the statement that is being executed;
- n is a natural number that represents the number of memory cells available for allocation.
- C is a set of actions, which contains $\mathbf{const}(*x)$, $\mathbf{assume}(*x = \mathbf{null})$ and $\mathbf{assume}(*x \neq \mathbf{null})$.

The operational semantics of the language $\mathcal{L}$ is given by a labeled transition relation $\langle H, R, s, n, C\rangle \xrightarrow{\rho}_D \langle H', R', s', n', C'\rangle$. The label ρ is as follows.

$$\rho \text{ (label)} \quad ::= \quad \mathbf{malloc} \mid \mathbf{free} \mid \tau$$

The ρ, an *action*, is **malloc**, **free**, $\mathbf{assume}(*x = \mathbf{null})$, $\mathbf{assume}(*x \neq \mathbf{null})$, $\mathbf{startconst}(*x)$, $\mathbf{endconst}(*x)$ or τ. The action **malloc** expresses an allocation of a memory cell; **free** expresses a deallocation of a memory cell; $\mathbf{assume}(*x = \mathbf{null})$ and $\mathbf{assume}(*x \neq \mathbf{null})$ express the guard part of conditional are $*x = \mathbf{null}$ and $*x \neq \mathbf{null}$ respectively; $\mathbf{startconst}(*x)$ means $*x$ should be constant from this point; $\mathbf{endconst}(*x)$ means the $*x$ no longer be a constant from this point; τ expresses the other actions. We often omit τ in $\xrightarrow{\tau}_D$. We use a metavariable σ for a finite sequence of actions $\rho_1 \dots \rho_n$. We write $\xrightarrow{\rho_1 \cdots \rho_n}_D$ for $\xrightarrow{\rho_1}_D \xrightarrow{\rho_2}_D \dots \xrightarrow{\rho_n}_D$. We write $\xRightarrow{\rho}_D$ for $\longrightarrow^*_D \xrightarrow{\rho}_D \longrightarrow^*_D$. We write $\xRightarrow{\rho_1 \cdots \rho_n}_D$ for $\xRightarrow{\rho_1}_D \dots \xRightarrow{\rho_n}_D$.

Figure 1 depicts the relation $\xrightarrow{\rho}_D$. Several important rules are listed as follows.

- Sem-ConstSkip: That a memory cell pointed to by $x$is no longer a constant is expressed by doing nothing.
- Sem-ConstSeq: That a memory cell pointed to by x should be a constant in a stamtement s is expressed by adding a statement $\mathbf{endconst}(*x)$ at the end of statement s.
- Sem-Free: Deallocation of a memory cell pointed to by x is expressed by deleting the entry for $R(x)$ from the heap. This action increments the number of available cells (i.e., n) by one (i.e., $n + 1$).
- Sem-Malloc and Sem-OutOfMem: Allocation of a memory cell is expressed by adding a fresh entry to the heap. This action is allowed only if the number of available cells is positive; if the number is zero, then the configuration leads to an error state **OutOfMemory**.

- SEM-ASSIGNEXN,SEM-FREEEXN,SEM-DEREFEXN and SEM-FREEEXN : These rules express an illegal access to memory. If such action is performed, then the configuration leads to exceptional state **MemEx**. This state **MemEx** is not seen as an erroneous state in the current paper, hence a well-typed program may lead to these states. The command **free**(x) , if x is a null pointer, leads to **MemEx** in the current semantics, although it is equivalent to **skip** in the C language.
- SEM-CONSTEXN: expresses that if a constant $*x$ is changed in s it will raise **ConstEx** exception.

Our goal is to guarantee *total* memory-leak freedom and reject memory leaks. By our language $\mathcal{L}$, they are formally defined as follows:

Definition 1 (total memory-leak freedom)**.** *A program $\langle D, s\rangle$ is* totally memory-leak free *if there is a natural number n such that it does not require more than n cells.*

Definition 2 (Memory leak)**.** *A configuration $\langle H, R, s, n, C\rangle$* goes overflow *if there is σ such that $\langle H, R, s, n, C\rangle \overset{\sigma}{\Longrightarrow}$* **OutOfMemory**. *A program $\langle D, s\rangle$* consumes at least n cells *if $\langle \emptyset, \emptyset, s, n, \epsilon\rangle$ goes overflow.*

2 Type system

2.1 Types

The syntax of the types is as follows.

