2.6 SkipOOMiniJOOL 语言的静态语义

本节形式地描述 SkipOOMiniJOOL 语言的静态语义，就是为它设计一个形式的类型系统，它由一组可以使用的类型、断言以及推理这些断言的规则组成，用来规定良型（well-typed）的 SkipOOMiniJOOL 程序应具有的性质。静态检查将根据这些推理规则在编译时检查程序的良型性，如果检查通过，则程序是良型的（这里定义的良型程序可能会有运行时的数组下标表达式越界错误）。这里的类型系统是静态检查的依据，先前的非形式介绍中，若有与此冲突的地方，以此为准。

2.6.1 抽象语法

为使所定义的 SkipOOMiniJOOL 语言静态语义简洁，这里采用 SkipOOMiniJOOL 抽象语法（abstract syntax），其文法如图 2-17 所示。为减少产生式和相应的定型规则，允许在产生式右部使用 $exp_1, exp_2, \ldots, exp_n$ 的形式，并且若 $n = 0$，则为空。类型检查将根据静态语义自上而下、自左至右地检查程序。

program = *vardeclist_fundeclist_fundeflist*

vardeclist_fundeclist_fundeflist = *fundeclist_fundeflist*
| *btype id* “**;**” *vardeclist_fundeclist_fundeflist*
| *btype id* “**=**” *initexp* “**;**” *vardeclist_fundeclist_fundeflist*

fundeclist_fundeflist = “void” “main” “(” “)” “;” *fundeflist*
| *rtype id* “(”$ftype_1$“,” $ftype_2$“,” …“,” $ftype_n$“)” “;” *fundeclist_fundeflist* ($n \geq 0$)

fundeflist = $fundef_1$ $fundef_2$ … $fundef_n$ ($n > 0$)

initexp = *exp*
| “{” exp_1“,” exp_2“,”… “,” exp_n “}” ($n > 0$)

fundef = *rtype id* “(” $ftype_1$ id_1“,” $ftype_2$ id_2“,”…“,” $ftype_n$ id_n “)” *block* ($n \geq 0$)

block = “{” *blkstmts* “}”

blkstmts = ε
| *stmt blkstmts*
| *btype id* “**;**” *blkstmts*
| *btype id* “**=**” *initexp* “**;**” *blkstmts*

stmt = “;”
| *lval assignop exp* “;”
| *id* “(” exp_1“,” exp_1“,”… exp_n “)” “;” ($n \geq 0$)
| “break” “;”
| “continue” “;”
| “return” *exp* “;”
| “return” “;”
| *block*
| “if” “(” *exp* “)” *stmt*
| “if” “(” *exp* “)” *stmt* “else” *stmt*
| “while” “(” *exp* “)” *stmt*

| “print” “(” *exp* “)” “;”
| “read” “(” *lval* “)” “;”

lval = *id* | *id* “[” *exp* “]” | “(” *lval* “)”

exp = *lval*
| “(” *exp* “)”
| *const*
| *uop exp*
| *exp biop exp*
| *lval assignop exp*
| *id* “(” exp_1“,” exp_1“,”… exp_n “)” ($n \geq 0$)

assignop = “=” | “*=” | “/=” | “%=” | “+=” | “−=”

biop = “*” | “/” | “%” | “+” | “−”
| “<” | “<=” | “>=” | “>” | “==” | “!=” | “&&” | “||”

uop = “+” | “−” | “!”

const = *intconst* | *strconst* | “true” | “false”

图 2-17 SkipOOMiniJOOL 的抽象文法

抽象语法与具体语法虽然不完全一样，但是基于图 2-17 中的抽象文法定义的静态语义反映 SkipOOMiniJOOL 语言的静态语义。具体语法到抽象语法的变换主要有以下几点：

1) 忽略一些由语法直接能检查的符号，如忽略如“class Program”的类声明，只包含类体；忽略“static”等；
2) 将每条声明有 *n*（*n*>0）个变量的声明语句按变量自左至右出现的次序拆成 *n* 条声明语句，使得每条声明语句仅声明一个变量。这样拆分的目的是便于定义含有或不含初始化表达式的变量声明语句的定型规则。
3) 将所有的全局变量声明按原先的先后次序调整到程序的头部，在全局变量声明之后、所有函数定义之前，增加所有函数的声明。这样做是因为 SkipOOMiniJOOL 语言允许全局变量和函数先使用再定义，而下文的静态语义是遵循变量或函数先定义再使用的原则来定义的。

