SimpleScala Formalism

1 Syntax

$$x \in \textit{Variable} \qquad \textit{str} \in \textit{String} \qquad b \in \textit{Boolean} \qquad i \in \mathbb{Z} \qquad n \in \mathbb{N}$$
$$\textit{fn} \in \textit{FunctionName} \qquad \textit{cn} \in \textit{ConstructorName} \qquad \textit{un} \in \textit{UserDefinedTypeName}$$
$$T \in \textit{TypeVariable}$$

$$\begin{aligned}
\tau \in \textit{Type} &::= \textbf{string} \mid \textbf{boolean} \mid \textbf{integer} \mid \textbf{unitType} \mid \tau_1 \Rightarrow \tau_2 \mid (\ \vec{\tau}\ ) \mid \textit{un}[\vec{\tau}] \mid T \mid p \\
p \in \textit{TypePlaceholder} &::= \textbf{placeholder}\ n \\
e \in \textit{Exp} &::= x \mid \textit{str} \mid b \mid i \mid \textbf{unit} \mid e_1 \oplus e_2 \\
&\mid x \Rightarrow e \mid e_1(e_2) \mid \textit{fn}(e) \\
&\mid \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 \\
&\mid \{\overrightarrow{\textit{val}}\ e\} \\
&\mid (\ \vec{e}\ ) \\
&\mid \textit{cn}(e) \mid e\ \textbf{match}\ \{\overrightarrow{\textit{case}}\} \\
\textit{val} \in \textit{Val} &::= \textbf{val}\ x = e \\
\textit{case} \in \textit{Case} &::= \textbf{case}\ \textit{cn}(x) \Rightarrow e \mid \textbf{case}\ (\ \vec{x}\ ) \Rightarrow e \\
\oplus \in \textit{Binop} &::= + \mid - \mid \times \mid \div \mid \wedge \mid \vee \mid < \mid \leq \mid +\!\!+ \\
\textit{tdef} \in \textit{UserDefinedTypeDef} &::= \textbf{algebraic}\ \textit{un}[\vec{T}] = \overrightarrow{\textit{cdef}} \\
\textit{cdef} \in \textit{ConstructorDefinition} &::= \textit{cn}(\tau) \\
\textit{def} \in \textit{Def} &::= \textbf{def}\ \textit{fn}[\vec{T}](x : \tau_1) : \tau_2 = e \\
\textit{prog} \in \textit{Program} &::= \overrightarrow{\textit{tdef}}\ \overrightarrow{\textit{def}}\ e
\end{aligned}$$

2 Typing Rules

2.1 Type Domains

$$\begin{aligned}
\textit{fdefs} \in \textit{NamedFunctionDefs} &= \textit{FunctionName} \to (\overrightarrow{\textit{TypeVariable}} \times \textit{Type} \times \textit{Type}) \\
\textit{tdefs} \in \textit{TypeDefs} &= \textit{UserDefinedTypeName} \to (\overrightarrow{\textit{TypeVariable}} \times (\textit{ConstructorName} \to \textit{Type})) \\
\textit{cdefs} \in \textit{ConstructorDefs} &= \textit{ConstructorName} \to \textit{UserDefinedTypeName} \\
\gamma \in \textit{ConstraintStore} &= \textit{TypePlaceholder} \to \textit{Type} \\
\Gamma \in \textit{TypeEnv} &= \textit{Variable} \to \textit{Type} \\
\varsigma \in \textit{ThreadedTypeState} &= (\textit{ConstraintStore} \times \mathbb{N}) \\
\vdash \in \textit{TypeOf} &= (\textit{ThreadedTypeState} \times \textit{TypeEnv} \times \textit{Exp}) \to (\textit{Type} \times \textit{ThreadedTypeState}) \\
m \in \textit{ConstructorMapping} &= \textit{ConstructorName} \to \textit{Type} \\
\alpha \in \textit{TypeVariableMapping} &= \textit{TypeVariable} \to \textit{TypePlaceholder}
\end{aligned}$$

