Specification of Source §2 Stepper—2022 edition

Martin Henz, Tan Yee Jian, Zhang Xinyi, Zhao Jingjing,
Zachary Chua, Peter Jung, Poomklao Teerawatthanaprapha,
Hao Sitong

National University of Singapore
School of Computing

October 21, 2025

Reduction

The *reducer* $\Rightarrow$ is a partial function from programs/statements/expressions to programs/statements/expressions (with slight abuse of notation via overloading) and $\Rightarrow^*$ is its reflexive transitive closure. A *reduction* is a sequence of programs $p_1 \Rightarrow \cdots \Rightarrow p_n$, where p_n is not reducible, i.e. there is no program q such that $p_n \Rightarrow q$. Here, the program p_n is called the *result of reducing* p_1.
If p_n, the result of reducing a program, is of the form v`;`, we call v the result of the program evaluation. If p_n is the empty sequence of statements, we declare the result of the program evaluation to be the value undefined. Reduction can get "stuck": the result can be a program that has neither of these two forms, which corresponds to a runtime error.
A *value* is a primitive number expression, primitive boolean expression, a primitive string expression, a function definition expression or a function declaration expression.
The *substitution* function $p[n := v]$ on programs/statements/expressions replaces every free occurrence of the name n in statement p by value v. Care must be taken to introduce and preserve co-references in this process; substitution can introduce cyclic references in the result of the substitution. For example, n may occur free in v, in which case every occurrence of n in p will be replaced by v such that n in v refers cyclically to the node at which the replacement happens.

Programs

Program-intro: In a sequence of statements, we can always reduce the first one that isn't a value statement.

$$\frac{\textit{statement} \Rightarrow \textit{statement}'}{[\textit{value}\texttt{;}]\ \ \textit{statement}\ldots \Rightarrow [\textit{value}\texttt{;}]\ \ \textit{statement}'\ldots}$$

Program-reduce: When the first two statements in a program are value statements, the first one can be removed.

$$\frac{\textit{v1}, \textit{v2} \text{ are values}}{\textit{v1}\texttt{;}\ \ \textit{v2}\texttt{;}\ \ \textit{statement}\ldots \Rightarrow \textit{v2}\texttt{;}\ \ \textit{statement}\ldots}$$

Eliminate-function-declaration: Function declarations as the first statement optionally after one value statement are substituted in the remaining statements.

$$\frac{f = \texttt{function}\ \textit{name}\ \texttt{(}\ \textit{parameters}\ \texttt{)}\ \textit{block}}{[\textit{value}\texttt{;}]\ \ f\ \textit{statement}\ldots\ \Rightarrow\ [\textit{value}\texttt{;}]\ \ \textit{statement}\ldots[\textit{name} := f]}$$

Eliminate-constant-declaration: Constant declarations as the first statement optionally after one value statement are substituted in the remaining statements.

$$\frac{c = \texttt{const}\ \textit{name}\ \texttt{=}\ v}{[\textit{value}\texttt{;}]\ \ c\ \textit{statement}\ldots\ \Rightarrow\ [\textit{value}\texttt{;}]\ \ \textit{statement}\ldots[\textit{name} := v]}$$

Statements: Expression statements

Expression-statement-reduce: An expression statement is reducible if its expression is reducible.

$$\frac{e \Rightarrow e'}{e\texttt{;} \Rightarrow e'\texttt{;}}$$

Statements: Constant declarations

Evaluate-constant-declaration: The right-hand expressions in constant declarations are evaluated.

$$\frac{\textit{expression} \Rightarrow \textit{expression}'}{\texttt{const}\ \textit{name}\ \texttt{=}\ \textit{expression} \Rightarrow \texttt{const}\ \textit{name}\ \texttt{=}\ \textit{expression}'}$$

Statements: Conditionals

Conditional-statement-predicate: A conditional statement is reducible if its predicate is reducible.

$$\frac{e \Rightarrow e'}{\texttt{if (} e \texttt{) \{ } \cdots \texttt{ \} else \{ } \cdots \texttt{ \}} \Rightarrow \texttt{if (} e' \texttt{) \{ } \cdots \texttt{ \} else \{ } \cdots \texttt{ \}}}$$

SICP-JS Exercise 4.8 specifies that a conditional statement where the taken branch is non-value-producing is value-producing, with its value being `undefined`.