   这个差异意味着静态检查的实现需要先遍历一次 AST，将所有全局变量和函数的类型信息收集到符号表中，然后再依据这里的类型系统进行静态检查。编译器使用的符号表就相当于类型系统中的定型环境。要想设计一个形式的类型系统来体现允许全局变量和函数的先使用再定义，将是一件非常复杂的事情，因此这里的抽象语言采用先定义再使用的方式。

   需要提醒的是，在实际实现编译器时，并不需要将 SkipOOMiniJOOL 语言的程序变换成抽象语法所描述的形式，而只需按上面所说的先做一遍信息收集工作。
4) 改变一些产生式的定义，使得便于写定型规则。
5) 引入较简洁的非终结符名，也有少数地方引入了较长的非终结符名。
6) 非终结符 *vtype*、*ftype* 和 *rtype* 的产生式没有给出，其原因在 2.6.3 节中说明。

2.6.2 抽象类型、定型环境、断言与定型规则

1、抽象类型

SkipOOMiniJOOL 语言的抽象类型的 EBNF 规则定义如下。

T^p = int | bool | String

T^b = T^p | $T^p[N]$ | $T^p[\ ]$

T^v = T^p | $T^p[N]$

T^f = T^p | $T^p[\ ]$

T^r = void | int | bool

T = unit | void | $T_1^b \times T_2^b \times ... \times T_n^b$ | $T_1^f \times T_2^f \times ... \times T_n^f \rightarrow T^r$

其中，T^b 为基本类型集合，T^v 为局部变量或全局变量所允许的类型集合，T^f 为形参所允许的类型集合，, $T^p[N]$表示长度为正整数 n（N 的值为 n）、元素类型为 T^p 的一维数组类型集合，$T^p[\ ]$表示长度不定、元素类型为 T^p 的一维数组类型集合。

注意：如果形参为数组类型的话，则它只能是 $T^p[\ ]$，即不指定长度的数组类型。

$T_1^f \times T_2^f \times ... \times T_n^f \rightarrow T^r$ 中的 $T_1^f \times T_2^f \times ... \times T_n^f$ 用来表示函数的形参类型。若 n 为 0 则代表空积类型，也就是 void 类型。$T_1^f \times T_2^f \times ... \times T_n^f \rightarrow T^r$ 中的 T^r 为函数的返回类型。$T_1^b \times T_2^b \times ... \times T_n^b$ 为函数调用的实参类型，n 为 0 时和上面的含义一样。unit 代表语句类型。T 表示 SkipOOMiniJOOL 语言中所有抽象类型的集合，因为，当 n 为 1 时，$T_1^b \times T_2^b \times ... \times T_n^b$ 就是 T^b，而 T^v、T^f 和 T^r 能推出的类型也都包含在其中了。

在下文中，符号τ^p、τ^b、τ^v、τ^f、τ^r和τ分别表示属于 T^p、T^b、T^v、T^f、T^r 和 T 的任意类型。

2、定型环境

类型检查在给定的定型环境中进行。定型环境Γ 定义如下：

Γ = $\varnothing_\Gamma$ | $\Gamma, id : \tau$ | $\Gamma, ret : \tau^r$

定型环境Γ包含当前作用域中的变量、函数、以及当前函数的返回类型 *ret*。*ret* 是一个和程序声明的任何变量都不同名的虚拟变量，以方便 return 语句的类型检查。如果 *ret* 的类型是 void，则函数无返回值。$\varnothing_\Gamma$表示空环境。Γ看成是变量和函数类型描述的一个集合，即可以按任意次序使用它。

