Alpen-Adria-Universität Klagenfurt
Software Engineering Research Group
Technical Report Series

Method-Level Bug Prediction

Giovanni Liva, Muhammad Taimoor Khan

Report AAU-SERG-2019-001

AAU-SERG-2019-001

Published, produced and distributed by:

Software Engineering Research Group
Institute of Informatics Systems
Faculty of Technical Sciences
Alpen-Adria-Universität Klagenfurt
Universitätsstraße 65-67
9020 Klagenfurt
Austria

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://serg.aau.at/bin/view/Main/Publications

For more information about the Software Engineering Research Group:
http://serg.aau.at

Note: Appendix of TSE-2018-11-0423 submission

Proof of Soundness for Time Type System

Giovanni Liva[1] and Muhammad Taimoor Khan[2]

[1]Department of Software Engineering, Alpen-Adria Universität, Klagenfurt, Austria, giovanni.liva@aau.at
[2]School of Computing and Mathematical Sciences, University of Greenwich, London, UK, m.khan@gre.ac.uk

June 17, 2019

1 Introduction

In this article, we prove the soundness of the Time Type System (TTS) presented (and currently under review) in the TSE journal. We first recap syntax, semantics, and typing rules of TTS and then we prove its soundness. A type system is *sound* if well-typed programs do not incur run-time type errors, *i.e.*, they do not get stuck when evaluated according to the operational semantics:

$$\textit{Typing rules are sound} \implies \textit{no well-formed programs gets stuck.}$$

To be more precise, this means that when programs are well-typed they end up in a value or the computation is simply not finished and continues:

$$\vdash e : \tau \ \wedge \ e \longrightarrow^* e' \implies e' \in \textbf{Value} \ \vee \ \exists e''.\ e' \longrightarrow e''$$

Furthermore, we also prove the soundness of our repair strategy. A repair strategy is sound if the program before and after the repair is semantically equivalent. For this, we first presents the definition of semantically equivalent programs and then we prove the soundness of the repair strategy.

2 Syntax and Semantics

2.1 Syntax Rules

Figure 1 presents a subset of the generic programming language supported by TTS described using rules in Backus-Naur form. It presents the most interesting cases of the language with methods, statements, expressions, and boolean comparisons. Here, i represents an identifier for method names and *obj* for object names, n represents a numerical literal, and x ranges over program variables.

$$
\begin{array}{llll}
\textbf{Methods} & m & ::= & i(x_1, \_, x_n)\{\ s\ \} \\
\textbf{Statements} & s & ::= & e \mid x\ =\ e \mid \texttt{if}\ (b)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
 & & \mid & \texttt{while}\ (b)\ s \mid s_1; s_2 \mid \texttt{return}\ e \\
\textbf{Expressions} & e & ::= & n \mid x \mid e_1\ +\ e_2 \mid e_1\ -\ e_2 \mid e_1\ \times\ e_2 \\
 & & \mid & e_1\ \div\ e_2 \mid obj.m(e_1, \ldots, e_n) \\
 & & \mid & \frac{\texttt{min}}{\texttt{max}}(e_1, e_2) \\
\textbf{Booleans} & b & ::= & e_1\ <\ e_2 \mid e_1\ <=\ e_2 \mid e_1\ >=\ e_2 \\
 & & \mid & e_1\ >\ e_2 \mid b_1\ \texttt{\&\&}\ b_2 \mid b_1\ \texttt{||}\ b_2 \\
\textbf{Types} & \tau & ::= & T \mid D \mid DoT
\end{array}
$$

Figure 1: Programming language grammar supported by the Time Type System (TTS).

Methods m**.** We elide many details of the definition of a program and we represent it just as a list of methods. Each method m is composed of an identifier i, a list of variables $(x_1, \ldots, x_n)$ that represent its input parameters and a sequence of statements s.

Statements s**.** We include here the syntax for the assignment, if-else, while, sequence of statements and return statements. The other variants of conditional and loop statements typically provided in programming languages are handled in a similar way. The rule for the assignment statement assign the expression e to a variable x. The rules for the if-else and while statements contain a boolean expression b. Concerning the if-else statement, if b is true, the sequence of statements s_1 is executed otherwise s_2. Concerning the while statement, if b is true, the sequence of statements s is executed iteratively until b gets false. The rule for the return statement returns the expression e.

Expressions e**.** Regarding literals n, the rule considers only integer values. Furthermore, our syntax supports method calls in the form $obj.m(e_1, \ldots, e_n)$, the basic arithmetic operations, and min/max operations.

Booleans b**.** We support all the boolean operators for the comparison of timestamps. Note, that equality checks do not suffer from overflow comparison problems and therefore, we skip it. We also include the logical conjunction and disjunction of boolean expressions.

