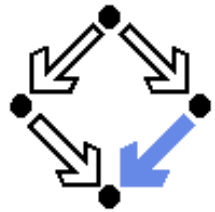
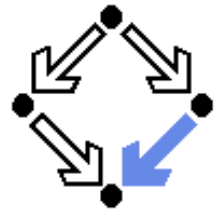
# Programs with Undefined Expressions

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

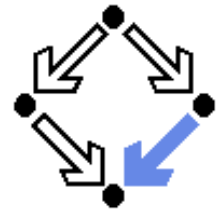
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at





1. **Programs with Undefined Expressions**
2. Avoiding Undefined Expressions
3. Introducing Expression Checks
4. Well-definedness of Expressions

# Undefined Expressions



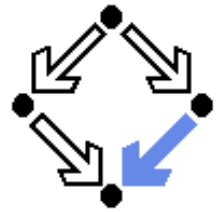
$\bot := \text{SUCH } v : v \notin \mathit{Value}$
$\mathit{Value}_\bot := \mathit{Value} \cup \{\bot\}$
$\mathit{StateFunction}_\bot := \mathit{State} \to \mathit{Value}_\bot$
$\mathit{StatePredicate} := \mathbb{P}(\mathit{State})$

$[\![ \_ ]\!]_\mathrm{D} : \textbf{Expression} \to \textbf{StatePredicate}$
$[\![ E ]\!]_\mathrm{D}(s) \Leftrightarrow \ldots$

$[\![ \_ ]\!]_\bot : \textbf{Expression} \to \textbf{StateFunction}_\bot$
$[\![ E ]\!]_\bot(s) = \text{IF } [\![ E ]\!]_\mathrm{D}(s) \text{ THEN } [\![ E ]\!](s) \text{ ELSE } \bot$

An expression may denote an "undefined" value.

# Programs with Undefined Expressions



$EXP := \text{SUCH } k : \ k \in Key$

$expthrow : State \to State$
$expthrow(s) = throw(s, EXP, read(s, \texttt{VAL}))$

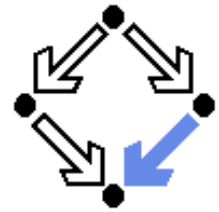
$[\![ \_ ]\!]_\perp$ : **Command** $\to$ **StateRelation**

...

$[\![ I{=}E ]\!]_\perp(s, s') \Leftrightarrow$
  LET $v = [\![ E ]\!]_\perp(s)$ IN
  IF $v = \perp$
    THEN $s' = expthrow(s)$
    ELSE $s' = write(s, I, v)$

When encountering an undefined value, every command raises an "evaluation exception".

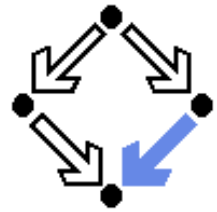
# Loops and Undefined Expressions



$$
\begin{aligned}
&\mathit{finiteExecution}_\perp : \\
&\mathbb{P}(\mathbb{N} \times \mathit{State}^\infty \times \mathit{State}^\infty \times \mathit{State} \times \mathit{StateFunction}_\perp \times \mathit{StateRelation}) \\
&\mathit{finiteExecution}_\perp(k, t, u, s, E, C) \Leftrightarrow \\
&\quad t(0) = s \wedge u(0) = s \wedge \\
&\quad \forall i \in \mathbb{N}_k : \\
&\qquad \neg \mathit{breaks}(\mathit{control}(u(i))) \wedge \mathit{executes}(\mathit{control}(t(i))) \wedge \\
&\qquad E(t(i)) \neq \perp \wedge E(t(i)) = \text{TRUE} \wedge C(t(i), u(i+1)) \wedge \\
&\qquad \text{IF } \mathit{continues}(\mathit{control}(u(i+1))) \vee \mathit{breaks}(\mathit{control}(u(i+1))) \\
&\qquad\quad \text{THEN } t(i+1) = \mathit{execute}(u(i+1)) \\
&\qquad\quad \text{ELSE } t(i+1) = u(i+1)
\end{aligned}
$$

Also an evaluation exception lets a loop terminate.