Conditional-statement-consequent: Immediately within a program or block statement, a conditional statement whose predicate is true reduces to the consequent block.

$$\frac{}{\texttt{if (true) \{ } \textit{program}_1 \texttt{ \} else \{ } \textit{program}_2 \texttt{ \}} \Rightarrow \texttt{\{ undefined; } \textit{program}_1 \texttt{ \}}}$$

Conditional-statement-alternative: Immediately within a program or block statement, a conditional statement whose predicate is false reduces to the alternative block.

$$\frac{}{\texttt{if (false) \{ } \textit{program}_1 \texttt{ \} else \{ } \textit{program}_2 \texttt{ \}} \Rightarrow \texttt{\{ undefined; } \textit{program}_2 \texttt{ \}}}$$

We can remove the `undefined;` if the conditional statement is in a block expression without affecting the return value of the block expression. This is performed to reduce the surprise factor.

Conditional-statement-blockexpr-consequent: Immediately within a block expression, a conditional statement whose predicate is true reduces to the consequent block.

$$\frac{}{\texttt{\{ } [\textit{value}\texttt{;}] \texttt{ if (true) \{ } \textit{program}_1 \texttt{ \} else \{ } \textit{program}_2 \texttt{ \} \}} \Rightarrow \texttt{\{} [\textit{value}\texttt{;}] \texttt{ \{ } \textit{program}_1 \texttt{ \}\}}}$$

Conditional-statement-blockexpr-alternative: Immediately within a block expression, a conditional statement whose predicate is false reduces to the alternative block.

$$\frac{}{\texttt{\{ } [\textit{value}\texttt{;}] \texttt{ if (false) \{ } \textit{program}_1 \texttt{ \} else \{ } \textit{program}_2 \texttt{ \} \}} \Rightarrow \texttt{\{} [\textit{value}\texttt{;}] \texttt{ \{ } \textit{program}_2 \texttt{ \}\}}}$$

Statements: Blocks

Block-statement-intro: A block statement is reducible if its body is reducible.

$$\frac{\textit{program} \Rightarrow \textit{program}'}{\texttt{\{ } \textit{program} \texttt{ \}} \Rightarrow \texttt{\{ } \textit{program}' \texttt{ \}}}$$

Block-statement-single-reduce: A block statement consisting of a single value statement reduces to that value statement.

$$\frac{}{\texttt{\{}\mathit{value}\texttt{;}\,\texttt{\}} \Rightarrow \mathit{value}\texttt{;}}$$

Block-statement-empty-reduce: An empty block statement (or a non-value- producing one, which will reduce to the empty block statement) is non-value-producing, i.e. reduces to ε.

$$\frac{}{\texttt{\{\}} \Rightarrow \varepsilon}$$

Expresssions: Blocks

Block-expression-intro: A block expression is reducible if its body is reducible.

$$\frac{\mathit{program} \;\Rightarrow\; \mathit{program}'}{\texttt{\{}\; \mathit{program}\; \texttt{\}} \;\Rightarrow\; \texttt{\{}\; \mathit{program}'\; \texttt{\}}}$$

Block-expression-single-reduce: A block expression consisting of a single value statement reduces to **`undefined`**.

$$\frac{}{\texttt{\{}\mathit{value}\texttt{;}\,\texttt{\}} \Rightarrow \texttt{undefined}}$$

Block-expression-empty-reduce: An empty block expression reduces to **`undefined`**.

$$\frac{}{\texttt{\{\}} \Rightarrow \texttt{undefined}}$$

Block-expression-return-reduce-1: In a block expression starting with a value statement and then a return statement, the value statement can be removed.

$$\frac{}{\texttt{\{}\mathit{value}\texttt{;}\;\; \texttt{return}\;\; \mathit{return\text{-}expression}\texttt{;}\;\; \mathit{statement}\ldots\texttt{\}} \;\Rightarrow\; \texttt{\{return}\;\; \mathit{return\text{-}expression}\texttt{;}\;\; \mathit{statement}\ldots\texttt{\}}}$$

Block-expression-return-reduce-2: A block expression starting with a return statement reduces to the expression of the return statement.

$$\frac{}{\texttt{\{return}\;\; \mathit{return\text{-}expression}\texttt{;}\;\; \mathit{statement}\ldots\,\texttt{\}} \Rightarrow \mathit{return\text{-}expression}}$$

Block expressions are currently used only as expanded forms of functions.