Types τ**.** For the sake of simplicity, we report only the time types. If we interpret the (real) time as a line, then Timestamp values T are used to represent arbitrary points along this line, while Duration values D can be used to represent an interval between two points. This differentiation is mandatory for our purposes since our repair strategy seeks for expressions that compare Timestamp values that can suffer from integer overflow problems. However, it is not possible to know a priori the exact time type of the time parameters/attributes because they do not always have an initialization expression that can be used to infer their time type. Therefore, we introduce the DoT time type

which expresses that a time expression is of type Duration or Timestamp.

2.2 Type Inference Rules

Our approach takes in input a well-typed Java program and outputs the program annotated with the time type information. The type inference rules describe how TTS assigns a time type to literals, variables, and expressions. The rules are expressed via operational semantics [1] and they consist of a set of premises and a conclusion. Both premises and the conclusion are judgments. A judgment has the form $e : \tau$ which means e has type τ. In the context of the paper, τ refers to a time type. Judgments include a type environment Γ that contains the set of type bindings from variables and expressions to their respective time types. Since assignments can change the binding of a variable to a new time type, we designed our type system to be *flow-sensitive* which is achieved by inserting an output environment Γ' in addition to the input environment. For the sake of readability, we have shortened the names of time types Timestamp and Duration with T and D, respectively.

Example. In the following we show an example of a typing rule consisting of two premises e_1 and e_2. The rule is read as: given that e_1 has type Duration and e_2 has type Timestamp in the type environment Γ, then the sum of the expressions e_1 and e_2 has type Timestamp. Therefore, variable x has type Timestamp in the output environment Γ'.

$$\frac{\Gamma \vdash e_1 : D \dashv \Gamma \quad \Gamma \vdash e_2 : T \dashv \Gamma}{\Gamma \vdash x = e_1 + e_2 : T \dashv \Gamma[x \mapsto T]}$$

The reasoning of the rule is based on the notion that if we add "some" time to a date, the result is still a date but in the future. Moreover, since the expression is assigned to variable x, the resulting environment extends Γ with a new mapping between x and its time type T. Figure 2-6 present the typing rules of TTS. Here, the function $isTimeVar$ returns *true* if the expression in input is a reference to a time variable, while the function *posType* that returns for the method m the set of positions for its arguments that are expected to be time related and their expected time types.

$$\frac{\Gamma_0 := \forall_{i=1}^{n} isTimeVar(x_i) \rightarrow \Gamma[x_i \mapsto DoT] \quad \Gamma_0 \vdash s \dashv \Gamma'}{\Gamma \vdash i(x_1, \_, x_n)\{\ s\ \} \dashv \Gamma'}\ [\textsc{Method}]$$

Figure 2: Methods.

$$\frac{\Gamma \vdash s_1 : \tau' \dashv \Gamma' \quad \Gamma' \vdash s_2 : \tau'' \dashv \Gamma''}{\Gamma \vdash s_1; s_2 \dashv \Gamma''}[\textsc{Stm}] \quad \frac{\Gamma \vdash e : \tau \dashv \Gamma'}{\Gamma \vdash \texttt{return}\ e : \tau \dashv \Gamma'}[\textsc{Ret}]$$

$$\frac{\Gamma \vdash b \dashv \Gamma' \quad \Gamma' \vdash s \dashv \Gamma''}{\Gamma \vdash \texttt{while}\ (b)\ s\ \dashv \Gamma''}[\textsc{While}] \quad \frac{\Gamma \vdash e : \tau \dashv \Gamma'[e \mapsto \tau]}{\Gamma \vdash x = e : \tau \dashv \Gamma'[x \mapsto \tau]}[\textsc{Assign}]$$

$$\frac{\Gamma \vdash b \dashv \Gamma' \quad \Gamma' \vdash s_1 \dashv \Gamma'' \quad \Gamma' \vdash s_2 \dashv \Gamma'''}{\Gamma \vdash \texttt{if}\ (b)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \dashv \Gamma'' \cup \Gamma'''}[\textsc{If}]$$

Figure 3: Statements.

$$\frac{\Gamma_0 := \Gamma \quad \forall_{i=1}^{n} \frac{\Gamma_{i-1} \vdash e_i : \tau \dashv \Gamma_i}{\Gamma_{i-1} \vdash e_i : \tau \dashv \Gamma_i[e_i \mapsto \tau]} \quad \Gamma'_0 := \Gamma_n \quad \forall_{(i,\tau) \in PosType(m)} \frac{\Gamma'_{i-1} \vdash e_i : \tau \dashv \Gamma'_i[e_i \mapsto \tau]}{}}{\Gamma \vdash obj.m(e_1, \ldots, e_n) \dashv \Gamma'_n}[\textsc{Et}]$$