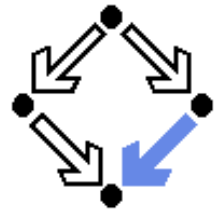
# Loops and Undefined Expressions



$$
\begin{aligned}
&[\![ \texttt{while (}E\texttt{) } C ]\!]_\perp(s, s') \Leftrightarrow \\
&\quad \exists k \in \mathbb{N}, t, u \in \mathit{State}^\infty : \\
&\qquad \mathit{finiteExecution}_\perp(k, t, u, s, [\![ E ]\!]_\perp, [\![ C ]\!]_\perp) \wedge \\
&\qquad ([\![ E ]\!]_\perp(t(k)) = \perp \vee [\![ E ]\!]_\perp(t(k)) \neq \text{TRUE} \vee \\
&\qquad\quad \neg(\mathit{executes}(\mathit{control}(u(k))) \vee \mathit{continues}(\mathit{control}(u(k))))) \wedge \\
&\qquad \text{IF } (\mathit{executes}(\mathit{control}(u(k))) \vee \mathit{continues}(\mathit{control}(u(k)))) \\
&\qquad\quad \wedge [\![ E ]\!]_\perp(t(k)) = \perp \\
&\qquad\quad \text{THEN } \mathit{expthrow}(t(k)) = s' \\
&\qquad\quad \text{ELSE } t(k) = s'
\end{aligned}
$$

Evaluation exception lets loop terminate in a “throwing” state.

# Relationship to Original Semantics



- If the expression value is not undefined, it is the same as in the original semantics:

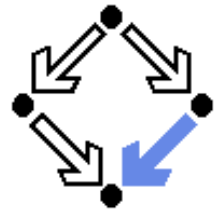
$$\forall E \in \text{Expression}:$$
$$\quad \forall s \in \mathit{State}:$$
$$\quad\quad [\![ E ]\!]_\perp(s) \neq \perp \Rightarrow [\![ E ]\!]_\perp(s) = [\![ E ]\!](s)$$

- A command produces either an original poststate or an "exception throwing" state (provided that such exceptions are not caught).

$$\forall C \in \text{Command}:$$
$$\quad C \text{ has no subcommand } \texttt{try } C_1 \texttt{ catch(}\mathit{EXP}\ I\texttt{)} C_2 \Rightarrow$$
$$\quad\quad \forall s, s' \in \mathit{State} : \mathit{executes}(\mathit{control}(s)) \wedge [\![ C ]\!]_\perp(s, s') \Rightarrow$$
$$\quad\quad\quad [\![ C ]\!](s, s') \vee \mathit{expthrows}(\mathit{control}(s'))$$

$$\mathit{expthrows} : \mathbb{P}(\mathit{Control})$$
$$\mathit{expthrows}(c) \Leftrightarrow \mathit{throws}(c) \wedge \mathit{key}(c) = \mathit{EXP}$$

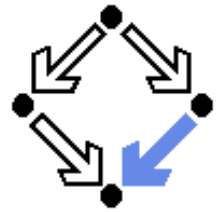
# Reasoning about Undefined Expressions



How to reason about programs with undefined expressions?

1. Construct a new calculus for the new semantics.
   - Extra work; rules become considerably more complicated.
2. Rule out that program may encounter such expressions.
   - Typically only erroneous programs encounter such expressions.
   - Introduce calculus for verifying "well-definedness" of programs.
3. Transform programs by adding "checks" for undefined expressions.
   - Transformed programs in original semantics should have same behavior as original programs in new semantics.
   - By (automatically) inserting such checks, we can stick to the verification calculi of the original semantics.

We deal with the last two solutions.