Expressions: Binary operators

Left-binary-reduce: An expression with binary operator can be reduced if its left sub-expression can be reduced.

$$\frac{e_1 \;\Rightarrow\; e_1'}{e_1\; \mathit{binary\text{-}operator}\; e_2 \;\Rightarrow\; e_1'\; \mathit{binary\text{-}operator}\; e_2}$$

And-shortcut-false: An expression with binary operator **`&&`** whose left sub-expression is **`false`** can be reduced to **`false`**.

$$\frac{}{\texttt{false \&\&}\; e \;\Rightarrow\; \texttt{false}}$$

And-shortcut-true: An expression with binary operator **`&&`** whose left sub-expression is **`true`** can be reduced to the right sub-expression.

$$\frac{}{\texttt{true \&\&}\; e \;\Rightarrow\; e}$$

Or-shortcut-true: An expression with binary operator `||` whose left sub-expression is `true` can be reduced to `true`.

$$\frac{}{\texttt{true || } e \ \Rightarrow \ \texttt{true}}$$

Or-shortcut-false: An expression with binary operator `||` whose left sub-expression is `false` can be reduced to the right sub-expression.

$$\frac{}{\texttt{false || } e \ \Rightarrow \ e}$$

Right-binary-reduce: An expression with binary operator can be reduced if its left sub-expression is a value and its right sub-expression can be reduced.

$$\frac{e_2 \ \Rightarrow \ e_2', \text{and } \textit{binary-operator}\text{is not } \texttt{\&\&} \text{ or } \texttt{||}}{v \ \textit{binary-operator} \ e_2 \ \Rightarrow \ v \ \textit{binary-operator} \ e_2'}$$

Prim-binary-reduce: An expression with binary operator can be reduced if its left and right sub-expressions are values and the corresponding function is defined for those values.

$$\frac{v \text{ is result of } v_1 \ \textit{binary-operator} \ v_2}{v_1 \ \textit{binary-operator} \ v_2 \ \Rightarrow \ v}$$

Expressions: Unary operators

Unary-reduce: An expression with unary operator can be reduced if its sub-expression can be reduced.

$$\frac{e \ \Rightarrow \ e'}{\textit{unary-operator} \ e \ \Rightarrow \ \textit{unary-operator} \ e'}$$

Prim-unary-reduce: An expression with unary operator can be reduced if its sub-expression is a value and the corresponding function is defined for that value.

$$\frac{v' \text{ is result of } \textit{unary-operator} \ v}{\textit{unary-operator} \ v \ \Rightarrow \ v'}$$

Expressions: conditionals

Conditional-predicate-reduce: A conditional expression can be reduced, if its predicate can be reduced.

$$\frac{e_1 \ \Rightarrow \ e_1'}{e_1 \ \texttt{?} \ e_2 \ \texttt{:} \ e_3 \ \Rightarrow \ e_1' \ \texttt{?} \ e_2 \ \texttt{:} \ e_3}$$

Conditional-true-reduce: A conditional expression whose predicate is the value `true` can be reduced to its consequent expression.

$$\frac{}{\texttt{true ?} \ e_1 \ \texttt{:} \ e_2 \ \Rightarrow \ e_1}$$

Conditional-false-reduce: A conditional expression whose predicate is the value `false` can be reduced to its alternative expression.

$$\frac{}{\texttt{false ?} \ e_1 \ \texttt{:} \ e_2 \ \Rightarrow \ e_2}$$

Expressions: function application

Application-functor-reduce: A function application can be reduced if its functor expression can be reduced.

$$\frac{e \;\Rightarrow\; e'}{e\ \texttt{(}\ \textit{expressions}\ \texttt{)} \;\Rightarrow\; e'\ \texttt{(}\ \textit{expressions}\ \texttt{)}}$$

Application-argument-reduce: A function application can be reduced if one of its argument expressions can be reduced and all preceding arguments are values.