$$\frac{m \in RT_\tau \quad \Gamma \vdash obj.m(e_1, \ldots, e_n) \dashv \Gamma'}{\Gamma \vdash obj.m(e_1, \ldots, e_n) : \tau \dashv \Gamma'}[\textsc{Rt}] \quad \frac{n \in \texttt{int} \vee n \in \texttt{long}}{\Gamma \vdash n : D \dashv \Gamma}[\textsc{Literal}]$$

$$\frac{\Gamma \vdash e_1 : \tau \dashv \Gamma' \quad \Gamma' \vdash e_2 : \tau \dashv \Gamma''}{\Gamma \vdash \frac{\texttt{min}}{\texttt{max}}(e_1, e_2) : \tau \dashv \Gamma''}[\textsc{Min-Max}] \quad \frac{x \in \Gamma \quad \tau := \Gamma(x)}{\Gamma \vdash x : \tau \dashv \Gamma}[\textsc{Var}]$$

$$\frac{\begin{array}{ll}\Gamma \vdash & e_1 : \tau' \quad \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau'' \quad \dashv \Gamma''\end{array} \quad \begin{cases} ① \vee ② & : \odot = - \\ ② \vee ③ \vee ④ & : \odot = + \\ ④ & : \odot = \times \\ ④ & : \odot = \div \end{cases}}{\Gamma \vdash e_1 \odot e_2 : \tau_0 \dashv \Gamma''}[\textsc{Int}]$$

$$① : \tau' = \tau'' \rightarrow \tau_0 := D \qquad ② : \tau' = T \wedge \tau'' = D \rightarrow \tau_0 := T$$
$$③ : \tau' = D \wedge \tau'' = T \rightarrow \tau_0 := T \qquad ④ : \tau' = D \wedge \tau'' = D \rightarrow \tau_0 := D$$

Figure 4: Expressions.

$$\frac{\Gamma_0 = \Gamma \quad \forall_{i=1}^{n} \Gamma_{i-1} \vdash b_i \dashv \Gamma_i \quad \odot \in \{\texttt{||}, \texttt{\&\&}\}}{\Gamma \vdash b_1 \odot \cdots \odot b_n \dashv \Gamma_n}[\textsc{Bool}]$$

$$\frac{\begin{array}{ll}\Gamma \vdash & e_1 : \tau \quad \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau \quad \dashv \Gamma''\end{array} \quad \odot \in \{<, <=, >=, >\}}{\Gamma \vdash e_1 \odot e_2 \dashv \Gamma''}[\textsc{Comp}]$$

Figure 5: Booleans.

$$\frac{\Gamma \vdash e_1 : \tau' \dashv \Gamma' \quad \Gamma' \vdash e_2 : \tau'' \dashv \Gamma'' \qquad \tau_0 := \begin{cases} \tau' & \tau' \neq DoT \wedge \tau'' = DoT \\ \tau'' & \tau'' \neq DoT \wedge \tau' = DoT \\ DoT & otherwise \end{cases}}{\Gamma \vdash \texttt{min}_{\texttt{max}}(e_1, e_2) : \tau_0 \dashv \Gamma''} [\text{DoT-Min-Max}]$$

$$\frac{\begin{array}{lll} \Gamma \vdash & e_1 : \tau' & \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau'' & \dashv \Gamma'' \end{array} \qquad \begin{cases} \textcircled{1} \vee \textcircled{2} \vee \textcircled{3} \vee \textcircled{4} & : \odot = - \\ \textcircled{4} \vee \textcircled{5} \vee \textcircled{6} \vee \textcircled{7} & : \odot = + \\ \textcircled{2} \vee \textcircled{8} & : \odot = \times \\ \textcircled{2} \vee \textcircled{8} & : \odot = \div \end{cases}}{\Gamma \vdash e_1 \odot e_2 : \tau_0 \dashv \Gamma''} [\text{DoT-Int}]$$

$\textcircled{1} : \tau' = T \wedge \tau'' = DoT \rightarrow \tau_0 := DoT$ $\quad \textcircled{2} : \tau' = D \wedge \tau'' = DoT \rightarrow \tau_0 := D$
$\textcircled{3} : \tau' = DoT \wedge \tau'' = T \rightarrow \tau_0 := D$ $\quad \textcircled{4} : \tau' = DoT \wedge \tau'' = D \rightarrow \tau_0 := DoT$
$\textcircled{5} : \tau' = T \wedge \tau'' = DoT \rightarrow \tau_0 := T$ $\quad \textcircled{6} : \tau' = D \wedge \tau'' = DoT \rightarrow \tau_0 := DoT$
$\textcircled{7} : \tau' = DoT \wedge \tau'' = T \rightarrow \tau_0 := T$ $\quad \textcircled{8} : \tau' = DoT \wedge \tau'' = D \rightarrow \tau_0 := D$

Figure 6: DoT Expression Type Inference.

