Static Semantics

- structural constraints not captured by BNF or abstract grammar
- resolvable at "semantic analysis time" (after symbol-table built)

ex: `Command ::= while Expression do Command`

- `Expression` must be boolean valued
- can be expressed by CFG, but introducing more non-terminals complicates grammar

ex: `Dec_list ::= Dec ; Dec_list | Dec`
`Dec ::=var Identifier:Type-denoter`

- constraint: no variable name declared twice
- cannot be expressed by a CFG (BNF)

ex: `Command ::= Identifier := Expression`

- constraint: id type = *var-t* where *t* = expr type
- in theory can be enforced by a huge CFG

Static Semantics: Typing Functions

- Types are a kind of coarse "value"
- Type checker: semantic maps that send `Expression`s, etc. to `Type` values
- `Type` values not "run-time" or "dynamic" values.
- `Type` a very simple semantic domain

$$\texttt{Type} = bool-type + int-type$$
$$+ var-bool-type + var-int-type + err-type$$

Type Environment

- "Static Environment" (implementable by symbol table)
- Type-Environ = `Identifier` → Type
  - — *typenv* : Type-Environ a type assignment that maps identifiers to types
  - — *typenv* usually produced by scanning a declaration list
  - — analogous to Environ (run-time or "dynamic" environments)
- Auxiliary functions for Type-Environ

  empty – environ : Type-Environ
  bind : `Identifier` × Type → Type-Environ
  overlay : Type-Environ × Type-Environ → Type-Environ
  find : Type-Environ × `Identifier` → Type

- Static semantic function signatures:

  typify : `Expression` → (Type-Environ → Type)
  constrain : `Command` → (Type-Environ → Boolean)
  declare : `Declaration` →
   (Type-Environ → Boolean × Type-Environ)

Expression Typing

- $typify$: Expression $\rightarrow$ Type-Environ $\rightarrow$ Type
  - — Expressions have a *type* value in a given type environment
  - — type environments produced by declaration

$$typify[\![N]\!]\ typenv\ = \quad int-type$$

$$typify[\![\mathbf{true}]\!]\ typenv\ = \quad bool-type$$

$$typify[\![\mathbf{false}]\!]\ typenv\ = \quad bool-type$$

$$typify[\![I]\!]\ typenv\ = \quad coerce-type(\ find(typenv, I)\ )$$

auxiliary function:

$coerce-type$: Type $\rightarrow$ Type

$$coerce-type(bool-type)\ =\ bool-type$$
$$coerce-type(int-type)\ =\ int-type$$
$$coerce-type(var-bool-type)\ =\ bool-type$$
$$coerce-type(var-int-type)\ =\ int-type$$

— $coerce-type$ does "type dereferencing"

Expression Typing (cont’d)

```
$typify[\![E_1 \texttt{+} E_2]\!]\ typenv\ =$
    if $int-type = typify\ [\![E_1]\!]\ typenv$
        $\wedge\ int-type = typify\ [\![E_2]\!]\ typenv$
     then $int-type$
     else $err-type$

$typify[\![E_1 \texttt{<} E_2]\!]\ typenv\ =$
    if $int-type = typify\ [\![E_1]\!]\ typenv$
        $\wedge\ int-type = typify\ [\![E_2]\!]\ typenv$
     then $bool-type$
     else $err-type$

$typify[\![\texttt{not}\ E]\!]\ typenv\ =$
    if $bool-type = typify\ [\![E]\!]\ typenv$
     then $bool-type$
     else $err-type$

 . . .
```

Command Typing

- $\mathit{constrain} : \texttt{Command} \rightarrow (\text{Type-Environ} \rightarrow \text{Boolean})$
- expressions are *constrained* as consistent or not in a given type environment

```
$constrain[\![\texttt{skip}]\!]\ typenv =$
    $\mathbf{true}$

$constrain[\![I \texttt{:=} E]\!]\ typenv =$
    $\mathbf{let}\ typval = typify[\![E]\!]\ typenv\ \mathbf{in}$
    $\mathbf{let}\ vartypval = find(typenv, I)\ \mathbf{in}$
    $(vartypval = var(typval))$

$var : \text{Type} \rightarrow \text{Type}$
$var(int-type) = var-int-type$
$var(bool-type) = var-bool-type$

$constrain[\![\texttt{let}\ D\ \texttt{in}\ C]\!]\ typenv =$
    $\mathbf{let}\ (ok, typenv') = declare[\![D]\!]\ typenv\ \mathbf{in}$
    $\mathbf{if}\ ok$
     $\mathbf{then}\ constrain[\![C]\!]\ (overlay(typenv', typenv))$
     $\mathbf{else}\ \mathbf{false}$
```

`Command` Typing (cont’d)

$$
\begin{aligned}
&\mathit{constrain}[\![C_1 \;\texttt{;}\; C_2]\!]\ \mathit{typenv} = \\
&\qquad \mathit{constrain}[\![C_1]\!]\ \mathit{typenv} \wedge \mathit{constrain}[\![C_2]\!]\ \mathit{typenv}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{constrain}[\![\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ \mathit{typenv} = \\
&\qquad (\mathit{typify}[\![E]\!]\ \mathit{typenv} = \mathit{bool-type}) \\
&\qquad \wedge\ \mathit{constrain}[\![C_1]\!]\ \mathit{typenv} \\
&\qquad \wedge\ \mathit{constrain}[\![C_2]\!]\ \mathit{typenv}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{constrain}[\![\texttt{while}\ E\ \texttt{do}\ C]\!]\ \mathit{typenv} = \\
&\qquad (\mathit{typify}[\![E]\!]\ \mathit{typenv} = \mathit{bool-type}) \\
&\qquad \wedge\ (\mathit{constrain}[\![C]\!]\ \mathit{typenv})
\end{aligned}
$$

`Declaration` Typing

- $declare$: `Declaration` → (Type-Environ → Boolean × Type-Environ)
  - declarations are *declared* in a type environment
  - produce a new elementary *type environment*
  - T : `Type-denoter` has a meaning in Type

```
$declare[\![\texttt{const}\ I \sim E]\!]\ typenv =$
        $\textbf{let}\ typ = typify[\![E]\!]\ typenv\ \textbf{in}$
        $\textbf{if}\ (typ = err-type)$
         $\textbf{then}\ (\ \textbf{false}, empty-environ\ )$
         $\textbf{else}\ (\ \textbf{true}, bind(I, typ)\ )$
$declare[\![\texttt{var}\ I\ \texttt{:}\ T]\!]\ typenv =$
        $\textbf{let}\ typ = type-denoted-by\ [\![T]\!]\ \textbf{in}$
        $(\ \textbf{true}, bind(I, var(typ))\ )$
$type-denoted-by[\![\texttt{bool}]\!] = bool-type$
$type-denoted-by[\![\texttt{int}]\!] = int-type$
```