National University of Singapore

CS4212 Project Assignment 2 – Part A

AY 2015/2016 Semester 1

Due Date: 18^{th} October 2015 (Sunday, 23:59 Hrs)

*** This is Part A of Assignment 2 ***

For Assignment 2, you are required to construct the second part of the front end of the **mOOL** compiler. Given a parse tree representing a **mOOL** program, you perform static checks over the parse tree, and then generate the corresponding intermediate code, defined by an intermediate representation language called *IR3*.

For Part A of this assignment, you will perform static checks over parse trees of **mOOL** programs, and create new parse trees annotated with type information. We call them *typed-parse trees.*

We will provide you with Part B of this assignment in the next few days. There, you will generate intermediate codes in *IR3*, for those **mOOL** programs which have successfully passed the static checks.

Please note that **the deadlines for submitting Part A and Part B of the Assignment 2 are the same.** Therefore, please start working on Part A.

1 Static Checker

For Part A, you need to construct a *static checker* that performs static checks on **mOOL** programs. The checker should include the following functionalities. Please indicate clearly to us if you have included new functionalities.

1. Class names: All class names must be unique.
2. Class hierarchy must be acyclic: No class can become a parent of itself through class inheritance declaration (by `extends` construct.)
3. Distinct same-scope identifiers:
   (a) No two attributes declared within a class can have the same name;
   (b) No two parameters formally declared within a method can have the same name;
   (c) No local variables formally declared within the same block in a method can have the same name.
4. Shadowing policy should be deployed: This ensures the following:
   (a) Given a method m declared in a class C, both the local variable declared in m *and* a parameter of m can *shadow* the fields in C (that is, having the same name as the fields.) E.g., the following two class declarations are allowed. Please check the comments written in the code for explanation.

```
class C1 {
    Int x ;
    Int y ;
    Int m1 (Int a, Int y) {
           // param y shadows attribute y of C1
          Int b ;
          b = a ;
          if (a == y)
              // y used here is the param y
             { return y + x ; }
               // x used here is the attribute x
          else { return  b ; }
    }
}
class C2 extends C1 {
    Int x ;
    Int y ;
    Int m1 (Int a, Int b) {
          Int y ;
          // local y shadows attribute y of C2
          y = a ;
          if (a == x)
             // x used here is the attribute x of C2
             { return x + y; }
               // y used here is the local y
          else {return b ; }
    }
}
```

5. Two methods declared within the same class (or in two inheritance-related classes[1]) cannot have the same name, except when they are **overloading** of each other: Two methods declared within the same class are overloading of each other if they have the **same** name but they have **different** signatures. A signature of a method is the combined type of its formal parameters. For instance, the signatures for the following two methods:

```
Int m1 (Int p1, C1 p2, Bool p3) { ... }
Int m1 (Int p1, C2 p2, Int p3) { ... }
```

   are: `(Int, C1, Bool)` and `(Int, C2, Int)` respectively. In addition, as they have the same name and they are declared in the same class, they are considered overloaded.

6. Two methods declared in two inheritance-related classes cannot have the same method name **and** the same signature, except when they both have the **same return type**. That is, method overriding is allowed: Two methods declared in two different classes related by direct or indirect inheritance (via the construct `extends`) are considered overriding if they

[1]Two classes are *inheritance-related* classes if they are related by direct or indirect inheritance.

have (1) the same method names, (2) the same signatures, and (3) the same return type.[2]

7. Private Modifier: A method/attribute declared in a class and with `private` modifier can only be accessed by objects belonging to that class.

8. Type checking: This makes sure that uses of program constructs conform to how the constructs have been declared. The type judgment for **mOOL** language is given in Appendix A, and was expounded in one of the lectures.

2 What've Been Given to You?

You are assumed to have the **mOOL** language specification at hand. If you don't, please retrieve it from Assignment 1 worksheet.

In addition, you are given, in Part A, the following information[3]:

1. A parsing-related files, `mOOL_lexer.mll` and `mOOL_parser.mly`. You need to use them to build a parser for **mOOL** programs. As you have already learned in Assignment 1, the result of parsing is a parse tree of type `mOOL_program`. **You must use the parser generated from these two files in this Assignment.** (These two files are given in the workbin.)

2. A module, `mOOL_structs.ml`, containing type definitions for typed-parse trees (capturing the result of static checks, given in the workbin.) Inside this module, there is also a display function for displaying **mOOL** code with annotated types.