2.3 Dynamic Semantics

In this Section, we define the dynamic semantics of TTS by a series of *reduction rule*. The dynamic semantics (also known as *runtime semantics*) specifies how programs are to be executed. The reduction rules describe operationally a (small step [1]) evaluation relation and they define a transition system. For a reduction $e \longrightarrow e'$, e is the source and e' the target of the reduction.

$$
\begin{array}{rcll}
i(x_1,\ldots,x_n)\{s\} & \longrightarrow & s & [\text{D-Method}] \\
s_1; s_2 & \longrightarrow & s_2 & [\text{D-Stm}] \\
\texttt{return}\ e & \longrightarrow & e & [\text{D-Ret}] \\
\frac{b \longrightarrow b'}{\texttt{while}\ b\ s} & \longrightarrow & s & [\text{D-While}\top] \\
\frac{b \longrightarrow \texttt{false}}{\texttt{while}\ b\ s} & \longrightarrow & \emptyset & [\text{D-While}\bot] \\
x = e & \longrightarrow & e & [\text{D-Assign}] \\
\frac{b \longrightarrow \texttt{true}}{\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2} & \longrightarrow & e_1 & [\text{D-If}\top] \\
\frac{b \longrightarrow \texttt{false}}{\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2} & \longrightarrow & e_2 & [\text{D-If}\bot] \\
\frac{b \longrightarrow b'}{\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2} & \longrightarrow & \texttt{if}\ b'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & [\text{D-If}] \\
obj.m(e_1,\ldots,e_n) & \longrightarrow & C[obj.m(e_1,\ldots,e_n)] & [\text{D-Rt}] \\
\frac{e_1 \longrightarrow e_1' \quad \ldots \quad e_n \longrightarrow e_n'}{C[obj.m(e_1,\ldots,e_n)]} & \longrightarrow & \emptyset & [\text{D-Et}] \\
n & \longrightarrow & \emptyset & [\text{D-Literal}] \\
\texttt{min}(e_1,e_2) & \longrightarrow & \texttt{if}\ e_1 < e_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & [\text{D-Min}] \\
\texttt{max}(e_1,e_2) & \longrightarrow & \texttt{if}\ e_1 > e_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & [\text{D-Max}] \\
x & \longrightarrow & \emptyset & [\text{D-Var}] \\
\frac{e_1 \longrightarrow e_1' \quad e_2 \longrightarrow e_2'}{e_1 \odot e_2} & \longrightarrow & \odot(e_1', e_2') & [\text{D-Int}] \\
b_1 \odot_1 \cdots \odot_{n-1} b_n & \longrightarrow & b_1; \ldots; b_n & [\text{D-Bool}] \\
\frac{b_1 \longrightarrow b_1' \quad b_2 \longrightarrow b_2'}{b_1 \odot b_2} & \longrightarrow & \odot(b_1', b_2') & [\text{D-Comp}]
\end{array}
$$

$$\textbf{Context} \quad C \quad ::= \quad [\,] \;\big|\; C\ obj.m(e_1,\ldots,e_n)$$

A *context* C is an expression with one sub expression replaced by a hole, denoted with [ ]. We use this to force an order on the reduction rules. In this manner, rule D-Rt applies always before rule D-Et.

3 Proof of Soundness of TTS

In this Section we will show soundness of TTS following the approach presented by Wright and Felleisen [2] which relies on two lemmas:

Lemma 1 (Preservation)**.** *As we evaluate a program, its type is preserved at each step. This property is also called* subject reduction.

$$\Gamma \vdash e : \tau \wedge e \longrightarrow e' \implies \Gamma \vdash e' : \tau$$

Lemma 2 (Progress)**.** *Every program either evaluates to a value or can be stepped into another program.*

$$\Gamma \vdash e : \tau \implies e \in \textbf{Value} \vee \exists e'.\ e \longrightarrow e'$$

The proof of soundness of our type system entails from these lemmas. The Preservation Lemma says that if a well-typed program takes a step of evaluation,

then the resulting program is also well-typed. This grants the use of induction on the number of steps taken in $e \longrightarrow^* e'$ to show that e' has the same type as of e. Then, the Progress Lemma can be applied on e' to show that it can continue and, therefore, that the evaluation does not get stuck.

First, we show these lemmas for TTS without the *DoT* type, and later we demonstrate how the lemmas can be extended to it. Notice that, we assume that a program is syntactically correct and all variables and method calls in the language are time related. This assumption is realistic because our type system operates on top of well-typed Java programs and the Time Semantics maps the source programming language into the language supported by TTS, retaining only the time related instructions. Furthermore, to simplify the exposition, we present the typing only for the Time Types, *i.e.*, D and T. Methods and Statements terms that we skip are typed with a **void** type, while Booleans are typed with **bool** type. Since only Expressions rules can type a term with a Time Type, we demonstrate the soundness only for these rules. The proof for the other terms, follows in similar manner.

