

Cb AST Validation

1 Naming Rules

- No duplicate names amongst globals, externs, and functions.
- No duplicate names amongst struct types, and for a given struct type no duplicate names amongst its fields.
- For a given function, no duplicate names amongst its parameters and no duplicate names amongst its locals.

2 Miscellaneous Rules

- There exists a function called `main` and its type is $() \rightarrow \text{int}$.
- For every function, all paths from that function's entry reach a `Return` statement.

3 Type Checking

3.1 Types

$\tau \in \text{Type} ::= \text{int} \mid \text{struct}_{Id} \mid \&\tau \mid (\vec{\tau}) \rightarrow \tau^?$

A type τ is either the integer type, a struct type (qualified by the name of the struct), a pointer type (qualified by the type being pointed to), or a function type (qualified by the types of the parameters and an optional return type). Examples:

- An integer value has type `int`.
- A struct named `foo` has type `structfoo`.
- A pointer to an int has type `&int`.
- A function taking an int and a pointer to an int and returning an int has type $(\text{int}, \&\text{int}) \rightarrow \text{int}$.
- A function taking an int and a pointer to an int and not returning anything has type $(\text{int}, \&\text{int}) \rightarrow \dots$

3.2 Type Environment

Let $\Gamma : Id \rightarrow \text{Type}$ be a map from identifiers (variable and function names) to types.

Let $\Delta : Id \rightarrow (Id \rightarrow \text{Type})$ be a map from identifiers (struct names) to a map from identifiers (field names) to types.

We initialize Γ_0 and Δ from the the AST `Program` node's `globals`, `structs`, `externs`, and `functions` fields. Note that for a function inside `externs` the function name is mapped to its function type, while for a function inside `functions` the function name is mapped to a *pointer* to the function type. Also, we don't include the `main` function in Γ_0 . At this point, Γ_0 has all the type information for global variables, extern declared functions, and internal defined functions (except `main`) and Δ has the typing information for all struct declarations.

Δ will stay constant throughout the type checking process. Γ_0 will be augmented when type checking each function's body to contain that function's parameters and locals.

3.3 Exp and Lval Typing Rules

Typing judgements for **Exp** and **Lval** nodes in the AST are of the form $\Gamma, \Delta \vdash e : \tau$, stating that given the type environment consisting of Γ and Δ , the AST node has type τ .

$$\frac{}{\Gamma, \Delta \vdash \text{Num}(n) : \text{int}} \text{ NUM}$$

$$\frac{\Gamma(\text{name}) = \tau}{\Gamma, \Delta \vdash \text{Id}(\text{name}) : \tau} \text{ ID}$$

$$\frac{}{\Gamma, \Delta \vdash \text{Nil} : \&_} \text{ NIL}$$

We're using a convenient ad-hoc notation for the NIL rule expressing that a **Nil** expression is a pointer, but it can be treated as a pointer to any type. When comparing two types to see whether they are equal, **Nil**'s type should be considered equal to any other pointer type.

$$\frac{\Gamma, \Delta \vdash e : \text{int}}{\Gamma, \Delta \vdash \text{Unop}(\text{Neg}, e) : \text{int}} \text{ NEG}$$

$$\frac{\Gamma, \Delta \vdash e : \&\tau}{\Gamma, \Delta \vdash \text{Unop}(\text{Deref}, e) : \tau} \text{ Deref}$$

$$\frac{\text{op} \in \{\text{Equal}, \text{NotEq}\} \quad \Gamma, \Delta \vdash \text{left} : \tau \quad \Gamma, \Delta \vdash \text{right} : \tau \quad \tau \in \{\text{int}, \&\tau'\}}{\Gamma, \Delta \vdash \text{Binop}(\text{op}, \text{left}, \text{right}) : \text{int}} \text{ BINOP-EQ}$$

$$\frac{\text{op} \notin \{\text{Equal}, \text{NotEq}\} \quad \Gamma, \Delta \vdash \text{left} : \text{int} \quad \Gamma, \Delta \vdash \text{right} : \text{int}}{\Gamma, \Delta \vdash \text{Binop}(\text{op}, \text{left}, \text{right}) : \text{int}} \text{ BINOP-REST}$$