3. Two sample **mOOL** source codes, called `ex1.mo` and `sample1.mo` respectively, and the results of running the static checker on them. The results are stored in the file `ex1.output` and `sample1.output` respectively. (More may be uploaded to the workbin.)

4. A makefile, `Makefile`, which allows you to create the executable for static checker. In Unix, running "make" on `Makefile` will produce an executable named `mOOL_checker`. **You must prepare a Makefile for generation of executable.** You will be severely penalized if your submission does not have a working and faithful `Makefile`.

5. A sample type-checker module, `mOOL_annotatedtyping.ml`, that helps you to jump start your development of the checker.

6. The main program, `mOOL_main.ml`, that links all pieces together to form the static checker.

Given a **mOOL** program as input, your static checker is required to generate the corresponding typed parse tree, which contains type annotation at each node of the parse tree.

Your static checker program will work directly on the parse tree (of `mOOL_program`) generated by the given **mOOL** parser. The result of static checking is the production of another parse tree, which is similar to the original parse tree, but with

[2]Can two methods declared within the same class be overridden by each other? The answer is "NO".

[3]Resources required for Part B will be given to you in due course.

type annotations included. Thus, both the input and output of static checker are parse trees of type `mOOL_program`. This is made possible because the type `mOOL_program` contains variant types, which allow co-existence of "raw program construct" types and "typed program construct" types. For instance, the type definition of **mOOL** identifiers is as follows:

```
type var_id =
   | SimpleVarId of string
   | TypedVarId of typed_var_id
```

The first type constructor `SimpleVarID` is used as the type of identifier in the original parse tree (produced by the parser), and the second type constructor `TypedVarID` is used as the type of the same identifier, but as produced by the static checker, and it contains type and scope information.

3 Deliverable

Testing of Your Programs

You are required to create some sample programs to test your product. Put in a subfolder those sample programs that you have created, and show their corresponding result.

Submission of your product

Instructions will be provided in Part B of Assignment 2. Please stay tune ...

4 Late Submission

We try to discourage you from submitting your assignment beyond deadline. This is to ensure that you have time to prepare for other modules, as well as time for other assignments handed out in this module.

Any submission after the deadline will have its mark automatically deducted by certain percentages. Your submission time is determined by the time recorded in IVLE submission folders. If you have multiple submissions, we will take the latest submission time. Any submission to places other than the appropriate submission folders (such as to the instructor's email account) will be discarded and ignored.

Here is the marking scheme:

Submit by 23:59HRS of	Maximum Mark	If your score is ... %	It becomes ... %
18th Oct (Sun)	100	80	80
19th Oct (Mon)	80	80	64
20th Oct (Tue)	60	80	48
After 20th Oct (Tue)	0	-	0

A Type Checking of mOOL Programs

A.1 Class Descriptor, Type Environment and Legitimate Types

The *type environment* comprises two entities: *class descriptor* $\mathcal{C}$ and *local environment* Γ.

1. *Class Descriptor* $\mathcal{C}$ maps a class name to a triple consisting of: the class name of its parent, the types of each of its field declarations, and the signatures of each of its method declarations. Note that, in case the class declared is **not** inherited from any other class, the type of its parent attribute will be $\top$.

$$\begin{array}{rcl} \mathcal{C} & :: & \langle cname \rangle \ \rightarrow \ (\langle cname \rangle \cup \{\top\}) \times \langle fds \rangle \times \langle msigs \rangle \\ \langle fds \rangle & = & \langle id \rangle \ \rightarrow \ T \\ \langle msigs \rangle & = & \langle id \rangle \ \rightarrow \ T^{+} \end{array}$$

2. *Local environment* Γ maps variables declared locally to their types.

$$\Gamma \quad :: \quad \langle id \rangle \ \rightarrow \ T$$

The type class T is declared as follows:

$$T \ = \ \langle cname \rangle \cup \{int, bool, void, \bot\}.$$

At the beginning of type checking of a **mOOL** program, we first fill the class descriptor with all type information declared in the program. This includes class inheritance information, the attribute declarations and method signatures occurred in each class. On the other hand, the initial local environment contains empty information (or it may contain some global type information, such as the types of the primitive operations).

In addition, we assume that all types *referred to* in the program must be ensured to be *legitimate*. That is, they must be found in the type class T.