3、断言

SkipOOMiniJOOL 语言的静态语义将由以下断言来描述。

$\Gamma \vdash \circ$	定型环境Γ是良形的
$\Gamma \vdash \tau$	在定型环境Γ下，类型τ是良形的
$\Gamma \vdash \tau_1 \cong \tau_2$	在定型环境Γ下，τ_1和τ_2是相容的类型
$\Gamma \vdash M : \tau$	在定型环境Γ下，M 的类型为τ，M 表示表达式、语句和程序等

一个定型环境Γ是良形的，是指Γ除了满足上面文法规定的语法外，Γ中还没有重名。该良形性的要求可以用推理规则表示，由于文法已经给出，其它约束又很简单，因此Γ良形性的规则在此略去。

在上述断言中，形如$\Gamma \vdash M : \tau$的断言是**定型断言**，它表示在定型环境Γ下，M 具有类型τ。下面的推理规则重点用来描述什么样的良形程序是良型的。

4、推理规则

推理规则是在一组已知的、有效的断言基础上，声称某个断言的有效性。在本书中采取如下形式来描述：

(规则名)　　推理规则

推理规则的一般形式是

$$\frac{\Gamma_1 \vdash S_1, \ldots, \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

在每条推理规则中，横线上面是一组称为**前提**的断言，横线下面是一个称为**结论**的断言。断言除了上面给出的几种形式外，还可以是整数的关系断言和有关集合的断言。推理规则的含义是，当所有的前提都有效时，则结论也有效。前提个数可以为零，这样的规则称为**公理**，即横线下的断言是有效的。

为了保证语言是安全的，SkipOOMiniJOOL 语言将副条件（side condition）引入到传统的定型规则中。这些副条件用来说明在变量值或引用状态（主要指数组类型的形参对实参数组的引用）上的显式需求。为了与一般的定型前提相区分，副条件包含在一对花括号中，写在相应的定型规则的右边。下面的规则中，除个别情况外，一般的副条件在静态检查中难以完成，因为它可能涉及程序运行时才能确定的性质。

2.6.3 良形的类型以及类型兼容

1、良形的类型

由于 SkipOOMiniJOOL 语言很简单，类型表达式中不会出现类型变量和程序定义的类型名字等。因此类型τ只要符合前面有关类型的文法，则它就是良形的。类型表达式的良形性规范也可以用推理规则表示，由于它等价于前面有关类型的文法，因此免去用推理规则再描述一遍。

由于略去了有关类型表达式良形性的推理规则，在下面的推理规则中，直接用τ^p、τ^b、τ^v、τ^f、τ^r和τ来表示各种类型表达式的实例。这也是为了集中精力关注定型规则的设计和使用。正因为这样，2.6.1 节抽象文法没有给出非终结符 *vtype*、*ftype* 和 *rtype* 的产生式，τ^b、τ^f和τ^r分别表示它们的实例。

2、类型兼容

当不同类型的变量用于赋值和参数传递时，需要比较它们的类型兼容性。对任意两个类型τ_1、τ_2来说，如果它们是兼容的，则记为$\tau_1 \cong \tau_2$。以下描述了 SkipOOMiniJOOL 语言的类型兼容推理规则。其中在 TCMP_ARRAY1 规则中的 *N.value* 表示 *N* 所表示的整数值。

(TCMP_INT)　　$\dfrac{}{\Gamma \vdash \text{int} \cong \text{int}}$

(TCMP_BOOL)　　$\dfrac{}{\Gamma \vdash \text{bool} \cong \text{bool}}$

(TCMP_STRING)　　$\dfrac{}{\Gamma \vdash \text{string} \cong \text{string}}$

(TCMP_VOID)　　$\dfrac{}{\Gamma \vdash \text{void} \cong \text{void}}$

(TCMP_ARRAY1) $$\frac{\Gamma \vdash \tau^p[N_1] \quad \Gamma \vdash \tau^p[N_2] \quad N_1.value = N_2.value}{\Gamma \vdash \tau^p[N_1] \cong \tau^p[N_2]}$$

(TCMP_ARRAY2) $$\frac{\Gamma \vdash \tau_1^p[] \quad \Gamma \vdash \tau_2^p[N] \quad \Gamma \vdash \tau_1^p \cong \tau_2^p}{\Gamma \vdash \tau_1^p[] \cong \tau_2^p[N]}$$