P	(behavioral types)	$::=$	$\mathbf{0} \mid P_1; P_2 \mid \mathbf{malloc} \mid \mathbf{free} \mid \alpha \mid \mu\alpha.P$
			$\mid (*x)(P_1, P_2) \mid P_1 + P_2 \mid \mathbf{const}(*x)P \mid \mathbf{endconst}(*x)$
Γ	(variable type environment)	$::=$	$\{x_1, x_2, \ldots, x_n\}$
Ψ	(dependent function type)	$::=$	$(\vec{x})P$
Θ	(function type environment)	$::=$	$\{f_1 : \Psi_1, \ldots, f_n : \Psi_n\}$
k	(constant values)	$::=$	$\mathbf{null}(*x) \mid \neg\mathbf{null}(*x) \mid \mathbf{const}(*x)$
F	(constant value environment)	$::=$	$\{k_1, ..., k_n\}$

Behavioral types ranged over by P express the abstaction of behaviors of a program. The type $\mathbf{0}$ represents the do-nothing behavior; the type $P_1; P_2$ represents the sequential execution of P_1 and P_2; The type **malloc** represents an allocation of a memory cell exactly once; the type **free** represents a deallocation; the type $\mu\alpha.P$ represents the behavior of α defined by the recursive equation $\alpha = P$; the type $(*x)(P_1, P_2)$ represents that P_1 or P_2 is obtained dependent on $*x$; the type $P_1 + P_2$ represents the choice between P_1 and P_2; the α is a type variable; the type $\mathbf{const}(*x)P$ represents that $*x$ is a constant in behavioral type P ; the type $\mathbf{endconst}(*x)$ represents $*x$ no longer be a constant from this point.

A type environments for variables ranged over by Γ is a set of variables. Since our interest is the behavior of a program, not the types of values, a variable type environment does not carry information on the types of variables.

Dependent function types ranged over by Ψ represents the behavior of a function; $\vec{x}$ is the formal arguments of the function.

$$\frac{C' = filter\_C(C, *x)}{\langle H, R, \mathbf{endconst}(*x), n, C\rangle \rightarrow_D \langle H, R, \mathbf{skip}, n, C'\rangle} \quad (\textsc{Sem-ConstSkip})$$

$$\langle H, R, \mathbf{const}(*x)s, n, C\rangle \rightarrow_D \langle H, R, s; \mathbf{endconst}(*x), n, C \cup \{\mathbf{const}(*x)\}\rangle \quad (\textsc{Sem-ConstSeq})$$

$$\langle H, R, \mathbf{skip}; s, n, C\rangle \longrightarrow_D \langle H, R, s, n, C\rangle \quad (\textsc{Sem-Skip})$$

$$\frac{\langle H, R, s_1, n, C\rangle \xrightarrow{\rho}_D \langle H', R', s_1', n', C'\rangle}{\langle H, R, s_1; s_2, n, C\rangle \xrightarrow{\rho}_D \langle H', R', s_1'; s_2, n', C'\rangle} \quad (\textsc{Sem-Seq})$$

$$\frac{x' \notin \mathbf{Dom}(R)}{\langle H,\ R,\ \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s, n, C\rangle \longrightarrow_D \langle H,\ R\left\{x' \mapsto \mathbf{null}\right\},\ [x'/x]\, s, n, C\rangle} \quad (\textsc{Sem-LetNull})$$

$$\frac{x' \notin \mathbf{Dom}(R)}{\langle H,\ R,\ \mathbf{let}\ x = y\ \mathbf{in}\ s, n, C\rangle \longrightarrow_D \langle H,\ R\left\{x' \mapsto R(y)\right\},\ [x'/x]\, s, n, C\rangle} \quad (\textsc{Sem-LetEq})$$

$$\frac{H(R(x)) = \mathbf{null}, \mathbf{const}(*x) \notin C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2,\ n, C\rangle \rightarrow_D \langle H,\ R,\ s_1,\ n, C\rangle} \quad (\textsc{Sem-IfNullT})$$

$$\frac{H(R(x)) \neq \mathbf{null}, \mathbf{const}(*x) \notin C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2,\ n, C\rangle \rightarrow_D \langle H,\ R,\ s_2,\ n, C\rangle} \quad (\textsc{Sem-IfNullF})$$

$$\frac{H(R(x)) = \mathbf{null}, \mathbf{const}(*x) \in C \text{ and } \mathbf{assume}(*x \neq \mathbf{null}) \notin C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2,\ n, C\rangle \rightarrow_D \langle H,\ R,\ s_1,\ n, C \cup \{\mathbf{assume}(*x = \mathbf{null})\}\rangle} \quad (\textsc{Sem-IfConstNullT})$$