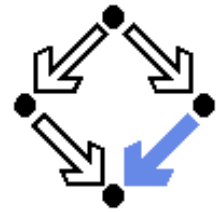
1. Programs with Undefined Expressions

2. **Avoiding Undefined Expressions**

3. Introducing Expression Checks

4. Well-definedness of Expressions

# Avoiding Undefined Expressions

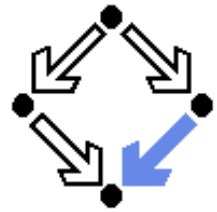


Introduce judgement for verifying the "well-definedness" of a program.

$$
\begin{array}{l}
C \checkmark P \Leftrightarrow \\
\quad C \text{ has no subcommand } \texttt{try } C_1 \texttt{ catch(}EXP\ I\texttt{) } C_2 \ \Rightarrow \\
\quad\quad \forall s, s' \in \mathit{State} : \mathit{executes}(\mathit{control}(s)) \wedge [\![ P ]\!](s) \Rightarrow \\
\quad\quad\quad ([\![ C ]\!](s, s') \Leftrightarrow [\![ C ]\!]_{\perp}(s, s'))
\end{array}
$$

When executed in a state where property $P$ holds, command $C$ does not encounter an undefined value.

# Avoiding Undefined Expressions



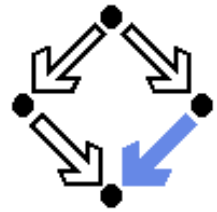
$$\_ \overset{\mathrm{D}}{\simeq} \_ : \mathbb{P}(\text{Expression} \times \text{Formula})$$

$$E \overset{\mathrm{D}}{\simeq} F_D \Leftrightarrow \forall s \in \mathit{State} : [\![ E ]\!]_{\mathrm{D}}(s) \Leftrightarrow [\![ F_D ]\!](s)$$

$$\frac{E \overset{\mathrm{D}}{\simeq} F_D \qquad \forall s \in \mathit{State} : [\![ (\texttt{now.executes AND } P) \texttt{ => } F_D ]\!](s)}{I{=}E \checkmark P}$$

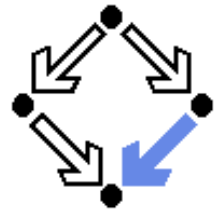
Rules match those of "termination calculus" $C \downarrow F$.

# Loops Avoiding Undefined Expressions

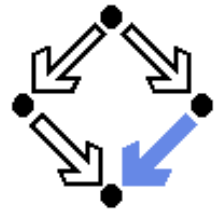


$$
\frac{\begin{array}{l}
E \overset{\mathrm{D}}{\simeq} F_D \\
\forall s \in \mathit{State} : [\![ (\texttt{now.executes AND } P) \texttt{ => } F_D ]\!](s) \\
E \simeq H \\
C : [\, F \,]^{F_c, F_b, F_r, \{K_1, \ldots, K_m\}}_{I_1, \ldots, I_n} \\
\ldots \\
R \;=\; H \texttt{ AND} \\
\quad \texttt{EXISTS } \$J_1, \ldots, \$J_n \texttt{: EXSTATE } \#I_s \texttt{:} \\
\qquad P[\#I_s/\texttt{now}][\$J_1/I_1, \ldots, \$J_n/I_n] \\
C \checkmark R \\
\mathrm{POST}(C, R) = Q \\
\forall s \in \mathit{State} : \\
\quad [\![ (\texttt{(now.executes OR now.continues)} \\
\qquad \texttt{AND } Q) \texttt{ => } F_D ]\!](s)
\end{array}}{\texttt{while}(E)\ C \checkmark P}
$$