(TCMP_ARRAY3) $$\frac{}{\Gamma \vdash \tau^p[] \cong \tau^p[]}$$

(TCMP_PRODUCT) $$\frac{\Gamma \vdash \tau_1 \cong \tau_1' \quad \Gamma \vdash \tau_2 \cong \tau_2' \quad \ldots \quad \Gamma \vdash \tau_n \cong \tau_n'}{\Gamma \vdash \tau_1 \times \tau_2 \times \ldots \times \tau_n \cong \tau_1' \times \tau_2' \times \ldots \times \tau_n'}$$

规则(TCMP_PRODUCT)描述了函数的实参列表与形参列表之间的类型兼容规则。

2.6.4 表达式

表达式定型规则的前提是某产生式右部各子表达式的定型，而结论是该产生式整个右部的定型。下面分类给出定型规则。

1、常量

常量包括布尔值 true 和 false、整数以及字符串。下面的规则是显然的。

(EXP_TRUE) $$\frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

(EXP_FALSE) $$\frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

(EXP_INT) $$\frac{}{\Gamma \vdash intconst : \text{int}}$$

(EXP_STRING) $$\frac{}{\Gamma \vdash strconst : \text{String}}$$

推理规则中出现非终结符 *intconst*、*strconst* 表明它是一条规则模板，例如(EXP_INT)规则表示任何 *intconst* 的实例都是 int 类型。

产生式 *exp* = *const* 右部只有一个非终结符，*const* 是什么类型则 *exp* 也是什么类型，对这样的情况，通常无需定型规则。

2、带括号的表达式

(EXP_BRACKET) $$\frac{\Gamma \vdash exp : \tau}{\Gamma \vdash (exp) : \tau}$$

3、左值

以下四条是左值表达式的定型规则，其中中间两条对应非终结符 *lval* 的同一个产生式。第 3 条规则中的 *id* 在声明时没有指定长度（即充当形参），也就是没有 *length* 属性，靠获得实参数组的长度来作为它运行时的一个属性。

(LVAL_VAR) $$\frac{}{\Gamma, id : \tau^p \vdash id : \tau^p}$$

(LVAL_ARRAY1) $$\frac{\Gamma \vdash id : \tau^p[N] \quad \Gamma \vdash exp : \text{int}}{\Gamma \vdash id[exp] : \tau^p} \{0 \le exp.value < N.value\}$$

(LVAL_ARRAY2) $$\frac{\Gamma \vdash id : \tau^p[] \quad \Gamma \vdash exp : \text{int}}{\Gamma \vdash id[exp] : \tau^p}\{0 \le exp.value < id.length\}$$

(LVAL _ BRACKET) $$\frac{\Gamma \vdash lval : \tau^p}{\Gamma \vdash (lval) : \tau^p}$$

在这些规则的断言$\Gamma \vdash M : \tau$中，环境Γ和 M 中都出现文法的终结符 *id*，不同 *id* 的区分是靠它们的属性（名字）。

4、双目运算

SkipOOMiniJOOL 语言的双目运算分以下三大类：在 int 型值上的算术运算，其表达式的定型规则由(EXP_INT_ARITH)给出；在 int 型值上的比较运算，其定型规则由(EXP_INT_COMP)给出；在 bool 值上的运算，其定型规则由(EXP_BOOL_ARITH)给出。

(EXP_INT_ARITH) $$\frac{\Gamma \vdash exp_1 : \text{int} \quad \Gamma \vdash exp_2 : \text{int}}{\Gamma \vdash exp_1\ biop\ exp_2 : \text{int}}$$

其中 *biop* = “+” | “ –” | “*” | “/” | “%”

(EXP_INT_COMP) $$\frac{\Gamma \vdash exp_1 : \text{int} \quad \Gamma \vdash exp_2 : \text{int}}{\Gamma \vdash exp_1\ biop\ exp_2 : \text{bool}}$$

其中 *biop* = “<” | “<=” | “>=” | “>” | “==” | “!=”

(EXP_BOOL_ARITH) $$\frac{\Gamma \vdash exp_1 : \text{bool} \quad \Gamma \vdash exp_2 : \text{bool}}{\Gamma \vdash exp_1 biop\ exp_2 : \text{bool}}$$