$$\frac{e \;\Rightarrow\; e'}{v\ \texttt{(}\ v_1 \ldots v_i\ e \ldots\ \texttt{)} \;\Rightarrow\; v\ \texttt{(}\ v_1 \ldots v_i\ e' \ldots\ \texttt{)}}$$

Function-declaration-application-reduce: The application of a function declaration can be reduced, if all arguments are values.

$$\frac{f = \texttt{function}\ n\ \texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \textit{block}}{f\ \texttt{(}\ v_1 \ldots v_n\ \texttt{)} \;\Rightarrow\; \textit{block}[x_1 := v_1] \ldots [x_n := v_n][n := f]}$$

Function-definition-application-reduce: The application of a function definition can be reduced, if all arguments are values.

$$\frac{f = \texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \texttt{=>}\ b, \text{ where } b \text{ is an expression or block}}{f\ \texttt{(}\ v_1 \ldots v_n\ \texttt{)} \;\Rightarrow\; b[x_1 := v_1] \ldots [x_n := v_n]}$$

Substitution

Identifier: An identifier with the same name as x is substituted with e_x.

$$\frac{}{x[x := e_x] \ = \ e_x}$$

$$\frac{\textit{name} \ \neq \ x}{\textit{name}[x := e_x] \ = \ \textit{name}}$$

Expression statement: All occurrences of x in e are substituted with e_x.

$$\frac{}{e\texttt{;}\,[x := e_x] \ = \ e\,[x := e_x]\texttt{;}}$$

Binary expression: All occurrences of x in the operands are substituted with e_x.

$$\frac{}{(e_1 \ \textit{binary-operator} \ e_2)[x := e_x] \ = \ e_1[x := e_x] \ \textit{binary-operator} \ e_2[x := e_x]}$$

Unary expression: All occurrences of x in the operand are substituted with e_x.

$$\frac{}{(\textit{unary-operator} \ e)[x := e_x] \ = \ \textit{unary-operator} \ e[x := e_x]}$$

Conditional expression: All occurrences of x in the operands are substituted with e_x.

$$\frac{}{(e_1 \ \texttt{?} \ e_2 \ \texttt{:} \ e_3)[x := e_x] \ = \ e_1[x := e_x] \ \texttt{?} \ e_2[x := e_x] \ \texttt{:} \ e_3[x := e_x]}$$

Logical expression: All occurrences of x in the operands are substituted with e_x.

$$\frac{}{(e_1 \ \texttt{||} \ e_2)[x := e_x] \ = \ e_1[x := e_x] \ \texttt{||} \ e_2[x := e_x]}$$

$$\frac{}{(e_1 \ \texttt{\&\&} \ e_2)[x := e_x] \ = \ e_1[x := e_x] \ \texttt{\&\&} \ e_2[x := e_x]}$$

Call expression: All occurrences of x in the arguments and the function expression of the application e are substituted with e_x.

$$\frac{}{(e \ \texttt{(} \ x_1 \ldots x_n \ \texttt{)})[x := e_x] \ = \ e[x := e_x] \ \texttt{(} \ x_1[x := e_x] \ldots x_n[x := e_x] \ \texttt{)}}$$

Function declaration: All occurrences of x in the body of a function are substituted with e_x under given circumstances.

1. Function declaration where x has the same name as a parameter.

$$\frac{\exists\, i \in \{1, \cdots, n\} \text{ s.t. } x \ = \ x_i}{(\texttt{function} \ \textit{name} \ \texttt{(} \ x_1 \ldots x_n \ \texttt{)} \ \textit{block})[x := e_x] \ = \ \texttt{function} \ \textit{name} \ \texttt{(} \ x_1 \ldots x_n \ \texttt{)} \ \textit{block}}$$

2. Function declaration where x does not have the same name as a parameter.

(a) No parameter of the function occurs free in e_x.

$$\frac{\forall\, i \in \{1, \cdots, n\} \text{ s.t. } x \; \neq \; x_i, \;\; \forall\, j \in \{1, \cdots, n\} \text{ s.t. } x_j \text{ does not occur free in } e_x}{\begin{array}{c}(\texttt{function}\ \textit{name}\ \texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \textit{block})[x := e_x] \\ = \\ \texttt{function}\ \textit{name}\ \texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \textit{block}[x := e_x]\end{array}}$$