# Loops Avoiding Undefined Expressions



$$
\begin{array}{l}
\ldots \\
\mathit{Invariant}(G, H, F)_{I_1,\ldots,I_n} \\
R \;=\; H \;\texttt{AND} \\
\quad \texttt{EXISTS \$}J_1, \ldots, \texttt{\$}J_n\texttt{: EXSTATE \#}I_s, \texttt{\#}I_t\texttt{:} \\
\qquad P[\texttt{\#}I_s/\texttt{now}][\texttt{\$}J_1/I_1, \ldots, \texttt{\$}J_n/I_n]\ \texttt{AND} \\
\qquad (G[\texttt{now/next}][I_1/I_1\texttt{'}, \ldots, I_n/I_n\texttt{'}] \\
\qquad\quad [\texttt{\#}I_s/\texttt{now}][\texttt{\$}J_1/I_1, \ldots, \texttt{\$}J_n/I_n]\ \texttt{=>} \\
\qquad\quad G[\texttt{\#}I_s/\texttt{now}][\texttt{\$}J_1/I_1, \ldots, \texttt{\$}J_n/I_n] \\
\qquad\qquad [\texttt{\#}I_t/\texttt{next}][I_1/I_1\texttt{'}, \ldots, I_n/I_n\texttt{'}]\ \texttt{AND} \\
\qquad\quad (\texttt{!(\#}I_t\texttt{.continues OR \#}I_t\texttt{.breaks) =>} \\
\qquad\qquad \texttt{now == \#}I_t)) \\
\ldots \\
\hline
\texttt{while(}E\texttt{)}\ C \checkmark P
\end{array}
$$



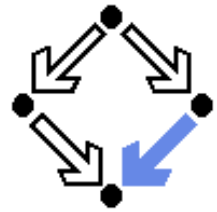
1. Programs with Undefined Expressions

2. Avoiding Undefined Expressions

**3. Introducing Expression Checks**

4. Well-definedness of Expressions

# Checked Programs

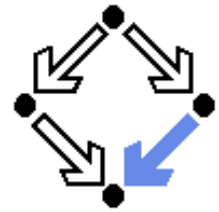


Transform program such that it "mimics" new semantics.

$$\begin{array}{l} \textsc{check}(C) = C' \;\Leftrightarrow \\ \quad C \text{ has no subcommand } \texttt{try}\ C_1\ \texttt{catch(}\mathit{EXP}\ I\texttt{)}\ C_2 \;\Rightarrow \\ \quad\quad \forall s, s' \in \mathit{State} : \mathit{executes}(\mathit{control}(s)) \Rightarrow \\ \quad\quad\quad [\![\, C \,]\!]_\perp(s, s') \Leftrightarrow [\![\, C' \,]\!](s, s') \end{array}$$

Program $C'$ behaves in the old semantics as program $C$ does in the original semantics.

# Checked Programs



$$\_ \overset{\mathrm{D}}{\simeq} \_ : \mathbb{P}(\text{Expression} \times \text{Expression})$$

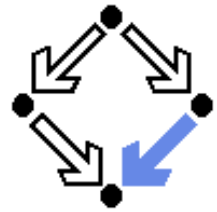
$$E \overset{\mathrm{D}}{\simeq} E_D :\Leftrightarrow \forall s \in \mathit{State} : [\![ E ]\!]_{\mathrm{D}}(s) \Leftrightarrow [\![ E_D ]\!](s) = \mathrm{TRUE}$$

$$\mathrm{CHECK}\ E_D \equiv \texttt{if (!}E_D\texttt{) throw } \mathit{EXP}\ \texttt{VAL}$$

$$\frac{E \overset{\mathrm{D}}{\simeq} E_D}{\mathrm{CHECK}(I = E) = \mathrm{CHECK}\ E_D;\ I = E}$$

Sound calculus for all commands (except loops).

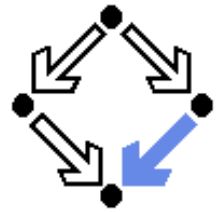
# Checked Loops (without `continue`)



$$\frac{\begin{array}{l} E \overset{\mathrm{D}}{\simeq} E_D \\ [\,C\,]^{\texttt{FALSE},F_b,F_r,\{K_1,\ldots,K_m\}}_{I_1,\ldots,I_n} \\ \textsc{check}(C) = C' \end{array}}{\begin{array}{l} \textsc{check}(\texttt{while}\ (E)\ C) = \\ \quad \textsc{check}\ E_D\texttt{;}\ \texttt{while}\ (E)\ (C'\texttt{;}\ \textsc{check}\ E_D) \end{array}}$$

Rule is sound, if loop does not contain `continue`.