其中 *biop* = “&&” | “||” | “==” | “!=”

5、单目运算

SkipOOMiniJOOL 语言的单目运算分两大类：在 int 型值上的一元运算，由规则(EXP_INT_UARITH)给出；在 bool 值上的一元运算，由规则(EXP_BOOL_NOT)给出。

(EXP_INT_UARITH) $$\frac{\Gamma \vdash exp : \text{int}}{\Gamma \vdash uop\ exp : \text{int}}$$

其中 *uop* = “+” | “–”

(EXP_BOOL_NOT) $$\frac{\Gamma \vdash exp : \text{bool}}{\Gamma \vdash !exp : \text{bool}}$$

6、方法调用

规则(EXPLIST_ CALL)描述了函数调用的定型规则。实参的类型可以为$\tau^p[]$，这时实参是当前函数的形参。

(EXP_FUN_CALL) $$\frac{\Gamma \vdash exp_1 : \tau_1^b \qquad \Gamma \vdash exp_2 : \tau_2^b \qquad \ldots \qquad \Gamma \vdash exp_n : \tau_n^b \qquad \Gamma \vdash id : \tau_1^f \times \tau_2^f \times \ldots \times \tau_n^f \to \tau^r \quad \Gamma \vdash \tau_1^f \times \tau_2^f \times \ldots \times \tau_n^f \cong \tau_1^b \times \tau_2^b \times \ldots \times \tau_n^b}{\Gamma \vdash id(exp_1, exp_2, \ldots, exp_n) : \tau^r}$$

7、赋值

赋值表达式的定型规则按表达式的类型区分为如下两条。它们规定赋值运算符左边的表达式必须是左值，而且左右两边的表达式必须有相同的类型。

(EXP_INT_ASSIGN) $$\frac{\Gamma \vdash lval : \text{int} \quad \Gamma \vdash exp : \text{int}}{\Gamma \vdash lval\ assignop\ exp : \text{int}}$$

其中 *assignop* = “=” | “+=” | “–=” | “*=” | “/=” | “%=”

(EXP_OTHER_ASSIGN) $$\frac{\Gamma \vdash lval : \text{bool} \quad \Gamma \vdash exp : \text{bool}}{\Gamma \vdash lval = exp : \text{bool}}$$

$$\frac{\Gamma \vdash lval : \text{String} \quad \Gamma \vdash exp : \text{String}}{\Gamma \vdash lval = exp : \text{String}}$$

2.6.5 语句和语句块

语句和语句块通常没有类型，为便于表达语句和语句块中没有类型错误，故给它以 unit 类型。下面分别给出各种语句的定型规则。

1、表达式语句

表达式语句中的表达式只允许是赋值表达式或函数调用。它们的定型规则如下：

(STMT_ASSIGN) $$\frac{\Gamma \vdash lval\ assignop\ exp : \tau^{p}}{\Gamma \vdash lval\ assignop\ exp; : \text{unit}}$$

(STMT_FUN_CALL) $$\frac{\Gamma \vdash id(exp_1, exp_2, ...exp_n) : \tau^{r}}{\Gamma \vdash id(exp_1, exp_2, ...exp_n); : \text{unit}}$$

注意：按前面的约定，(STMT_FUN_CALL)规则包括了无参函数调用的情况。

2、分支和循环语句

(STMT_IF) $$\frac{\Gamma \vdash exp : \text{bool} \quad \Gamma \vdash stmt : \text{unit}}{\Gamma \vdash \text{if}(exp)\ stmt : \text{unit}}$$

(STMT_IF_ELSE) $$\frac{\Gamma \vdash exp : \text{bool} \quad \Gamma \vdash stmt_1 : \text{unit} \quad \Gamma \vdash stmt_2 : \text{unit}}{\Gamma \vdash \text{if}(exp)\ stmt_1\ \text{else}\ stmt_2 : \text{unit}}$$

(STMT_WHILE) $$\frac{\Gamma \vdash exp : \text{bool} \quad \Gamma \vdash stmt : \text{unit}}{\Gamma \vdash \text{while}\ (exp)\ stmt : \text{unit}}$$