$$\frac{\Gamma, \Delta \vdash \text{ptr} : \&\tau \quad \Gamma, \Delta \vdash \text{idx} : \text{int}}{\Gamma, \Delta \vdash \text{ArrayAccess}(\text{ptr}, \text{idx}) : \tau} \text{ ARRAY}$$

$$\frac{\Gamma, \Delta \vdash \text{ptr} : \&\text{struct}_{id} \quad \Delta(id)(\text{fld}) = \tau}{\Gamma, \Delta \vdash \text{FieldAccess}(\text{ptr}, \text{fld}) : \tau} \text{ FIELD}$$

$$\frac{\text{callee} \neq \text{main} \quad \Gamma, \Delta \vdash \text{callee} : \&(\vec{\tau}) \rightarrow \tau_r \quad \forall (e, \tau) \in (\text{args}, \vec{\tau}). [\Gamma, \Delta \vdash e : \tau]}{\Gamma, \Delta \vdash \text{Call}(\text{callee}, \text{args}) : \tau_r} \text{ ECALL-INTERNAL}$$

$$\frac{\Gamma, \Delta \vdash \text{callee} : (\vec{\tau}) \rightarrow \tau_r \quad \forall(\mathbf{e}, \tau) \in (\mathbf{args}, \vec{\tau}) . [\Gamma, \Delta \vdash \mathbf{e} : \tau]}{\Gamma, \Delta \vdash \text{Call}(\text{callee}, \mathbf{args}) : \tau_r} \text{ECALL-EXTERN}$$

We need two different call rules because internally defined functions are present in Γ as pointer types (in order to allow for function pointers and indirect calls) while externally defined functions are present in Γ as function types (because we can't call them indirectly via pointers).

3.4 Statement Typing Rules

Typing judgements for statement nodes in the AST are of the form $\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{stmt} : \mathbf{ok}$, where $\tau_r^?$ is the (optional) return type of the function containing stmt and \mathbf{loop} is a boolean indicating whether stmt is contained inside a while loop.

$$\frac{\mathbf{loop} = \mathbf{true}}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Break} : \mathbf{ok}} \text{BREAK}$$

$$\frac{\mathbf{loop} = \mathbf{true}}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Continue} : \mathbf{ok}} \text{CONTINUE}$$

$$\frac{\tau_r^? = \mathbf{none}}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Return}(\mathbf{none}) : \mathbf{ok}} \text{RETURN-1}$$

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \tau \quad \tau_r^? = \tau}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Return}(\mathbf{e}) : \mathbf{ok}} \text{RETURN-2}$$

$$\frac{\Gamma, \Delta \vdash \mathbf{lhs} : \tau \quad \Gamma, \Delta \vdash \mathbf{e} : \tau \quad \tau \notin \{\text{struct}_{id}, (\vec{\tau}) \rightarrow \tau_r\}}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Assign}(\mathbf{lhs}, \mathbf{e}) : \mathbf{ok}} \text{ASSIGN-EXP}$$

$$\frac{\Gamma, \Delta \vdash \mathbf{lhs} : \&\tau \quad \Gamma, \Delta \vdash \mathbf{e} : \text{int} \quad \tau \neq (\vec{\tau}) \rightarrow \tau_r}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Assign}(\mathbf{lhs}, \text{New}(\tau, \mathbf{e})) : \mathbf{ok}} \text{ASSIGN-NEW}$$

$$\frac{\text{callee} \neq \mathbf{main} \quad \Gamma, \Delta \vdash \text{callee} : \&(\vec{\tau}) \rightarrow \tau_1^? \quad \forall(\mathbf{e}, \tau) \in (\mathbf{args}, \vec{\tau}) . [\Gamma, \Delta \vdash \mathbf{e} : \tau]}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Call}(\text{callee}, \mathbf{args}) : \mathbf{ok}} \text{SCALL-INTERNAL}$$

$$\frac{\Gamma, \Delta \vdash \text{callee} : (\vec{\tau}) \rightarrow \tau_1^? \quad \forall(\mathbf{e}, \tau) \in (\mathbf{args}, \vec{\tau}) . [\Gamma, \Delta \vdash \mathbf{e} : \tau]}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{Call}(\text{callee}, \mathbf{args}) : \mathbf{ok}} \text{SCALL-EXTERN}$$