# Checked Loops (with `continue`)



Every execution of `continue` is immediately followed by the evaluation of the expression of the enclosing loop.

- Idea: precede every occurrence of `continue` by a check of the corresponding loop expression.

  ```
  // transformation of continue
  {check E_D; continue }
  ```

  - Calculus needs to take into account enclosing loop expression.

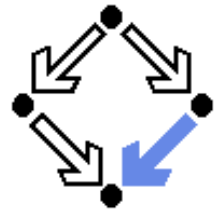
- Caveat: `continue` may occur in a different variable context.

  ```
  while (I > 0) {...{var I; ...continue; ...}...}
  ```

  - Can be prevented by automatic renaming of local variables.

Adapt calculus to perform transformation correspondingly.

# Checked Loops



$$\frac{E \overset{\mathrm{D}}{\simeq} F_D \qquad \forall s \in \mathit{State} : [\![ (\texttt{now.executes AND } P) \texttt{ => } F_D ]\!](s)}{\textsc{check}_{E'}(I\texttt{=}E) = P}$$

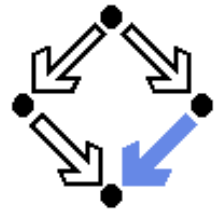
$$\frac{\forall s_1, s_2 \in \mathit{State} : s_1 = s_2 \textsc{ except } I \Rightarrow [\![ E' ]\!]_\perp(s_1) = [\![ E' ]\!]_\perp(s_2) \qquad \textsc{check}_{E'}(C) = C'}{\textsc{check}_{E'}(\texttt{var } I\texttt{; } C) = \texttt{var } I\texttt{; } C'}$$

$$\frac{E' \overset{\mathrm{D}}{\simeq} E_D}{\textsc{check}_{E'}(\texttt{continue}) = \texttt{check } E_D\texttt{; continue}}$$

$$\frac{E \overset{\mathrm{D}}{\simeq} E_D \qquad \textsc{check}_{E}(C) = C'}{\textsc{check}_{E'}(\texttt{while (}E\texttt{) } C) = \textsc{check } E_D\texttt{; while (}E\texttt{) (}C'\texttt{; }\textsc{check } E_D\texttt{)}}$$

What about the soundness of the transformation?

# Soundness of Transformation

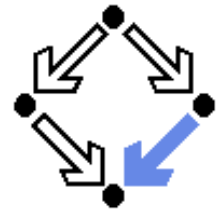


Transformed command is *not* completely equivalent to original one.

- Original command may (in new semantics) yield a "continuing" state with an undefined loop expression value.
  - Transformed command then generates (in original semantics) a "throwing" state with an "evaluation exception".

Equivalence of programs is not completely preserved!

# Soundness of Transformation

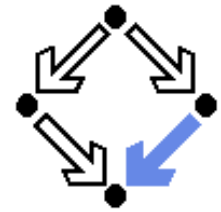


Assume $C$ has no subcommand `try` $C_1$ `catch(`$EXP\ I$`)` $C_2$.

$$
\begin{array}{l}
\textsc{check}_{E'}(C) = C' \ \Leftrightarrow \\
\quad \forall s, s' \in \mathit{State} : \mathit{executes}(\mathit{control}(s)) \Rightarrow \\
\qquad (\llbracket\, C \,\rrbracket_\perp(s, s') \wedge \neg \llbracket\, C' \,\rrbracket(s, s') \ \Rightarrow \\
\qquad\quad \mathit{continues}(\mathit{control}(s')) \wedge \llbracket\, E' \,\rrbracket_\perp(s') = \perp \wedge \\
\qquad\quad \llbracket\, C' \,\rrbracket(s, \mathit{expthrow}(s')) \wedge \\
\qquad (\llbracket\, C' \,\rrbracket(s, s') \wedge \neg \llbracket\, C \,\rrbracket_\perp(s, s') \ \Rightarrow \\
\qquad\quad \exists s'' \in \mathit{State} : \llbracket\, C \,\rrbracket_\perp(s, s'') \wedge s' = \mathit{expthrow}(s'') \wedge \\
\qquad\qquad \mathit{continues}(\mathit{control}(s'')) \wedge \llbracket\, E' \,\rrbracket_\perp(s'') = \perp) \wedge \\
\qquad (\llbracket\, C' \,\rrbracket(s, s') \ \Rightarrow \\
\qquad\quad \neg(\mathit{continues}(\mathit{control}(s')) \wedge \llbracket\, E' \,\rrbracket_\perp(s') = \perp))
\end{array}
$$

There is a set of states on which the transformed command behaves different from the original one.