$$\frac{H(R(x)) \neq \mathbf{null}, \mathbf{const}(*x) \in C \text{ and } \mathbf{assume}(*x = \mathbf{null}) \notin C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2,\ n, C\rangle \rightarrow_D \langle H,\ R,\ s_2,\ n, C \cup \{\mathbf{assume}(*x \neq \mathbf{null})\}\rangle} \quad (\textsc{Sem-IfConstNullF})$$

$$\frac{\mathbf{const}(*x) \notin C}{\langle H\{R(x) \mapsto v\}, R, *x \leftarrow y, n, C\rangle \longrightarrow_D \langle H\left\{R(x) \mapsto R(y)\right\}, R, \mathbf{skip}, n, C\rangle} \quad (\textsc{Sem-Assign})$$

$$\frac{x' \notin \mathbf{Dom}(R) \qquad R(y) \in \mathbf{Dom}(H)}{\langle H,\ R,\ \mathbf{let}\ x = *y\ \mathbf{in}\ s, n, C\rangle \longrightarrow_D \langle H,\ R\left\{x' \mapsto H(R(y))\right\},\ [x'/x]\, s, n, C\rangle} \quad (\textsc{Sem-LetDeref})$$

$$\frac{R(x) \neq \mathbf{null} \text{ and } R(x) \in \mathbf{Dom}(H)}{\langle H\{R(x) \mapsto v\},\ R,\ \mathbf{free}(x), n, C\rangle \xrightarrow{\mathbf{free}}_D \langle H \backslash R(x),\ R,\ \mathbf{skip}, n+1, C\rangle} \quad (\textsc{Sem-Free})$$

$$\frac{l \notin \mathbf{Dom}(H) \qquad n > 0}{\langle H,\ R,\ \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, n, C\rangle \xrightarrow{\mathbf{malloc}}_D \langle H\left\{l \mapsto v\right\},\ R\left\{x' \mapsto l\right\},\ [x'/x]\, s, n-1, C\rangle} \quad (\textsc{Sem-Malloc})$$

$$\frac{D(f) = (\vec{y})s}{\langle H,\ R,\ f(\vec{x}), n, C\rangle \longrightarrow_D \langle H,\ R,\ [\vec{x}/\vec{y}]\, s, n, C\rangle} \quad (\textsc{Sem-Call})$$

$$\frac{R(x) = \mathbf{null} \text{ or } R(x) \notin \mathbf{Dom}(H)}{\langle H,\ R,\ \mathbf{free}(x), n, C\rangle \xrightarrow{\mathbf{free}}_D \mathbf{MemEx}} \quad (\textsc{Sem-FreeExn})$$

$$\frac{R(x) = \mathbf{null} \text{ or } R(x) \notin \mathbf{Dom}(H)}{\langle H,\ R,\ *x \leftarrow y, n, C\rangle \longrightarrow_D \mathbf{MemEx}} \quad (\textsc{Sem-AssignExn})$$

$$\frac{R(y) = \mathbf{null} \text{ or } R(y) \notin \mathbf{Dom}(H)}{\langle H,\ R,\ \mathbf{let}\ x = *y\ \mathbf{in}\ s, n, C\rangle \longrightarrow_D \mathbf{MemEx}} \quad (\textsc{Sem-DerefExn})$$

$$\frac{\mathbf{const}(*x) \in C}{\langle H\{R(x) \mapsto v\}, R, *x \leftarrow y, n, C\rangle \longrightarrow_D \mathbf{ConstEx}} \quad (\textsc{Sem-AssignConstExn})$$

$$\frac{H(R(x)) \neq H'(R'(x)) \qquad \langle H, R, s, n, C\rangle \xrightarrow{\rho}_D \langle H', R', s', n', C\rangle}{\langle H, R, \mathbf{const}(*x)s, n, C\rangle \xrightarrow{\rho}_D \mathbf{ConstEx}} \quad (\textsc{Sem-ConstExn})$$

$$\langle H,\ R,\ \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, 0, C\rangle \xrightarrow{\mathbf{malloc}}_D \mathbf{OutOfMemory} \quad (\textsc{Sem-OutOfMem})$$

Figure 1: Operational semantics of $\mathcal{L}$.