3、return 语句

return 语句的定型规则主要是保证 return 后的表达式 *exp* 的类型与当前所定义的函数的返回类型（保存在 *ret* 中）一样。而对于不带表达式的 return 语句，则要求 *ret* 的类型必须是 void。

(STMT_RETVAL) $$\frac{\Gamma \vdash exp : \tau^{r} \quad \Gamma \vdash ret : \tau^{r}}{\Gamma \vdash \text{return}\ exp; : \text{unit}}$$

(STMT_RET) $$\frac{\Gamma \vdash ret : \text{void}}{\Gamma \vdash \text{return}\ ; : \text{unit}}$$

4、其他简单语句

(STMT_EMPTY) $$\frac{}{\Gamma \vdash ; : \text{unit}}$$

(STMT_BREAK) $$\frac{}{\Gamma \vdash \text{break}\ ; : \text{unit}}$$

(STMT_CONT) $$\frac{}{\Gamma \vdash \text{continue}\ ; : \text{unit}}$$

(STMT_PRINT) $$\frac{\Gamma \vdash exp : \tau^p}{\Gamma \vdash \text{print } (exp);: \text{unit}}$$

(STMT_READ) $$\frac{\Gamma \vdash lval : \text{int}}{\Gamma \vdash \text{read } (lval);: \text{unit}}$$

5、复合语句块

语句块是一个由花括号包含的、0 个或多个局部变量声明或上述 *stmt* 语句组成的序列。局部变量声明的推理规则主要在于检查局部变量的类型是否声明成τ^p、$\tau^p[N]$或者$\tau^p[]$。如果局部变量在声明时指定了初始化表达式，还要检查局部变量的声明类型是否与初始化表达式的类型兼容。若局部变量声明为$\tau^p[]$类型，则必须带有数组初始化表达式，此时由初始化表达式中的表达式个数决定所声明的数组变量的长度。当类型检查通过时，将声明的局部变量及其类型添加到定型环境中。

在引入语句块的定型规则前，先考虑数组类型变量初始化的定型规则。各初值表达式的类型都一样。

(VEXP_INIT_ARRAY) $$\frac{\Gamma \vdash exp_1 : \tau^p \quad \Gamma \vdash exp_2 : \tau^p \quad \ldots \quad \Gamma \vdash \exp_n : \tau^p}{\Gamma \vdash \{ exp_1, exp_2, \ldots, exp_n \} : \tau^p[N]} \{N.value = n\}$$

以下是针对语句块花括号中的不同情况给出的定型规则。规则(BSTMTS_EMPTY)指出空语句序列是良形的。规则(BSTMTS_STMT_SEQ)定义以 *stmt* 开头的语句序列的良形性。值得注意的是规则(BSTMTS_VDEC_NOINIT_SEQ)和(BSTMTS_VDEC_INIT_SEQ)，它们逆向体现遇到变量声明时，将其添加到定型环境中，然后再基于添加后的定型环境检查剩余语句序列的类型。

(BSTMTS_EMPTY) $$\frac{}{\Gamma \vdash \varepsilon : \text{unit}}$$

(BSTMTS_VDEC_NOINIT_SEQ) $$\frac{\Gamma,\ id : \tau^v \vdash blkstmts : \text{unit} \quad id \notin dom(\Gamma)}{\Gamma \vdash \tau^v\ id; blkstmts : \text{unit}}$$

(BSTMTS_VDEC_INIT_SEQ) $$\frac{\Gamma, id : \tau^v \vdash blkstmts : \text{unit} \quad \Gamma \vdash initexp : \tau^v \quad \Gamma \vdash \tau^b \cong \tau^v \quad x \notin dom(\Gamma)}{\Gamma \vdash \tau^b\ id = initexp; blkstmts : \text{unit}}$$

(BSTMTS _STMT_SEQ) $$\frac{\Gamma \vdash stmt : \text{unit} \quad \Gamma \vdash blkstmts : \text{unit}}{\Gamma \vdash stmt\ blkstmts : \text{unit}}$$