# Relationship to Original Program

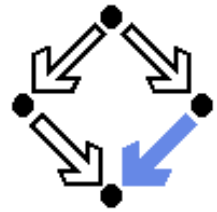


Assume $\text{CHECK}_{E'}(C) = C'$ can be derived.

$$\forall s, s' \in \mathit{State} : \mathit{executes}(\mathit{control}(s)) \wedge [\![ C ]\!]_{\perp}(s, s') \Rightarrow$$
$$\text{IF } \mathit{continues}(\mathit{control}(s')) \wedge [\![ E' ]\!]_{\perp}(s'') = \perp$$
$$\text{THEN } [\![ C' ]\!](s, \mathit{expthrow}(s'))$$
$$\text{ELSE } [\![ C' ]\!](s, s')$$

(Only) if it can be ruled out that the execution of $C$ yields a continuing state with undefined loop expression value, then a valid specification $C' : F$ (in the original semantics) also yields a valid specification $C : F$ (in the new semantics).

# Consequences for Reasoning

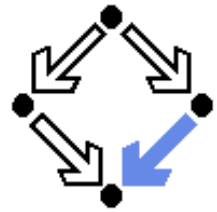


$C = \texttt{while}\ (E)\ C_0$
$\text{CHECK}_E(C_0) = C_0'$
$C_0' : [\, F \,]^{F_c, F_b, F_r, \{K_1, \ldots, K_m\}}_{I_1, \ldots, I_n}$

- Case $F_c = \texttt{F}$
  - Execution of loop body $C_0$ does not result in a continuing state.
  - $F$ is valid specification of $C_0$ in new semantics.
- Case $C \checkmark^{\text{L}} P$
  - Variant of "well-definedness" calculus to show that loop expressions are only evaluated in states where their values are defined.
  - $F$ is valid specification of $C_0$ in new semantics.

(Only) in these cases, equivalence to new semantics holds.

# Consequences for Reasoning

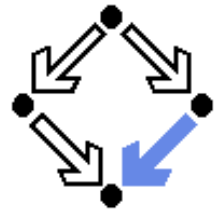


$C =$ `while (`$E$`)` $C_0$
$\text{CHECK}_E(C_0) = C_0'$
$C_0' : [\, F \,]^{F_c, F_b, F_r, \{K_1, \ldots, K_m\}}_{I_1, \ldots, I_n}$

- General:
  - None of above restrictions applies.
  - Only $F' =$ (`NOT next.throws` $EXP$) => $F$ is valid specification of $C_0$ in new semantics.
- Consequences for invariant reasoning:
  - Either use weaker specification $F'$ of loop body $C_0$.
  - Or write invariant for $C_0'$ (in original semantics) rather than for $C_0$ (in new semantics).

To preserve full reasoning power, the transparency of the transformation has to be sacrificed.

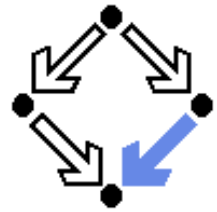
# Reasoning about Undefined Expressions



Develop verifier where user may choose between two modes.

- "Well-Definedness": check for well-definedness of programs, report violations of well-definedness as errors.
  - Default mode (only mode in e.g. ESC/Java2).
  - Evaluation exceptions are unintended in most programs.
- "Expression Checks": transform programs by introducing expression checks and verify transformed programs.
  - Undefined expression values give rise to evaluation exceptions.
  - Users may mostly think of programs as in the "new semantics".
  - In loop invariants users must replace "continuing" states with undefined loop expressions by states with "evaluation exceptions".
    - Transformation not completely transparent.