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \text{int} \quad \forall s \in \mathbf{tt} . [\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash s : \mathbf{ok}] \quad \forall s \in \mathbf{ff} . [\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash s : \mathbf{ok}]}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \text{If}(\mathbf{e}, \mathbf{tt}, \mathbf{ff}) : \mathbf{ok}} \text{IF}$$

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \text{int} \quad \forall s \in \mathbf{body}. [\Gamma, \Delta, \tau_r^?, \mathbf{true} \vdash s : \mathbf{ok}]}{\Gamma, \Delta, \tau_r^?, \mathbf{loop} \vdash \mathbf{While}(\mathbf{e}, \mathbf{body}) : \mathbf{ok}} \text{ WHILE}$$

3.5 Program Typing Rules

Typing judgements for the **Program** node and its children in the AST are of the form $\Gamma, \Delta \vdash \mathbf{node} : \mathbf{ok}$.

$$\frac{\forall g \in \mathbf{globs}. [\Gamma_0, \Delta \vdash g : \mathbf{ok}] \quad \forall s \in \mathbf{structs}. [\Gamma_0, \Delta \vdash s : \mathbf{ok}] \quad \forall f \in \mathbf{funcs}. [\Gamma_0, \Delta \vdash f : \mathbf{ok}]}{\Gamma_0, \Delta \vdash \mathbf{Program}(\mathbf{globs}, \mathbf{structs}, \mathbf{externs}, \mathbf{funcs}) : \mathbf{ok}} \text{ PROGRAM}$$

The **PROGRAM** rule is the entry point for the type checker; it kicks everything off by calling all the other rules directly or transitively. Note that it explicitly uses Γ_0 , the initial type environment, whereas all other rules use whatever type environment is passed to them.

$$\frac{\tau \notin \{\mathbf{struct}_{id}, (\vec{\tau}) \rightarrow \tau_r\}}{\Gamma, \Delta \vdash \mathbf{Decl}(\mathbf{name}, \tau) : \mathbf{ok}} \text{ GLOBAL}$$

$$\frac{\forall \mathbf{Decl}(\mathbf{name}, \tau) \in \mathbf{flds}. [\tau \notin \{\mathbf{struct}_{id}, (\vec{\tau}) \rightarrow \tau_r\}]}{\Gamma, \Delta \vdash \mathbf{Struct}(\mathbf{name}, \mathbf{flds}) : \mathbf{ok}} \text{ STRUCT}$$

$$\frac{\begin{array}{l} \Gamma' = \mathbf{prms} \sqcup \mathbf{locals.decls} \quad \forall \mathbf{Decl}(\mathbf{name}, \tau) \in \Gamma'. [\tau \notin \{\mathbf{struct}_{id}, (\vec{\tau}) \rightarrow \tau_r\}] \quad \Gamma'' = \Gamma \sqcup \Gamma' \\ \forall (\mathbf{Decl}(\mathbf{name}, \tau), \mathbf{e}) \in \mathbf{locals}. [\Gamma'', \Delta \vdash \mathbf{e} : \tau] \quad \forall s \in \mathbf{stmts}. [\Gamma'', \Delta, \tau_r^?, \mathbf{false} \vdash s : \mathbf{ok}] \end{array}}{\Gamma, \Delta \vdash \mathbf{Function}(\mathbf{name}, \mathbf{prms}, \tau_r^?, \mathbf{locals}, \mathbf{stmts}) : \mathbf{ok}} \text{ FUNCTION}$$

The notation $\Gamma' = \mathbf{prms} \sqcup \mathbf{locals.decls}$ means to turn the parameter and local declarations (ignoring any optional local initializations) into a type environment, i.e., a map from names to types, allowing any entries from the right-hand side to override entries with the same name on the left-hand side. In other words, a local declaration overrides a parameter declaration with the same name. Similarly for $\Gamma'' = \Gamma \sqcup \Gamma'$ any entry from Γ' overrides an entry with the same name in Γ .