A.2 Inheritance Relations

$$\frac{c \ \in \ T}{\mathcal{C} \ \vdash \ c \ \preceq \ c} \quad \text{[Inh-Ref]}$$

$$\frac{\langle c' \ , \ \_ \ , \ \_ \ \rangle \ = \ \mathcal{C}.c_1 \qquad c' \neq \top \qquad \mathcal{C} \ \vdash \ c' \ \preceq \ c_2}{\mathcal{C} \ \vdash \ c_1 \ \preceq \ c_2} \quad \text{[Inh-Tran]}$$

We also define a *least upper bound* operator, denoted by $\sqcup$, that determines a subtype which is a parent type (or the type itself) of two given types. We define this operator formally in two steps:

1. Given two types t_1 and t_2 both in T, a type t is said to be an *upper bound* of t_1 and t_2 if and only if $\mathcal{C} \ \vdash \ t_1 \preceq \ t$ and $\mathcal{C} \ \vdash \ t_2 \preceq \ t$.

2. Let U be the set of all upper bounds of t_1 and t_2, then

$$t_1 \sqcup t_2 = \begin{cases} t, & \text{if } t \in U, \quad \text{and} \quad \forall t' \in U, \mathcal{C} \vdash t \preceq t' \\ error, & \text{otherwise} \end{cases}$$

Finally, we abbreviate the inheritance relation by dropping the class descriptor $\mathcal{C}$, and writing simply $t_1 \preceq t_2$, when it is clear from the context.

A.3 Type Checking A mOOL Program

A **mOOL** program is well typed if the type checker returns $isOK$ upon return.

$$\frac{\begin{array}{c} P = mainC\ C_1 \cdots C_n \qquad \Gamma = [\,] \\ \mathcal{C} = initialize(P) \qquad \langle \mathcal{C}, \Gamma \rangle \vdash mainC\ isOK \\ \langle \mathcal{C}, \Gamma \rangle \vdash C_i\ isOK \quad \forall i \in \{1, \ldots, n\} \end{array}}{\vdash mainC\ C_1 \cdots C_n\ isOK} \quad [\textbf{mOOL}\text{-Pgm}]$$

A.4 Type Checking Classes

Before type checking all the methods declared in a class, the local environment is set to include respectively the types/signatures associated with the attributes/methods declared in the class, and the types for the special identifiers *this* and *super*.

$$\frac{\begin{array}{c} \langle c', (a_1 \mapsto t_1, \ldots, a_n \mapsto t_n), (ms_1, \ldots, ms_k) \rangle = \mathcal{C}.c \\ ms_i = (m_i \mapsto ((t_{i1}, \ldots, t_{ip}) \to tr_i)) \quad \forall i \in \{1, \ldots, k\} \\ \Gamma' = \Gamma[this \mapsto c, super \mapsto c', a_1 \mapsto t_1, \ldots, a_n \mapsto t_n, ms_1, \ldots, ms_k] \\ \langle \mathcal{C}, \Gamma' \rangle \vdash md_i\ isOK \quad \forall i \in \{1, \ldots, k\} \end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash \texttt{class}\ c\ \texttt{extends}\ c'\ \{t_1\ a_1; \ldots; t_n\ a_n;\ md_1; \ldots, md_k\ \}\ isOK} \quad [\text{SubCDecl}]$$

$$\frac{\begin{array}{c} \langle \top, (a_1 \mapsto t_1, \ldots, a_n \mapsto t_n), (ms_1, \ldots, ms_k) \rangle = \mathcal{C}.c \\ ms_i = (m_i \mapsto ((t_{i1}, \ldots, t_{ip}) \to tr_i)) \quad \forall i \in \{1, \ldots, k\} \\ \Gamma' = \Gamma[this \mapsto c, super \mapsto \top, a_1 \mapsto t_1, \ldots, a_n \mapsto t_n, ms_1, \ldots, ms_k] \\ \langle \mathcal{C}, \Gamma' \rangle \vdash md_i\ isOK \quad \forall i \in \{1, \ldots, k\} \end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash \texttt{class}\ c\ \{t_1\ a_1; \ldots; t_n\ a_n;\ md_1; \ldots, md_k\ \}\ isOK} \quad [\text{CDecl}]$$

A.5 Searching for a Method

When a method m from a class c is invoked with actual arguments-return types $(t'_1, \ldots, t'_n) \to t'_0$), we try to look for a matching method signature in the existing type environment, Γ. If this fails, we follow the ancestral path of the class hierarchy to locate the closest ancestor class having matching method signature. The search fails when no such matching method signature can be found in the path along the class hierarchy.