2.2 Rules

$$\frac{\tau = \Gamma(x)}{\varsigma \cdot \Gamma \vdash x : \tau \cdot \varsigma}\ (\textsc{var}) \qquad \frac{}{\varsigma \cdot \Gamma \vdash str : \mathbf{string} \cdot \varsigma}\ (\textsc{string}) \qquad \frac{}{\varsigma \cdot \Gamma \vdash b : \mathbf{boolean} \cdot \varsigma}\ (\textsc{boolean}) \qquad \frac{}{\varsigma \cdot \Gamma \vdash i : \mathbf{integer} \cdot \varsigma}\ (\mathbb{Z})$$

$$\frac{}{\varsigma \cdot \Gamma \vdash \mathbf{unit} : \mathbf{unitType} \cdot \varsigma}\ (\textsc{unit}) \qquad \frac{\begin{array}{l} \mathtt{binopTypes}(\oplus) = \tau_1 \cdot \tau_2 \cdot \tau_3 \\ \varsigma_1 \cdot \Gamma \vdash e_1 : \tau_{e1} \cdot \varsigma_2 \quad \mathtt{unify}(\varsigma_2, \tau_1, \tau_{e1}) = \varsigma_3 \\ \varsigma_3 \cdot \Gamma \vdash e_2 : \tau_{e2} \cdot \varsigma_4 \quad \mathtt{unify}(\varsigma_4, \tau_2, \tau_{e2}) = \varsigma_f \end{array}}{\varsigma_1 \cdot \Gamma \vdash e_1 \oplus e_2 : \tau_3 \cdot \varsigma_f}\ (\textsc{binop})$$

$$\frac{\begin{array}{l} \tau_x \cdot \varsigma_2 = \mathtt{freshPlaceholder}(\varsigma_1) \\ \Gamma_2 = \Gamma_1[x \mapsto \tau_x] \quad \varsigma_2 \cdot \Gamma_2 \vdash e : \tau_e \cdot \varsigma_f \end{array}}{\varsigma_1 \cdot \Gamma_1 \vdash x \Rightarrow e : \tau_x \Rightarrow \tau_e \cdot \varsigma_f}\ (\textsc{anonfun}) \qquad \frac{\begin{array}{l} \tau_p \cdot \varsigma_2 = \mathtt{freshPlaceholder}(\varsigma_1) \\ \tau_r \cdot \varsigma_3 = \mathtt{freshPlaceholder}(\varsigma_2) \\ \varsigma_3 \cdot \Gamma \vdash e_1 : \tau_{e1} \cdot \varsigma_4 \quad \varsigma_5 = \mathtt{unify}(\varsigma_4, \tau_{e1}, \tau_p \Rightarrow \tau_r) \\ \varsigma_5 \cdot \Gamma \vdash e_2 : \tau_{e2} \cdot \varsigma_6 \quad \varsigma_f = \mathtt{unify}(\varsigma_6, \tau_{e2}, \tau_p) \end{array}}{\varsigma_1 \cdot \Gamma \vdash e_1(e_2) : \tau_r \cdot \varsigma_f}\ (\textsc{anoncall})$$

$$\frac{\tau_p \cdot \tau_r \cdot \varsigma_2 = \mathtt{freshenFn}(\varsigma_1, fn) \quad \varsigma_2 \cdot \Gamma \vdash e : \tau_e \cdot \varsigma_3 \quad \varsigma_f = \mathtt{unify}(\varsigma_3, \tau_e, \tau_p)}{\varsigma_1 \cdot \Gamma \vdash fn(e) : \tau_r \cdot \varsigma_f}\ (\textsc{namedcall})$$

$$\frac{\begin{array}{l} \varsigma_1 \cdot \Gamma \vdash e_1 : \tau_{e1} \cdot \varsigma_2 \quad \varsigma_3 = \mathtt{unify}(\varsigma_2, \tau_{e1}, \mathbf{boolean}) \\ \varsigma_3 \cdot \Gamma \vdash e_2 : \tau_{e2} \cdot \varsigma_4 \quad \varsigma_4 \cdot \Gamma \vdash e_3 : \tau_{e3} \cdot \varsigma_5 \\ \varsigma_f = \mathtt{unify}(\varsigma_5, \tau_{e2}, \tau_{e3}) \end{array}}{\varsigma_1 \cdot \Gamma \vdash \mathbf{if}\ (e_1)\ e_2\ \mathbf{else}\ e_3 : \tau_{e2} \cdot \varsigma_f}\ (\textsc{if}) \qquad \frac{\tau \cdot \varsigma_f = \mathtt{typeofBlock}(\varsigma_1, \Gamma, \overrightarrow{val}, e)}{\varsigma_1 \cdot \Gamma \vdash \{\overrightarrow{val}\ e\} : \tau \cdot \varsigma_f}\ (\textsc{block})$$