由语句序列的良形性，就可以得到复合语句块的良形性。

(STMT_BLOCK) $$\frac{\Gamma \vdash blkstmts : \text{unit}}{\Gamma \vdash \{blkstmts\} : \text{unit}}$$

2.6.6 函数的定义

从定型规则的设计来说，由于在所有函数定义前有这些函数的声明，该规则的设计是简单的。它也需要像上一小节的(BSTMTS_VDEC_INIT_SEQ)规则那样逆向理解。注意，该规则包括了无参函数的情况。

(FDEF)
$$\frac{\Gamma \vdash id : \tau_1^f \times \tau_2^f \times \ldots \times \tau_n^f \to \tau^r \quad \Gamma,\ id_1 : \tau_1^f, id_2 : \tau_2^f, \cdots, id_n : \tau_n^f, ret : \tau^r \vdash \circ \quad \Gamma,\ id_1 : \tau_1^f, id_2 : \tau_2^f, \cdots, id_n : \tau_n^f, ret : \tau^r \vdash block : \text{unit}}{\Gamma \vdash \tau^r\ id(\tau_1^f\ id_1, \tau_2^f\ id_2, \cdots, \tau_n^f\ id_n)\ block : \text{unit}}$$

注意：参数类型是任意的τ^f，即参数类型不能是$\tau^p[N]$。

2.6.7 程序

程序中有 0 个或多个全局变量声明，后跟 1 个或多个函数声明，后跟相应这些函数声明的函数定义。有关它们的产生式是从右向左归约的，以便像(BSTMTS_VDEC_INIT_SEQ)规则和(FDEF)规则那样，体现遇到变量声明和函数声明就要将它们添加到定型环境中。

首先，所有的函数定义是在同样的定型环境下检查类型。

(PROG_FDEF)
$$\frac{\Gamma \vdash\ fundef_1 : \text{unit} \quad \Gamma \vdash\ fundef_2 : \text{unit}\ \ldots\ \Gamma \vdash\ fundef_n : \text{unit}}{\Gamma \vdash\ fundef_1\ fundef_2\ \ldots fundef_n : \text{unit}}$$

下面两条规则面向函数声明。用规则(PROG_FDEC1)来体现 main 函数一定存在，且类型是 void→void。规则(PROG_FDEC2)针对其它有参和无参的函数声明。

(PROG_FDEC1)
$$\frac{\Gamma, \text{main} : \text{void} \to \text{void} \vdash\ fundeflist : \text{unit} \quad \text{main} \notin dom(\Gamma)}{\Gamma \vdash \text{void main}();\ fundeflist : \text{unit}}$$

(PROG_FDEC2)
$$\frac{\Gamma, id : \tau_1^f \times \tau_2^f \times \ldots \times \tau_n^f \to \tau^r \vdash\ fundeclist\_fundeflist : \text{unit} \quad id \notin dom(\Gamma)}{\Gamma \vdash \tau^r\ id(\tau_1^f, \tau_2^f, \ldots \tau_n^f);\ \ fundeclist\_fundeflist : \text{unit}}$$

下面两条规则有关全局变量的声明，分别考虑无初值表达式和有初值表达式两种情况。

(PROG_VDEC1)
$$\frac{\Gamma, id : \tau^v \vdash vardeclist\_fundeclist\_fundeflist : \text{unit} \quad id \notin dom(\Gamma)}{\Gamma \vdash \tau^v\ id\ ; vardeclist\_fundeclist\_fundeflist : \text{unit}}$$

(PROG_VDEC2)
$$\frac{\Gamma, id : \tau^v \vdash vardeclist\_fundeclist\_fundeflist : \text{unit} \quad \Gamma \vdash initexp : \tau^v \quad \Gamma \vdash \tau^b \cong \tau^v \quad id \notin dom(\Gamma)}{\Gamma \vdash \tau^b\ id = initexp\ ; vardeclist\_fundeclist\_fundeflist : \text{unit}}$$

下面的规则(PROG_WT)用来强调整个程序在空环境下的良型性。

(PROG_WT)
$$\frac{\varnothing_\Gamma \vdash vardeclist\_fundeclist\_fundeflist : \text{unit}}{\varnothing_\Gamma \vdash\ program : \text{unit}}$$