Method *searchSig* defined below performs the search recursively. It calls the auxiliary method *srchSig*1 to verify if the method signature found at current class level matches the actual call signature. This verification checks if the

actual argument types are subtypes of the declared method's formal parameter types.[4]

$$
\begin{array}{l}
searchSig(\langle \mathcal{C}, \Gamma\rangle, c, m, (t'_1, \ldots, t'_n)) \;=\\
\quad let \quad (b, (t_1, \ldots, t_n) \to t) \;=\; srchSig1(\langle \mathcal{C}, \Gamma\rangle, c, m, (t'_1, \ldots, t'_n))\\
\quad in \quad\; if \quad b \quad then\, (t_1, \ldots, t_n) \to t\\
\qquad\quad else\; let \quad \langle c', \_\,, \_\,\rangle = \mathcal{C}.c\\
\qquad\qquad\; in \quad if \quad (c' = \top) \quad then \quad error\\
\qquad\qquad\qquad else\; let \quad \langle\, \_\,, \_\,, \Gamma_{ms}\,\rangle = \mathcal{C}.c'\\
\qquad\qquad\qquad\quad\; in \quad searchSig(\langle \mathcal{C}, \Gamma_{ms}\rangle, c', m, (t'_1, \ldots, t'_n))
\end{array}
$$

$$
\begin{array}{l}
srchSig1(\langle \mathcal{C}, \Gamma\rangle, c, m, (t'_1, \ldots, t'_n)) \;=\\
\quad if\; ((t_1, \ldots, t_n) \to t) \;=\; \Gamma.m \quad and \quad \forall\, i \;\in\; \{1, \ldots, m\} \;:\; \mathcal{C} \vdash (t'_i \;\preceq\; t_i)\\
\quad then \quad (true,\; (t_1, \ldots, t_n) \to t)\\
\quad else \quad (false,\; (\bot, \ldots, \bot) \to \bot)
\end{array}
$$

A.6 Type Checking Method Declaration

When type checking a method declaration, the local environment is augmented with the types of the parameters declared in the method, **and the return type** of the method, associated with and unique special identifier *Ret*.

$$
\frac{\begin{array}{c}((t_1, \ldots, t_n) \to t_0) \in \Gamma.m\\ \langle \mathcal{C}, \Gamma[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n, Ret \mapsto t_0]\rangle \;\vdash_S\; S \;:\; t\\ \mathcal{C} \;\vdash\; t \;\preceq\; t_0\end{array}}{\langle \mathcal{C}, \Gamma\rangle \;\vdash\; t_0\; m(t_1\; v_1, \ldots, t_n\; v_n)\; S\; isOK} \quad \text{[MDecl]}
$$

A.7 Type Checking Statements

Type checking of statements is specified by the judgment: $\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; S \;:\; t$.

$$
\frac{\langle \mathcal{C}, \Gamma[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]\rangle \;\vdash_S\; S \;:\; t}{\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; \{\; t_1\; v_1; \ldots; t_n\; v_n;\; S\;\} \;:\; t} \quad \text{[Block]}
$$

$$
\frac{\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; S_1 \;:\; t_1 \qquad \langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; S_2 \;:\; t_2}{\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; S_1; S_2 \;:\; t_2} \quad \text{[Seq]}
$$

$$
\frac{\langle \mathcal{C}, \Gamma\rangle \;\vdash_E\; e \;:\; t_e \qquad \Gamma.v \;=\; t_v \qquad \mathcal{C} \;\vdash\; t_e \preceq t_v}{\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; v \texttt{ = } e \;:\; void} \quad \text{[VarAss]}
$$

$$
\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma\rangle \;\vdash_E\; e_1 \;:\; t_1 \qquad \langle \mathcal{C}, \Gamma\rangle \;\vdash_E\; e_2 \;:\; t_2\\ (a, t_a) \in attrib(\mathcal{C}.t_1) \qquad \mathcal{C} \;\vdash\; t_2 \preceq t_a\end{array}}{\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; e_1.a \texttt{ = } e_2 \;:\; void} \quad \text{[FdAss]}
$$

$$
\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; S_1 \;:\; t_1 \qquad \langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; S_2 \;:\; t_2\\ \langle \mathcal{C}, \Gamma\rangle \;\vdash_E\; e \;:\; bool \qquad t = t_1 \sqcup t_2\end{array}}{\langle \mathcal{C}, \Gamma\rangle \;\vdash_S\; \texttt{if}\; e\; \texttt{then}\; S_1\; \texttt{else}\; S_2 \;:\; t} \quad \text{[Cond]}
$$

[4] We have not included in our discussion the effect of modifier associated to a method: `public` and `private`. Such inclusion is not difficult, and will be left as exercise.