$$\frac{|\vec{e}| > 1 \quad \vec{\tau} \cdot \varsigma_f = \mathtt{typeofSeq}(\varsigma_1, \Gamma, \vec{e})}{\varsigma_1 \cdot \Gamma \vdash (\ \vec{e}\ ) : (\ \vec{\tau}\ ) \cdot \varsigma_f}\ (\textsc{tuple}) \qquad \frac{\begin{array}{l} un \cdot \vec{\tau} \cdot \tau_e \cdot \varsigma_2 = \mathtt{freshenCn}(\varsigma_1, cn) \\ \varsigma_2 \cdot \Gamma \vdash e : \tau'_e \cdot \varsigma_3 \quad \varsigma_f = \mathtt{unify}(\varsigma_3, \tau_e, \tau'_e) \end{array}}{\varsigma_1 \cdot \Gamma \vdash cn(e) : un[\vec{\tau}] \cdot \varsigma_f}\ (\textsc{constructor})$$

$$\frac{\begin{array}{l} \varsigma_1 \cdot \Gamma_1 \vdash e_1 : \tau_{e1} \cdot \varsigma_2 \quad \vec{\tau} \cdot \varsigma_3 = \mathtt{tupleTemplate}(\varsigma_2, |\vec{x}|) \\ \varsigma_4 = \mathtt{unify}(\varsigma_3, \tau_{e1}, (\ \vec{\tau}\ )) \quad \Gamma_2 = \mathtt{multiAddEnv}(\Gamma_1, \vec{x}, \vec{\tau}) \\ \varsigma_4 \cdot \Gamma_2 \vdash e_2 : \tau_f \cdot \varsigma_f \end{array}}{\varsigma_1 \cdot \Gamma_1 \vdash e_1\ \mathbf{match}\ \{(\mathbf{case}\ (\ \vec{x}\ ) \Rightarrow e_2) :: \mathbf{nil}\} : \tau_f \cdot \varsigma_f}\ (\textsc{match-tup})$$

$$\frac{\begin{array}{l} \tau_{e1} \cdot m \cdot \varsigma_2 = \mathtt{userTypeTemplate}(\varsigma_1, \overrightarrow{case}) \quad \varsigma_2 \cdot \Gamma \vdash e_1 : \tau'_{e1} \cdot \varsigma_3 \\ \varsigma_4 = \mathtt{unify}(\varsigma_3, \tau_{e1}, \tau'_{e1}) \quad \tau_f \cdot \varsigma_5 = \mathtt{freshPlaceholder}(\varsigma_4) \\ \varsigma_f = \mathtt{typeofCases}(\varsigma_5, \Gamma, \overrightarrow{case}, \tau_f) \end{array}}{\varsigma_1 \cdot \Gamma \vdash e_1\ \mathbf{match}\ \{\overrightarrow{case}\} : \tau_f \cdot \varsigma_f}\ (\textsc{match-constructors})$$

3 Helpers

3.1 `binopTypes`

Returns the expected left parameter type, right parameter type, and return type for the given binary operation.

$\mathtt{binopTypes} \in \mathit{Binop} \rightarrow (\mathit{Type} \times \mathit{Type} \times \mathit{Type})$

$\mathtt{binopTypes}(\oplus) =$

$$\begin{cases} \mathbf{integer} \cdot \mathbf{integer} \cdot \mathbf{integer} & \text{if } \oplus \in \{+, -, \times, \div\} \\ \mathbf{boolean} \cdot \mathbf{boolean} \cdot \mathbf{boolean} & \text{if } \oplus \in \{\wedge, \vee\} \\ \mathbf{integer} \cdot \mathbf{integer} \cdot \mathbf{boolean} & \text{if } \oplus \in \{<, \leq\} \\ \mathbf{string} \cdot \mathbf{string} \cdot \mathbf{string} & \text{if } \oplus = +\!\!+ \end{cases}$$

3.2 `unify`

Performs unification on the given two types over a *ThreadedTypeState*. Note that this is a partial function which is only defined for inputs which will unify. If the function is called with an input under which it's not defined, then the function cannot be applied. With the purpose of the typing rules in mind, unification failure means that typecheking should fail.

$\texttt{unify} \in (\mathit{ThreadedTypeState} \times \mathit{Type} \times \mathit{Type}) \rightarrow \mathit{ThreadedTypeState}$

$\texttt{unify}((\gamma \cdot n), \tau_1, \tau_2) =$

$\quad (\texttt{unifyCS}(\gamma, \tau_1, \tau_2) \cdot n)$

3.3 `unifyCS`

Like `unify`, but it operates directly on *ConstraintStore*. It too is a partial function.