All in all, more complex than originally (naively) expected.



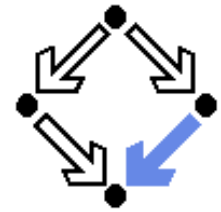
1. Programs with Undefined Expressions

2. Avoiding Undefined Expressions

3. Introducing Expression Checks

**4. Well-definedness of Expressions**

# Expressions



For introducing expression checks, expression language must be rich enough to describe well-definedness of any expression.

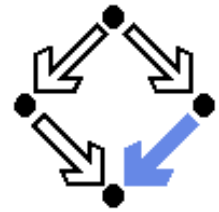
$E \in \text{Expression}.$

$E ::=$

$I \mid$
`0` $\mid$ `1` $\mid$ `-`$E \mid E_1$`+`$E_2 \mid E_1$`/`$E_2 \mid$
`true` $\mid$ `false` $\mid$ `!`$E \mid E_1$`&&`$E_2 \mid E_1$`||`$E_2 \mid$
$E_1$`==`$E_2 \mid E_1$`<`$E_2 \mid E_1$`<=`$E_2 \mid$
`null` $\mid$ `new` $E \mid E_1$`.length` $\mid E_1$`[`$E_2$`]` $\mid E_1$`[`$E_2$`|->`$E_3$`]`

We omit semantics of the expressions.

# Definedness Expressions



$[\![ \_ ]\!]_{\mathrm{DE}}$ : **Expression** $\rightarrow$ **Expression**

$[\![ I ]\!]_{\mathrm{DE}} = \texttt{true}$

...

$[\![ \texttt{-}E ]\!]_{\mathrm{DE}} = [\![ E ]\!]_{\mathrm{DE}}$

$[\![ E_1\texttt{+}E_2 ]\!]_{\mathrm{DE}} = [\![ E_1 ]\!]_{\mathrm{DE}}$ && $[\![ E_2 ]\!]_{\mathrm{DE}}$

$[\![ E_1\texttt{/}E_2 ]\!]_{\mathrm{DE}} = [\![ E_1 ]\!]_{\mathrm{DE}}$ && $[\![ E_2 ]\!]_{\mathrm{DE}}$ && !($E_2$ == 0)

...

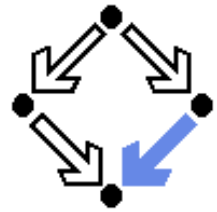
$[\![ \texttt{new}\ E ]\!]_{\mathrm{DE}} = [\![ E ]\!]_{\mathrm{DE}}$ && 0 <= $E$

$[\![ E\texttt{.length} ]\!]_{\mathrm{DE}} = [\![ E ]\!]_{\mathrm{DE}}$&& !($E$ == null)

$[\![ E_1\texttt{[}E_2\texttt{]} ]\!]_{\mathrm{DE}} = [\![ E_1 ]\!]_{\mathrm{DE}}$ && $[\![ E_2 ]\!]_{\mathrm{DE}}$ &&
!($E_1$ == null) && 0 <= $E_2$ && $E_2$ < $E_1$.length

$[\![ E_1\texttt{[}E_2\texttt{|->}E_3\texttt{]} ]\!]_{\mathrm{DE}} = [\![ E_1 ]\!]_{\mathrm{DE}}$ && $[\![ E_2 ]\!]_{\mathrm{DE}}$ && $[\![ E_3 ]\!]_{\mathrm{DE}}$ &&
!($E_1$ == null) && 0 <= $E_2$ && $E_2$ < $E_1$.length

# Formulas



For reasoning about well-definedness, formula/term language must be rich enough to describe well-definedness of any expression.