Function types ranged over by Θ is a mapping from function names to dependent function types.

k represents constant values, where $\mathbf{null}(*x)$ represents $(*x)$ is a null pointer; $\neg\mathbf{null}(*x)$ represents $(*x)$ is not a null pointer; $\mathbf{const}(*x)$ represents $(*x)$ is a constant.

Constant value environment ranged over by F is a set of constant variables.

Figure 2 depicts semantics of behavioral types with dependent types, and they are given by the labeled transition system. The relation $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$ means that P can make an action ρ, and P turns into P' after it makes action ρ; F and F' record constant value environment before and after action ρ respectively.

Notation $filter\_T(F, *x)$ is defined by a pseudcode as follows:

$$
\begin{array}{rcl}
filter\_T(F, *x) & = & let\ F' = F - \mathbf{const}(*x)\ in \\
& & if\ \mathbf{const}(*x) \notin\ F'\ then\ return\ (F' \backslash \{\mathbf{null}(*x), \neg\mathbf{null}(*x)\}) \\
& & else\ return\ F'
\end{array}
$$

2.2 Typing rules

The type judgment for statements is of the form $\Theta; \Gamma \vdash s : P$, which represents that under the function type environment Θ and the variable type environment Γ, the abstracted behavioral type of statement s is P.

Before showing typing rules for statements in Figure 3, we need explain several important definitions. The first one is $OK_n(P, F)$, a predicate, where P represents the behavior of a program which consumes at most n memory cells.

Definition 3 ($\sharp_\rho(\sigma)$). *$\sharp_\rho(\sigma)$ is the number of ρ in the sequence σ.*

Definition 4. *$OK_n(P, F)$ holds if, (1) $\forall P'$ and σ. if $\langle P, F\rangle \xrightarrow{\sigma} \langle P', F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \le n$ and (2) $OK(F)$*

Definition 5. *$OK(F)$ holds if F does not contain both $\mathbf{null}(*x)$ and $\neg\mathbf{null}(*x)$.*

Definition 6 (Subtyping). *$P_1 \le P_2$ is the largest relation such that, for any P_1', F' and ρ, if $\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F'\rangle$, then there exists P_2' such that $\langle P_2, F\rangle \stackrel{\rho}{\Longrightarrow} \langle P_2', F'\rangle$ and $P_1' \le P_2'$.*

2.3 Type soundness

Theorem 2.1. *If $\vdash \langle D, s\rangle : n$ for some n, then $\langle D, s\rangle$ is totally memory-leak free.*

The proof is based on the following lemmas: preservation and lack of immediate overflow.

Definition 7. *we write $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$, if $\Theta; \Gamma \vdash s : P$ and $OK_n(P, F)$ with $C \approx F$.*

Lemma 2.2 (Preservation). *suppose that $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$. If $\langle H, R, s, n, C\rangle \xrightarrow{\rho} \langle H', R', s', n', C'\rangle$ then $\exists P', F'$ s.t. (1) $\Theta; \Gamma \vdash \langle H', R', s', n', C'\rangle : \langle P', F'\rangle$ and (2) $\langle P, F\rangle \stackrel{\rho}{\Longrightarrow} \langle P', F'\rangle$.*

Lemma 2.3 (Lack of immediate overflow). *If $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$, then $\langle H, R, s, n, C\rangle \stackrel{\mathbf{malloc}}{\not\longrightarrow}$ $\mathbf{OutOfMemory}$.*

$$\langle \mathbf{0}; P, F\rangle \to \langle P, F\rangle \quad (\textsc{Tr-Skip}) \qquad \langle \mathbf{malloc}, F\rangle \xrightarrow{\mathbf{malloc}} \langle \mathbf{0}, F\rangle \ (\textsc{Tr-Malloc})$$

$$\langle \mathbf{free}, F\rangle \xrightarrow{\mathbf{free}} \langle \mathbf{0}, F\rangle \quad (\textsc{Tr-Free}) \qquad \langle \mu\alpha.P, F\rangle \to \langle [\mu\alpha.P/\alpha]P, F\rangle \ (\textsc{Tr-Rec})$$

$$\langle P_1 + P_2, F\rangle \to \langle P_1, F\rangle \ (\textsc{Tr-ChoiceL}) \qquad \langle P_1 + P_2, F\rangle \to \langle P_2, F\rangle \ (\textsc{Tr-ChoiceR})$$

$$\frac{\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F'\rangle}{\langle P_1; P_2, F\rangle \xrightarrow{\rho} \langle P_1'; P_2, F'\rangle} \qquad (\textsc{Tr-Seq})$$