Proof of Preservation Lemma. We assume $e : \tau \wedge e \longrightarrow e'$ and we want to show that $\Gamma \vdash e' : \tau$. We do this by well-founded induction on typing derivations. Since the number of typing derivation is finite, the relation of sub-derivation is well-founded. Thus, given $e \longrightarrow e'$ there are only the following possible applicable evaluation rules: D-Rt, D-Et, D-Literal, D-Min, D-Max, D-Var, and D-Int.

- Case D-Rt ($e = obj.m(e_1, \ldots, e_n)$). Since we have a typing derivation for e, we know that rule Et is applied on e iff the method call is in the list of RT_τ methods. If the method call is a RT_τ method call, we have the step $obj.m(e_1, \ldots, e_n)) \longrightarrow C[obj.m(e_1, \ldots, e_n))]$. By the induction hypothesis, the rule types e with τ. Furthermore, we know that rule Et does not type method calls and thus, we can conclude that $\Gamma \vdash e' : \tau$. Similarly, in the case the method call is not a RT_τ method call, the rule types e with type **void** and the same conclusion holds.

- Case D-Et ($e = C[obj.m(e_1, \ldots, e_n)]$). The typing derivation of $\Gamma \vdash e : \tau$ must form like this:

$$\frac{\frac{m \in RT_\tau}{\Gamma \vdash obj.m(e_1,\ldots,e_n) : \tau \dashv \Gamma'}}{\Gamma' \vdash C[obj.m(e_1, \ldots, e_n)] : \tau \dashv \Gamma'}$$

  We know that $e \longrightarrow \emptyset$ which eliminates the method call. Hence, we can trivially conclude $\Gamma \vdash \emptyset : \tau$.

- Case D-Literal ($e = n$). This case can be trivially verified since the rule Literal is a constructor for the type D and its only step is termination.

- Case D-Min and D-Max ($e = \frac{\texttt{min}}{\texttt{max}}(e_1, e_2)$). In this case, we cover two different rules that differ only on the boolean expression b used in their step $e \longrightarrow \texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2$, which do not alter the type of the expression. The typing derivation of $\Gamma \vdash e : \tau$ must form like this:
  From the induction hypothesis, we know that $e_1 : \tau$ and $e_2 : \tau$ and following the step rule, we know that $\texttt{if } b \texttt{ then } e_1 : \tau \texttt{ else } e_2'' : \tau$. Hence, we can conclude that $\Gamma \vdash (\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2) : \tau$.

$$\frac{\Gamma \vdash e_1 : \tau \dashv \Gamma' \quad \Gamma' \vdash e_2 : \tau \dashv \Gamma''}{\Gamma \vdash \frac{\texttt{min}}{\texttt{max}}(e_1, e_2) : \tau \dashv \Gamma''}$$

- Case D-Var ($e = x$). The step rule is $x \longrightarrow \emptyset$ and by induction hypothesis we know $\Gamma \vdash x : \tau$. Hence, we conclude $\Gamma \vdash \emptyset : \tau$.
- Case D-Int ($e = \frac{e_1 \longrightarrow e_1' \quad e_2 \longrightarrow e_2'}{e_1 \odot e_2}$). The typing derivation of $\Gamma \vdash e : \tau$ must form like this:

  $$\frac{\frac{\Gamma \vdash e_1' : \tau'}{\Gamma \vdash e_1 : \tau'} \quad \frac{\Gamma \vdash e_2' : \tau''}{\Gamma \vdash e_2 : \tau''}}{\Gamma \vdash e_1 \odot e_2 : \tau_0}$$

  We know that $e \longrightarrow \odot(e_1', e_2')$, so we need to show that $\Gamma \vdash \odot(e_1, e_2) : \tau_0$. This follows by the induction hypothesis where we know $e_1 : \tau'$ and $e_2 : \tau''$, and the definition of $\odot(\cdot, \cdot)$.

□

Proof of Progress Lemma. We assume $\Gamma \vdash e : \tau$ and we want to show that $e \in \mathbf{V} \ \vee \ \exists e'.\ e \longrightarrow e'$, where $\mathbf{V}$ is the set of Value. We prove this by induction on the typing derivation of e. Since for a well-formed program the number of typing derivation is finite, the induction is well-founded. We recall the definition of an expression in TTS:

$$\begin{array}{rcl} e & ::= & n \mid x \mid e_1 \ + \ e_2 \mid e_1 \ - \ e_2 \mid e_1 \ \times \ e_2 \\ & \mid & e_1 \ \div \ e_2 \mid obj.m(e_1, \_, e_n) \mid b \\ & \mid & \frac{\texttt{min}}{\texttt{max}}(e_1, e_2) \end{array}$$

There are six cases:

- Case $e = n$. Since $n \in \mathbf{V}$, it holds.
- Case $e = x$. This case is not possible because we would have $\emptyset \vdash x : \tau$ and from an empty environment no type can be assigned to x, invalidating the hypothesis of well-formed program.
- Case $e = e_1 \odot e_2$, where $\odot \in \{+, -, \times, \div\}$. For $\Gamma \vdash e_1 \odot e_2 : \tau$ there is a typing derivation and it must have the form:

  $$\frac{\begin{array}{lll} \Gamma \vdash & e_1 : \tau' & \dashv \Gamma' \\ \Gamma' \vdash & e_2 : \tau'' & \dashv \Gamma'' \end{array} \quad \begin{cases} \textcircled{1} \vee \textcircled{2} & : \odot = - \\ \textcircled{2} \vee \textcircled{3} \vee \textcircled{4} & : \odot = + \\ \textcircled{4} & : \odot = \times \\ \textcircled{4} & : \odot = \div \end{cases}}{\Gamma \vdash e_1 \odot e_2 : \tau_0 \dashv \Gamma''} [\text{Int}]$$

  Using the induction hypothesis, we know that $e_1 \in \mathbf{V} \ \vee \ \exists e_1' \ .\ e_1 \longrightarrow e_1'$ and $e_2 \in \mathbf{V} \ \vee \ \exists e_2' \ .\ e_2 \longrightarrow e_2'$. We have four different possibilities now:

  - Both e_1 and e_2 are Values. This means that e_1 and e_2 are either a variable or a literal value. Then, $e_1 \odot e_2$ proceeds to $\odot(e_1, e_2)$ as desired.

– e_1 and e_2 are not a Value, then $\exists e_1' \,.\, e_1 \longrightarrow e_1'$ and $\exists e_2' \,.\, e_2 \longrightarrow e_2'$ such that we have:

$$\frac{e_1 \longrightarrow e_1' \quad e_2 \longrightarrow e_2'}{e_1 \odot e_2 \longrightarrow \odot(e_1', e_2')}$$

– e_1 is a Value but e_2 is not. Then, the typing derivation must be of the form:

$$\frac{e_2 \longrightarrow e_2'}{e_1 \odot e_2 \longrightarrow \odot(e_1, e_2')}$$

– e_2 is a Value but e_1 is not. Then, the typing derivation must be of the form:

$$\frac{e_1 \longrightarrow e_1'}{e_1 \odot e_2 \longrightarrow \odot(e_1', e_2)}$$

- Case $e = obj.m(e_1, \_, e_n)$. We enforce an ordering between the typing rules RT and ET using a context C. Thus, first the typing of RT starts doing the step $obj.m(e_1, \ldots, e_n) \longrightarrow C[obj.m(e_1, \ldots, e_n)]$. Then, rule ET can be matched resulting in a derivation of the form:

$$C[obj.m(e_1, \ldots, e_n)] \longrightarrow e_1; \ldots; e_n$$

- Case $e = \texttt{min}(e_1, e_2)$. In this case, we can use the induction hypothesis to derive that $e_1 \in \mathbf{V} \ \vee \ \exists e_1' \,.\, e_1 \longrightarrow e_1'$ and $e_2 \in \mathbf{V} \ \vee \ \exists e_2' \,.\, e_2 \longrightarrow e_2'$. We have four different possibilities now:

  – Both e_1 and e_2 are Values. This means that e_1 and e_2 are either a variable or a literal value. Then, $\texttt{min}(e_1, e_2)$ proceeds to $\texttt{if}\ e_1 < e_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, as desired.

  – e_1 and e_2 are not a Value, such that $\exists e_1' \,.\, e_1 \longrightarrow e_1'$ and $\exists e_2' \,.\, e_2 \longrightarrow e_2'$. Then, the derivation rule must be of the form:

$$\frac{\dfrac{\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 < e_2}}{\Gamma \vdash \texttt{if}\ e_1 < e_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}}{\Gamma \vdash \texttt{min}(e_1, e_2) : \tau}$$

  Hence, $\texttt{min}(e_1, e_2)$ proceeds to $\texttt{if}\ e_1 < e_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, as desired.

  – e_1 is a Value and e_2 is not. Then, it follows from the previous case without a reduction of e_1.

  – e_2 is a Value and e_1 is not. Then, it follows from the previous case without a reduction of e_2.

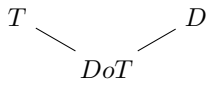
- Case $e = \texttt{max}(e_1, e_2)$. This case follows the same proof for the case $e = \texttt{min}(e_1, e_2)$, where only the `if` expression is changed with $e_1 > e_2$.

□

3.1 Subtyping

We show the previous lemmas without considering the *DoT* time type. Here we first introduce the subtyping rules and we adjust the lemmas for them.