(b) A parameter of the function occurs free in e_x.

$$\frac{\forall\, i \in \{1, \cdots, n\} \text{ s.t. } x \; \neq \; x_i, \;\; \exists\, j \in \{1, \cdots, n\} \text{ s.t. } x_j \text{ occurs free in } e_x}{\begin{array}{c}(\texttt{function}\ \textit{name}\ \texttt{(}\ x_1 \ldots x_j \ldots x_n\ \texttt{)}\ \textit{block})[x := e_x] \\ = \\ (\texttt{function}\ \textit{name}\ \texttt{(}\ x_1 \ldots y \ldots x_n\ \texttt{)}\ \textit{block}[x_j := y])[x := e_x]\end{array}}$$

Substitution is applied to the whole expression again as to recursively detect and rename all parameters of the function declaration that clash with variables that occur free in e_x, at which point (i) takes place. Note that the name y is not declared in, nor occurs in *block* and e_x.

Lambda expression: All occurrences of x in the body of a lambda expression are substituted with e_x under given circumstances.

1. Lambda expression where x has the same name as a parameter.

$$\frac{\exists\, i \in \{1, \cdots, n\} \text{ s.t. } x \; = \; x_i}{(\texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block})[x := e_x] \;\; = \;\; \texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block}}$$

2. Lambda expression where x does not have the same name as a parameter.

(a) No parameter of the lambda expression occurs free in e_x.

$$\frac{\forall\, i \in \{1, \cdots, n\} \text{ s.t. } x \; \neq \; x_i, \;\; \forall\, j \in \{1, \cdots, n\} \text{ s.t. } x_j \text{ does not occur free in } e_x}{(\texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block})[x := e_x] \;\; = \;\; \texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block}[x := e_x]}$$

(b) A parameter of the lambda expression occurs free in e_x.

$$\frac{\forall\, i \in \{1, \cdots, n\} \text{ s.t. } x \; \neq \; x_i, \;\; \exists\, j \in \{1, \cdots, n\} \text{ s.t. } x_j \text{ occurs free in } e_x}{(\texttt{(}\ x_1 \ldots x_j \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block})[x := e_x] \;\; = \;\; (\texttt{(}\ x_1 \ldots y \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block}[x_j := y])[x := e_x]}$$

Substitution is applied to the whole expression again as to recursively detect and rename all parameters of the lambda expression that clash with variables that occur free in e_x, at which point (i) takes place. Note that the name y is not declared in, nor occurs in *block* and e_x.

Block expression: All occurrences of x in the statements of a block expression are substituted with e_x under given circumstances.

1. Block expression in which x is declared.

$$\frac{x \text{ is declared in } \textit{block}}{\textit{block}[x := e_x] \ = \ \textit{block}}$$

2. Block expression in which x is not declared.

   (a) No names declared in the block occurs free in e_x.

$$\frac{x \text{ is not declared in } \textit{block}, \ \ \textit{name} \text{ declared in } \textit{block} \text{ does not occur free in } e_x}{\textit{block}[x := e_x] \ = \ [\textit{block}[0][x := e_x], \ \ldots, \ \textit{block}[n][x := e_x]]}$$

   (b) A name declared in the block occurs free in e_x.

$$\frac{x \text{ is not declared in } \textit{block}, \ \ \textit{name} \text{ declared in } \textit{block} \text{ occurs free in } e_x}{\textit{block}[x := e_x] \ = \ [\textit{block}[0][\textit{name} := y], \ \ldots, \ \textit{block}[n][\textit{name} := y]][x := e_x]}$$

   Substitution is applied to the whole expression again as to recursively detect and rename all declared names of the block expression that clash with variables that occur free in e_x, at which point (i) takes place. Note that the name y is not declared in, nor occurs in *block* and e_x.