$\texttt{unifyCS} \in (\mathit{ConstraintStore} \times \mathit{Type} \times \mathit{Type}) \rightarrow \mathit{ConstraintStore}$

$\texttt{unifyCS}(\gamma, \tau_1, \tau_2) =$

$\quad \text{let } \tau_1' = \texttt{lookupTypeSingleLevel}(\gamma, \tau_1)$

$\quad \text{let } \tau_2' = \texttt{lookupTypeSingleLevel}(\gamma, \tau_2)$

$$\begin{cases} \texttt{unifyPlaceholderType}(\gamma, p, \tau_2') & \text{if } \tau_1' = p \\ \texttt{unifyPlaceholderType}(\gamma, p, \tau_1') & \text{if } \tau_2' = p \\ \gamma & \text{if } \tau_1' = \tau_2' \wedge (\tau_1' \in \{\textbf{string}, \textbf{boolean}, \textbf{integer}, \textbf{unit}\} \vee \tau_1' = T) \\ \texttt{unifyList}(\gamma, \vec{\tau_3}, \vec{\tau_4}) & \text{if } \tau_1' = (\ \vec{\tau_3}\ ) \wedge \tau_2' = (\ \vec{\tau_4}\ ) \\ \texttt{unifyList}(\gamma, \vec{\tau_3}, \vec{\tau_4}) & \text{if } \tau_1' = un[\vec{\tau_3}] \wedge \tau_1' = un[\vec{\tau_4}] \end{cases}$$

3.4 `lookupTypeSingleLevel`

Looks up a type in the constraint store. If the type is a non-placeholder type, it simply returns it as-is.

$\texttt{lookupTypeSingleLevel} \in (\mathit{ConstraintStore} \times \mathit{Type}) \rightarrow \mathit{Type}$

$\texttt{lookupTypeSingleLevel}(\gamma, \tau) =$

$$\begin{cases} \texttt{lookup}(\gamma, p) & \text{if } \tau = p \\ \tau & \text{otherwise} \end{cases}$$

3.5 `lookup`

$\texttt{lookup} \in (\mathit{ConstraintStore} \times \mathit{TypePlacehelper}) \rightarrow \mathit{Type}$

$\texttt{lookup}(\gamma, p) =$

$$\begin{cases} \texttt{lookup}(\gamma, p') & \text{if } p \in \texttt{keys}(\gamma) \wedge \gamma(p) = p' \\ \tau & \text{if } p \in \texttt{keys}(\gamma) \wedge \gamma(p) = \tau \wedge \tau \neq p' \\ p & \text{otherwise} \end{cases}$$

3.6 `unifyPlaceholderType`

Unifies a placeholder with a given type. This is a partial function, with the same caveat as `unify`. This performs the occurs check.

$\texttt{unifyPlaceholderType} \in (\mathit{ConstraintStore} \times \mathit{TypePlaceholder} \times \mathit{Type}) \rightarrow \mathit{ConstraintStore}$

$\texttt{unifyPlaceholderType}(\gamma, p, \tau) =$

$$\begin{cases} \gamma & \text{if } p = \tau \\ \gamma[p \mapsto \tau] & \text{if } \neg\texttt{typeContains}(\gamma, \tau, p) \end{cases}$$

3.7 typeContains

Determines if the given type contains the given placeholder. This is used as part of the occurs check in `unifyPlaceholderType`.

$\texttt{typeContains} \in (\mathit{ConstraintStore} \times \mathit{Type} \times \mathit{TypePlaceholder}) \to \mathit{Boolean}$

$\texttt{typeContains}(\gamma, \tau, p) =$

$$\begin{cases} \textbf{false} & \text{if } \tau \in \{\textbf{string}, \textbf{boolean}, \textbf{integer}, \textbf{unit}\} \lor \tau = T \\ \texttt{typeContains}(\gamma, \tau_1, p) \lor \texttt{typeContains}(\gamma, \tau_2, p) & \text{if } \tau = \tau_1 \Rightarrow \tau_2 \\ \texttt{typesContains}(\gamma, \vec{\tau_1}, p) & \text{if } \tau = (\ \vec{\tau_1}\ ) \\ \texttt{typesContains}(\gamma, \vec{\tau_1}, p) & \text{if } \tau = un[\vec{\tau_1}] \\ p = p' & \text{if } \tau = p \land \texttt{lookup}(\gamma, p) = p' \\ \texttt{typeContains}(\gamma, \tau', p) & \text{if } \tau = p \land \texttt{lookup}(\gamma, p) = \tau' \land \tau' \neq p' \end{cases}$$

3.8 typesContains

Returns true if any of the given types contains the given placeholder. This is used as part of the occurs check in `unifyPlaceholderType` (called indirectly through `typeContains`).