The statement $\tau_1 <: \tau_2$ reads as τ_1 is a sub-type of τ_2, which means that a τ_2 can be used wherever a τ_1 is expected. One way to think of this is in terms of sets of values corresponding to these types. Any value of type τ_1 must also be a value of type τ_2. Assuming a type interpretation $\mathrm{T}[[\tau]]$ that gives the set of elements, we understand $\tau_1 <: \tau_2$ to mean $\mathrm{T}[[\tau_1]] \subseteq \mathrm{T}[[\tau_2]]$. The $<:$ subtyping relation is transitive, *i.e.*, if $\tau_1 <: \tau_2 \ \wedge \ \tau_2 <: \tau_3 \rightarrow \tau_1 <: \tau_3$, and reflexive, *i.e.*, every type is a subtype of itself. Therefore, we define the following subtyping relationship:



From this, the *Preservation* lemma follows trivially because in every instance we can replace *DoT* with either *T* or *D* without encountering typing errors. Instead, for the *Progression* lemma, we need to show that the typing rules for *DoT*, presented in Figure 6, can always proceed. The proof is similar to the third case, *i.e.*, $e = e_1 \odot e_2$, in which we have an extra step before proceeding to compute $\odot(e'_1, e'_2)$. The rule for *DoT* types requires to compute the maximal type, *i.e.*, the higher possible available type in the sub-typing lattice that satisfies the typing rule. This step is accomplished using the unification via pattern matching algorithm [3] with the cases depicted in rule Int. The algorithm always terminates with either the maximal type or a failure if nothing matched. In the case of failure, *DoT* is considered the maximal type and the computation can proceed further, granting the validity of the *Progress* lemma.

From here onwards the proof of repairing soundness starts.

4 Soundness of Repairing

In this Section we will show soundness of our repairing sub-system. The soundness statement is hypothesized in the following lemma.

Lemma 3 (Repair)**.** *If a repair (R) of an expression (e) results in another expression (e') and separate evaluation of repaired expression (e) and repairing expression (e') yields different program states (π and π'), then the yielded states are semantically equivalent.*

$$\forall e, e' \in E, \pi, \pi' \in \Pi : e' = R[\![e]\!] \wedge E[\![e]\!](\pi, \pi') \wedge E[\![e']\!](\pi, \pi'') \Longrightarrow equal(\pi', \pi'')$$

Axiom 1 (Semantic Equivalence)**.** *Two states are semantically equivalent, if both states have same identifiers and their corresponding values.*

$$equal(\pi', \pi'') \Longleftrightarrow \forall i \in I, v, v' \in V : i \in D(\pi) \wedge i \in D(\pi') \wedge \langle i, v\rangle \in S(\pi) \\ \wedge \langle i, v'\rangle \in S(\pi') \Longrightarrow v = v'$$

where I is a memory address (i.e., a variable), V is Value, D is Domain (i.e., mathematical domain) and S is Store (i.e., a look up function).

Axiom 2 (State Composition)**.** *State composition is an operation that takes two states π' and π'' and produces a state π such that*

$$\forall i \in I, v \in V : i \in D(\pi'').\langle i, v\rangle \in S(\pi')$$

The resulting composite state is denoted as $\pi' \circ \pi''$.

5 Proof of Soundness of Repair

Our repair strategy refines timestamp comparison removing possible overflows. The soundness of the repair formally proves that the refined expression is semantically equivalent to the original expression, without suffering of overflows. How to mitigate overflows is presented in the TSE paper[4], therefore, in this proof we prove the semantic equivalence between the expressions before and after the repair strategy is applied.

The proof of soundness of repair is essentially a structural induction on the syntactic domain of expressions (e) as shown in Section 2.1. Since time values involve only arithmetic expressions and their comparison, therefore, only time sensitive expression are proof relevant which are Boolean. Based on the syntactic domain of boolean (b), we only prove the following interesting case, all other cases are analogous to this and they can be rehearsed easily:

- Case $e_1 < e_2$: Both e_1 and e_2 are expressions. We divided the expressions in two categories: (i) pure and (ii) with side-effects. Based on the syntactical expression domain (e), only method calls can modify the program state via side-effects, whereas all the other are pure expressions.

  i. Pure expression are without side-effects and therefore, they cannot alter the program state. Hence, we can conclude that the following repair produce a semantically equivalent expression.

  $$e_1 < e_2 \xrightarrow{\text{repair}} e_1 - e_2 < 0$$

  ii. Expressions with side-effects, *i.e.*, method calls, might change the program state. Therefore, the following repair is produced where $e_1 = obj_1.m_1(e'_1, \ldots, e'_n)$ and $e_2 = obj_2.m_2(e''_1, \ldots, e''_n)$.

  $$\begin{array}{c} obj_1.m_1(e'_1, \ldots, e'_n) < obj_2.m_2(e''_1, \ldots, e''_n) \\ \Big\downarrow \text{repair} \\ obj_1.m_1(e'_1, \ldots, e'_n) - obj_2.m_2(e''_1, \ldots, e''_n) < 0 \end{array}$$