$$\langle \mathbf{const}(*x)P, F\rangle \to \langle P; \mathbf{endconst}(*x), F \cup \{\mathbf{const}(*x)\}\rangle \qquad (\textsc{Tr-Const})$$

$$\frac{F' = \mathit{filter\_T}(F, *x)}{\langle \mathbf{endconst}(*x), F\rangle \to \langle \mathbf{0}, F'\rangle} \qquad (\textsc{Tr-Endconst})$$

$$\frac{\neg\mathbf{null}(*x) \notin F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_1, F \cup \{\mathbf{null}(*x)\}\rangle} \qquad (\textsc{Tr-NNullNotIn})$$

$$\frac{\mathbf{const}(*x) \notin F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_1, F\rangle} \ (\textsc{Tr-NotConst1}) \qquad \frac{\mathbf{null}(*x) \in F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_1, F\rangle} \ (\textsc{Tr-NullIn})$$

$$\frac{\mathbf{null}(*x) \notin F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_2, F \cup \{\neg\mathbf{null}(*x)\}\rangle} \qquad (\textsc{Tr-NullNotIn})$$

$$\frac{\neg\mathbf{null}(*x) \in F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_2, F\rangle} \qquad (\textsc{Tr-NNullIn})$$

$$\frac{\mathbf{const}(*x) \notin F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_2, F\rangle} \qquad (\textsc{Tr-NotConst2})$$

Figure 2: semantics of behavioral types with dependent types.

$$\Theta; \Gamma \vdash \mathbf{skip} : \mathbf{0} \quad \text{(T-Skip)}$$

$$\frac{\Theta; \Gamma \vdash s_1 : P_1 \quad \Theta; \Gamma \vdash s_2 : P_2}{\Theta; \Gamma \vdash s_1; s_2 : P_1; P_2} \quad \text{(T-Seq)}$$

$$\Theta; \Gamma, x, y \vdash *x \leftarrow y : \mathbf{0} \quad \text{(T-Assign)}$$

$$\Theta; \Gamma, x \vdash \mathbf{free}(x) : \mathbf{free} \quad \text{(T-Free)}$$

$$\frac{\Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \ \mathbf{in}\ s : \mathbf{malloc}; P} \quad \text{(T-Malloc)}$$

$$\frac{\Theta; \Gamma, x, y \vdash s : P}{\Theta; \Gamma, y \vdash \mathbf{let}\ x = y\ \mathbf{in}\ s : P} \quad \text{(T-LetEq)}$$

$$\frac{\Theta; \Gamma, x, y \vdash s : P}{\Theta; \Gamma, y \vdash \mathbf{let}\ x = *y\ \mathbf{in}\ s : P} \quad \text{(T-LetDeref)}$$

$$\frac{\Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s : P} \quad \text{(T-LetNull)}$$

$$\Theta; \Gamma, x \vdash \mathbf{endconst}(*x) : \mathbf{endconst}(*x) \quad \text{(T-Endconst)}$$

$$\frac{\Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma, x \vdash \mathbf{const}(*x)s : \mathbf{const}(*x)P} \quad \text{(T-Const)}$$

$$\frac{\Theta; \Gamma, x \vdash s_1 : P_1 \qquad \Theta; \Gamma, x \vdash s_2 : P_2}{\Theta; \Gamma, x \vdash \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 : (*x)(P_1, P_2)} \quad \text{(T-IfNull)}$$

$$\Theta, f : (\vec{y})P; \Gamma, \vec{x} \vdash f(\vec{x}) : P[\vec{x}/\vec{y}] \quad \text{(T-Call)}$$

$$\frac{\Theta; \Gamma \vdash s : P_1 \qquad P_1 \leq P_2}{\Theta; \Gamma \vdash s : P_2} \quad \text{(T-Sub)}$$

$$\frac{\Theta(f) = (\vec{x})P \qquad \mathbf{Dom}(D) = \mathbf{Dom}(\Theta) \qquad \Theta; x_1, \ldots, x_n \vdash s : P \text{ for each } f \mapsto (x_1, \ldots, x_n)s \in D}{\vdash D : \Theta} \quad \text{(T-Def)}$$

$$\frac{\vdash D : \Theta \qquad \Theta; \emptyset \vdash s : P \quad OK_n(P, F)}{\vdash \langle D, s \rangle : n} \quad \text{(T-Program)}$$

Figure 3: typing rules