$\texttt{typesContains} \in (\mathit{ConstraintStore} \times \overrightarrow{\mathit{Type}} \times \mathit{TypePlaceholder}) \to \mathit{Boolean}$

$\texttt{typesContains}(\gamma, \vec{\tau}, p) =$

$$\begin{cases} \textbf{false} & \text{if } \vec{\tau} = \textbf{nil} \\ \texttt{typeContains}(\gamma, \tau_1, p) \lor \texttt{typesContains}(\gamma, \vec{\tau_2}, p) & \text{if } \vec{\tau} = \tau_1 :: \vec{\tau_2} \end{cases}$$

3.9 unifyList

Unifies the two given lists of types. This is a partial function as the same stipulation as `unify`.

$\texttt{unifyList} \in (\mathit{ConstraintStore} \times \overrightarrow{\mathit{Type}} \times \overrightarrow{\mathit{Type}}) \to \mathit{ConstraintStore}$

$\texttt{unifyList}(\gamma, \vec{\tau_1}, \vec{\tau_2}) =$

$$\begin{cases} \gamma & \text{if } \vec{\tau_1} = \textbf{nil} \land \vec{\tau_2} = \textbf{nil} \\ \texttt{unifyList}(\gamma', \vec{\tau_1}', \vec{\tau_2}') & \text{if } \vec{\tau_1} = \tau_a :: \vec{\tau_1}' \land \vec{\tau_2} = \tau_b :: \vec{\tau_2}' \land \gamma' = \texttt{unifyCS}(\gamma, \tau_a, \tau_b) \end{cases}$$

3.10 freshPlaceholder

Returns a fresh placeholder type.

$\texttt{freshPlaceholder} \in \mathit{ThreadedTypeState} \to (\mathit{TypePlaceholder} \times \mathit{ThreadedTypeState})$

$\texttt{freshPlaceholder}((\gamma, n)) =$

$$(\textbf{placeholder}(n) \cdot (\gamma \cdot n + 1))$$

3.11 freshenFn

Returns the parameter type and the return type for the given function. This is not necessarily trivial, because it may be necessary to introduce fresh type placeholders. This is a partial function; it is only defined if the given function name is in *fdefs*. This has the same sort of stipulation as `unify`.

$\texttt{freshenFn} \in (\mathit{ThreadedTypeState} \times \mathit{FunctionName}) \to (\mathit{Type} \times \mathit{Type} \times \mathit{ThreadedTypeState})$

$\texttt{freshenFn}(\varsigma_1, \mathit{fn}) =$

$\quad \text{let } (\vec{T} \cdot \tau_p \cdot \tau_r) = \mathit{fdefs}(\mathit{fn})$

$\quad \text{let } (\vec{p} \cdot \tau'_p :: \tau'_r :: \textsf{nil} \cdot \varsigma_2) = \texttt{freshen}(\varsigma_1, \vec{T}, \tau_p :: \tau_r :: \textsf{nil})$

$\quad (\tau'_p \cdot \tau'_r \cdot \varsigma_2)$

3.12 freshen

Takes a listing of type variables along with a listing of types for which those type variables are in scope. Replaces the type variables in the input types with fresh placeholders corresponding to the type variables.

$\texttt{freshen} \in (\mathit{ThreadedTypeState} \times \overrightarrow{\mathit{TypeVariable}} \times \overrightarrow{\mathit{Type}}) \to (\overrightarrow{\mathit{TypePlaceholder}} \times \overrightarrow{\mathit{Type}} \times \mathit{ThreadedTypeState})$

$\texttt{freshen}(\varsigma_1, \vec{T}, \vec{\tau}) =$

$\quad \text{let } (\vec{p} \cdot \varsigma_2) = \texttt{typeListTemplate}(|\vec{T}|)$

$\quad \text{let } \alpha = \texttt{pairsToMap}(\texttt{zip}(\vec{T}, \vec{p}))$

$\quad (\vec{p} \cdot \texttt{makeFreshList}(\alpha, \vec{\tau}) \cdot \varsigma_2)$