Variable declaration: All occurrences of x in the declarators of a variable declaration are substituted with e_x.

$$\frac{}{\textit{declarations}[x := e_x] \ = \ [\textit{declarations}[0][x := e_x] \ldots \textit{declarations}[n][x := e_x]]}$$

Return statement: All occurrences of x in the expression that is to be returned are substituted with e_x.

$$\frac{}{(\texttt{return } e\texttt{;})[x := e_x] \ = \ \texttt{return } e[x := e_x]\texttt{;}}$$

Conditional statement: All occurrences of x in the condition, consequent, and alternative expressions of a conditional statement are substituted with e_x.

$$\frac{}{(\texttt{if (} \ e \ \texttt{)} \ \textit{block} \ \texttt{else} \ \textit{block})[x := e_x] \ = \ \texttt{if (} \ e[x := e_x] \ \texttt{)} \ \textit{block}[x := e_x] \ \texttt{else} \ \textit{block}[x := e_x]}$$

Array expression: All occurrences of x in the elements of an array are substituted with e_x.

$$\frac{}{[x_1, \ \ldots, \ x_n][x := e_x] \ = \ [x_1[x := e_x], \ \ldots, \ x_n[x := e_x]]}$$

Free names

Let $\triangleright$ be the relation that defines the set of free names of a given Source expression; the symbols p_1 and p_2 shall henceforth refer to unary and binary operations, respectively. That is, p_1 ranges over $\{\texttt{!}\}$ and p_2 ranges over $\{\texttt{||}, \texttt{\&\&}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{===}, \texttt{>}, \texttt{<}\}$.

Identifier:

$$\frac{}{x \ \triangleright \ \{x\}}$$

$$\frac{}{\textit{name} \ \triangleright \ \varnothing}$$

Boolean:

$$\frac{}{\texttt{true} \ \triangleright \ \varnothing}$$

$$\frac{}{\texttt{false} \ \triangleright \ \varnothing}$$

Expression statement:

$$\frac{e \ \triangleright \ S}{e\texttt{;} \ \triangleright \ S}$$

Unary expression:

$$\frac{e \ \triangleright \ S}{p_1(e) \ \triangleright \ S}$$

Binary expression:

$$\frac{e_1 \ \triangleright \ S_1, \ \ e_2 \ \triangleright \ S_2}{p_1(e_1, e_2) \ \triangleright \ S_1 \cup S_2}$$

Conditional expression:

$$\frac{e_1 \ \triangleright \ S_1, \ \ e_2 \ \triangleright \ S_2, \ \ e_3 \ \triangleright \ S_3}{e_1 \ \texttt{?} \ e_2 \ \texttt{:} \ e_3 \ \triangleright \ S_1 \cup S_2 \cup S_3}$$

Call expression:

$$\frac{e \ \triangleright \ S, \ \ e_k \ \triangleright \ T_k}{e(e_1, \ldots, e_n) \ \triangleright \ S \cup T_1 \cup \ldots \cup T_n}$$

Function declaration:

$$\frac{\textit{block} \ \triangleright \ S}{\texttt{function} \ \textit{name} \ \texttt{(} \ x_1 \ldots x_n \ \texttt{)} \ \textit{block} \ \triangleright \ S - \{x_1, \ldots, x_n\}}$$

Lambda expression:

$$\frac{\textit{block} \ \triangleright \ S}{\texttt{(} \ x_1 \ldots x_n \ \texttt{)} \ \texttt{=>} \ \textit{block} \ \triangleright \ S - \{x_1, \ldots, x_n\}}$$

Block expression:

$$\frac{\textit{block}[k] \ \triangleright \ S_k, \ \ T \text{ contains all names declared in } \textit{block}}{\textit{block} \ \triangleright \ (S_1 \cup \ldots \cup S_n) - T}$$

Constant declaration:

$$\frac{e \ \triangleright \ S}{\texttt{const } \textit{name} \texttt{ = } e\texttt{;} \ \triangleright \ S}$$

Return statement:

$$\frac{e \ \triangleright \ S}{\texttt{return } e\texttt{;} \ \triangleright \ S}$$

Conditional statement:

$$\frac{e \ \triangleright \ S, \ \ \textit{block}_1 \ \triangleright \ T_1, \ \ \textit{block}_2 \ \triangleright \ T_2}{\texttt{if (} e \texttt{) } \textit{block}_1 \texttt{ else } \textit{block}_2 \ \triangleright \ S \cup T_1 \cup T_2}$$