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ bool \qquad \langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ S \ : \ t}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ \texttt{while}\ e\ \{\ S\ \}\ :\ t} \quad \text{[While]}$$

$$\frac{dom(\Gamma)(v) \ = \ t \quad t \in \{int,\ bool,\ string\}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ \texttt{readln}\ v \ : \ void} \quad \text{[Read]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ t \quad t \in \{int,\ bool,\ string\}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ \texttt{println}\ e \ : \ void} \quad \text{[Print]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_i \ : \ t'_i \qquad \forall i \in \{1, \ldots, p\} \\ ((t_1, \ldots, t_p) \to t_r) \ = \ searchSig(\langle \mathcal{C}, \Gamma \rangle, \Gamma(this), m, (t'_1, \ldots, t'_p))\end{array}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ m(e_1, \ldots, e_p) \ : \ t_r} \quad \text{[SLocalCall]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_i \ : \ t'_i \qquad \forall i \in \{1, \ldots, p\} \\ \langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_0 \ : \ c_0 \quad \langle \_, \_, \Gamma_{MS} \rangle = \mathcal{C}.c_0 \\ ((t_1, \ldots, t_p) \to t_r) \ = \ searchSig(\langle \mathcal{C}, \Gamma_{MS} \rangle, c_0, m, (t'_1, \ldots, t'_p))\end{array}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ e_0.m(e_1, \ldots, e_p) \ : \ t_r} \quad \text{[SGlobalCall]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ t \qquad t' = \Gamma(Ret) \qquad \mathcal{C} \ \vdash \ t \preceq t'}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ \texttt{return}\ e \ : \ t'} \quad \text{[Ret-T]}$$

$$\frac{\Gamma.Ret = void}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ \texttt{return} \ : \ void} \quad \text{[Ret-Void]}$$

A.8 Type Checking Expressions

Type checking of expressions is specified by the judgment: $\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ t$.

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ t \qquad \mathcal{C} \ \vdash \ t \ \preceq \ t'}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : t'} \quad \text{[Sub]}$$

$$\frac{x \in \langle Id \rangle \cup \{\texttt{this},\ \texttt{super}\} \qquad \Gamma.x \ = \ t \qquad t \neq \top}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ x \ : t} \quad \text{[Id]}$$

$$\frac{}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ \texttt{null} \ : \bot} \quad \text{[Null]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ t \qquad \langle \_, fds, \_ \rangle = \mathcal{C}.t \qquad t_a \ = \ fds.a}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e.a \ : t_a} \quad \text{[Field]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e \ : \ t' \qquad t \in dom(\mathcal{C}) \\ \mathcal{C} \ \vdash t \ \preceq \ t' \text{ or } \mathcal{C} \ \vdash t' \ \preceq \ t\end{array}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ (t)e \ : \ t} \quad \text{[Cast]}$$

$$\frac{c \in dom(\mathcal{C})}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ \texttt{new}\ c() \ : \ c} \quad \text{[New]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_i \ : \ t'_i \qquad \forall i \in \{1, \ldots, p\} \qquad ((t_1, \ldots, t_p) \to t_r) \ = \ searchSig(\langle \mathcal{C}, \Gamma \rangle, \Gamma(this), m, (t'_1, \ldots, t'_p))}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ m\texttt{(}e_1, \ldots, e_p\texttt{)} \ : \ t_r} \quad \text{[ELocalCall]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_i \ : \ t'_i \qquad \forall i \in \{1, \ldots, p\} \qquad \langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_0 \ : \ c_0 \quad \langle \_, \_, \Gamma_{MS} \rangle = \mathcal{C}.c_0 \qquad ((t_1, \ldots, t_p) \to t_r) \ = \ searchSig(\langle \mathcal{C}, \Gamma_{MS} \rangle, c_0, m, (t'_1, \ldots, t'_p))}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_S \ e_0.m\texttt{(}e_1, \ldots, e_p\texttt{)} \ : \ t_r} \quad \text{[EGlobalCall]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_1 \ : \ int \qquad \langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_2 \ : \ int \qquad aop \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_1 \ aop \ e_2 \ : \ int} \quad \text{[Arith]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_1 \ : \ int \qquad \langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_2 \ : \ int \qquad rop \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_1 \ rop \ e_2 \ : \ bool} \quad \text{[Rel]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_1 \ : \ bool \qquad \langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_2 \ : \ bool \qquad bop \in \{\texttt{||}, \texttt{\&\&}\}}{\langle \mathcal{C}, \Gamma \rangle \ \vdash_E \ e_1 \ bop \ e_2 \ : \ bool} \quad \text{[Bool]}$$