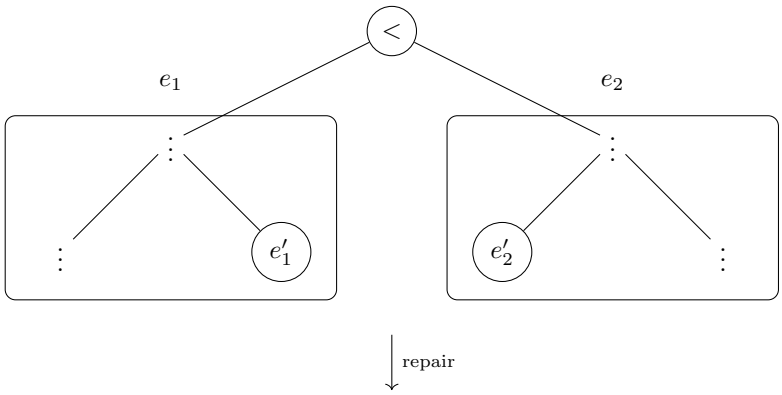
  There are three different cases that could occur:

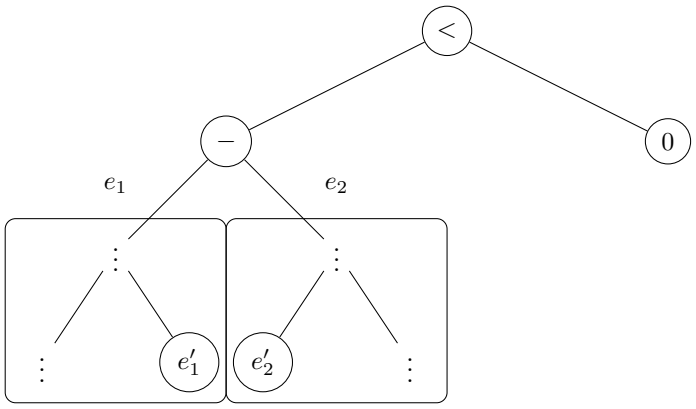
  - Both e_1 and e_2 do not have side effects. Therefore, $E[\![e_1 < e_2]\!]$ produces a program state π' and $R[\![e_1 < e_2]\!]$ produces a program state π'' such that $equal(\pi', \pi'')$ holds.
  - Either e_1 or e_2 has side effects. Let us say e_1 is the expression with side effects. Therefore, given the program state π, $E[\![e_1]\!](\pi, \pi') \wedge E[\![e_2]\!](\pi, \pi'')$ we cannot conclude directly conclude

that $equals(\pi', \pi'')$ holds. However, both expressions must be evaluated to proceed with the computation (see Section 2.3) and since e_1 modifies the state but e_2 not, we can conclude that $\pi_0 = \pi' \circ \pi'' \wedge \pi_1 = \pi'' \circ \pi' \rightarrow equals(\pi_0, \pi_1)$. Hence, $E[\![e_1]\!](\pi, \pi') \wedge E[\![e_2]\!](\pi, \pi'') \rightarrow equals(\pi', \pi'')$. The case where e_2 is the expression with side effects follows the same structure.

– Both e_1 and e_2 have side effects. Since both expressions have side effects, we cannot guarantee the symmetric property, *i.e.*, $\pi_0 = \pi' \circ \pi'' \wedge \pi_1 = \pi'' \circ \pi' \rightarrow \neg equals(\pi_0, \pi_1)$. Although in principle the side effects are preserved in the language of TTS, the dynamic interpretation of the target language might differ due to evaluation order. Since we used our repair technique on the Java programming language, we will proof the soundness of the repair for it. From the Java specification [5] on page 322: "The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.". The repair operator maintains the left to right evaluation order of e_1 and e_2 and therefore, we can conclude that it preserves the semantic equivalence.

We assumed that e_1 and e_2 are simple expressions. In the more general case, they are compound expressions. The proof for such a case is easily performed by induction on the size of the compound expressions following the aforementioned cases. The only case worth to be discussed is where e_1 and e_2 are compound expressions that both contain a method call with side effects, named e_1' and e_2' respectively. Therefore, the expression trees are of the form:





Since the left to right evaluation order of Java corresponds to a pre-order tree traversal of the abstract syntax tree of the expression, we can conclude that e'_1 will always be evaluated before e'_2 after the repair. Thus, the repair operator is sound and it maintains the semantic equivalence between the expressions.

References

[1] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[2] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and computation*, vol. 115, no. 1, pp. 38–94, 1994.

[3] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[4] G. Liva, M. T. Khan, F. Spegni, L. Spalazzi, and M. Pinzger, "Automatic repair of timestamp comparisons - TSE-2018-11-0423 - under review," 2019.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*. Addison-Wesley Professional, 2000.

AAU-SERG-2019-001
ISSN 1872-5392