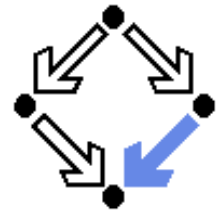
$[\![ \_ ]\!]$ : **Predicate** $\rightarrow$ *Predicate*

$[\![ \texttt{eq} ]\!](v_1, v_2) \Leftrightarrow v_1 = v_2$
$[\![ \texttt{lt} ]\!](v_1, v_2) \Leftrightarrow v_1 < v_2$
$[\![ \texttt{le} ]\!](v_1, v_2) \Leftrightarrow v_1 \leq v_2$

$[\![ \_ ]\!]$ : **Function** $\rightarrow$ *Function*

$[\![ \texttt{error} ]\!] = 0$
$[\![ \texttt{zero} ]\!] = 0$
$[\![ \texttt{one} ]\!] = 1$
$[\![ \texttt{neg} ]\!](v) = \ominus v$
$[\![ \texttt{add} ]\!](v_1, v_2) = v_1 \oplus v_2$
$[\![ \texttt{div} ]\!](v_1, v_2) = v_1 \oslash v_2$
$[\![ \texttt{null} ]\!] = \mathit{null}$
$[\![ \texttt{new} ]\!](v) = \mathit{new}(v)$
$[\![ \texttt{length} ]\!](v) = \mathit{length}(v)$
$[\![ \texttt{put} ]\!](v_1, v_2, v_3) = \mathit{put}(v_1, v_2, v_3)$
$[\![ \texttt{get} ]\!](v_1, v_2) = \mathit{get}(v_1, v_2)$

# Definedness Formulas



$[\![ \_ ]\!]_{\mathrm{DF}}$ : **Expression** $\rightarrow$ **Formula**

$[\![ I ]\!]_{\mathrm{DF}} = \texttt{TRUE}$

...

$[\![ \texttt{-}E ]\!]_{\mathrm{DF}} = [\![ E ]\!]_{\mathrm{DF}}$

$[\![ E_1\texttt{+}E_2 ]\!]_{\mathrm{DF}} = [\![ E_1 ]\!]_{\mathrm{DF}}$ AND $[\![ E_2 ]\!]_{\mathrm{DF}}$

$[\![ E_1\texttt{/}E_2 ]\!]_{\mathrm{DF}} = [\![ E_1 ]\!]_{\mathrm{DF}}$ AND $[\![ E_2 ]\!]_{\mathrm{DF}}$ && !eq($[\![ E_2 ]\!]_{\mathrm{TERM}}$, zero)

...

$[\![$ new $E ]\!]_{\mathrm{DF}} = [\![ E ]\!]_{\mathrm{DF}}$ AND le(zero, $[\![ E ]\!]_{\mathrm{TERM}}$)

$[\![ E$.length$]\!]_{\mathrm{DF}} = [\![ E ]\!]_{\mathrm{DF}}$ AND !eq($[\![ E ]\!]_{\mathrm{TERM}}$, null)

$[\![ E_1[E_2] ]\!]_{\mathrm{DF}} = [\![ E_1 ]\!]_{\mathrm{DF}}$ AND $[\![ E_2 ]\!]_{\mathrm{DF}}$ AND

!eq($[\![ E_1 ]\!]_{\mathrm{TERM}}$, null) AND

le(zero,$[\![ E_2 ]\!]_{\mathrm{TERM}}$) AND

lt($[\![ E_2 ]\!]_{\mathrm{TERM}}$,length($[\![ E_1 ]\!]_{\mathrm{TERM}}$))

$[\![ E_1[E_2\texttt{|->}E_3] ]\!]_{\mathrm{DF}} = [\![ E_1 ]\!]_{\mathrm{DF}}$ AND $[\![ E_2 ]\!]_{\mathrm{DF}}$ AND $[\![ E_3 ]\!]_{\mathrm{DF}}$ AND

!eq($[\![ E_1 ]\!]_{\mathrm{TERM}}$, null) AND

le(zero,$[\![ E_2 ]\!]_{\mathrm{TERM}}$) AND

lt($[\![ E_2 ]\!]_{\mathrm{TERM}}$,length($[\![ E_1 ]\!]_{\mathrm{TERM}}$))