Prof. Dr. Peter Thiemann
Matthias Keil Winter Semester 2016/2017

Compiler Construction

http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2016ws/

Exercise Sheet 3

1 Type-checking MiniJava - Solution

MiniJava is a strongly typed language with explicit types. This means that the type of every variable and every expression is known at compile-time. Detecting type-errors early (i.e. at compile time) supports programmers in writing (fail-)safe code.
MiniJava adheres for the most part to the type rules of Java. It provides two basic types for booleans and integers, and two reference types for integer arrays and objects. Types are defined by

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{int}[] \mid C$$

for all $C \in dom(CT)$ where the class table CT is a mapping from class names to class declarations. There exists a subtype relation $\prec$ between the types. This relation is reflexive and transitive (but not symmetric). For simplicity, we identify the class name with the class type here.

$$\tau \prec \tau \qquad \frac{\tau_1 \prec \tau_2 \qquad \tau_2 \prec \tau_3}{\tau_1 \prec \tau_3} \qquad \frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \ldots\ \}}{C \prec D}$$

Remark: One major difference between Java and MiniJava is that MiniJava does not specify `Object` to be the superclass of all other classes.

Type judgments define whether an expression, a statement, etc. is *well-typed*. For expressions, we use the type judgment $\Gamma \vdash_e e : \tau$ to say that an expression e is well-typed in Γ with type τ. The typing context (or type environment) Γ contains all variables with their types which are defined when typing the expression.
For the arithmetic and boolean expressions the type rules are straight-forward, for example:

$$\frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 < e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash_e e : \texttt{bool}}{\Gamma \vdash_e !e : \texttt{bool}}$$

The rules for $-, *$ and && are defined analogously. For constants, the type rules are trivial, as is the one for object allocation:

$$\frac{}{\Gamma \vdash_e \texttt{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash_e \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash_e i : \texttt{int}} \qquad \frac{}{\Gamma \vdash_e \texttt{new}\, C() : C}$$

The type of a variable can be determined by looking it up in the type environment:

$$\frac{id : \tau \in \Gamma}{\Gamma \vdash_e id : \tau}$$

A bit more involved is the type rule for method invocation:

$$\frac{\begin{array}{c}\Gamma \vdash_e e : C \\ paramsT(m, C) = (\tau_1, \ldots, \tau_n) \qquad returnT(m, C) = \tau \qquad \forall e_i : \Gamma \vdash_e e_i : \sigma_i, \quad \sigma_i \prec \tau_i\end{array}}{\Gamma \vdash_e e.m(e_1, \ldots, e_n) : \tau}$$

Here, $paramsT(m, C)$ denotes the types of the formal parameters of method m in class C. $returnT(m, C)$ denotes the return type of this method.
Because statements don't have a type, we use a different judgment $\Gamma \vdash_s s$ to denote well-typed statements. The corresponding type rules then have the following form:

$$\frac{\Gamma \vdash_e e : \mathtt{int}}{\Gamma \vdash_s \mathtt{System.out.println}(e);} \qquad \frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_2 \prec \tau_1}{\Gamma \vdash_s e_1 = e_2;}$$

$$\frac{\Gamma \vdash_e e : \mathtt{bool} \qquad \Gamma \vdash_s s}{\Gamma \vdash_s \mathtt{while}(e)\ \mathtt{do}\ s} \qquad \frac{\Gamma \vdash_e e : \mathtt{bool} \qquad \Gamma \vdash_s s_1 \qquad \Gamma \vdash_s s_2}{\Gamma \vdash_s \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2} \qquad \frac{\forall s_i : \Gamma \vdash_s s_i}{\Gamma \vdash_s \{s_1 \ldots s_n\}}$$

Further, a class is well-typed if all its methods are well-typed. A method is well-typed if its statement list is well-typed and the type of its return expression is a subtype of its return type. When type-checking a class or method, `this` must be entered with the correct type in Γ, as well as all fields of the class and the respective formal parameters and local variables of the method.

Solution: Type rules

To increase readability, we use $\overline{x}$ to denote the sequence $x_1, \ldots, x_n$. Similarly, $\overline{f : \tau}$ stands for $f_1 : \tau_1, \ldots, f_n : \tau_n$, and so on.

Arrays

$$\frac{\Gamma \vdash_e e : \mathtt{int}}{\Gamma \vdash_e \mathtt{new\ int}[e] : \mathtt{int}[]} \qquad \frac{\Gamma \vdash_e e : \mathtt{int}[]}{\Gamma \vdash_e e.\mathtt{length} : \mathtt{int}} \qquad \frac{\Gamma \vdash_e e_1 : \mathtt{int}[] \qquad \Gamma \vdash_e e_2 : \mathtt{int}}{\Gamma \vdash_e e_1[e_2] : \mathtt{int}}$$

Class typing

$$\frac{\overline{x : \sigma} \vdash_s s_i}{\vdash_c \mathtt{class}\ C\ \{\mathtt{public\ static\ void\ main}\ (\mathtt{String}[]\ p)\{\ \overline{\sigma\, x};\ \overline{s};\ \}\}} \qquad \frac{\overline{f : \tau}, \mathtt{this} : C \vdash_m m_i}{\vdash_c \mathtt{class}\ C\ \{\overline{\tau\, f};\ \overline{m}\}}$$

$$\frac{fields(C) = \overline{g : \sigma} \qquad \overline{g : \sigma}, \mathtt{this} : C \vdash_m m_i \qquad override(m_i, C, D)}{\vdash_c \mathtt{class}\ C\ \mathtt{extends}\ D\ \{\overline{\tau\, f};\ \overline{m}\}}$$

Method typing

$$\frac{\Gamma' = \Gamma, \overline{p : \tau}, \overline{x : \sigma} \qquad \Gamma' \vdash_s s_i \qquad \Gamma' \vdash_e e : \tau' \qquad \tau' \prec \tau}{\Gamma \vdash_m \texttt{public}\ \tau\ m\ (\overline{\tau\, p})\{\ \overline{\sigma\, x};\ \overline{s};\ \texttt{return}\ e; \}}$$

Auxiliary definitions

$$\frac{CT(C) = \texttt{class}\ C\ \{\overline{\tau\, f};\ \overline{m}\,\}}{fields(C) = \overline{f : \tau}}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, f};\ \overline{m}\,\} \qquad fields(D) = \overline{g : \sigma}}{fields(C) = \overline{g : \sigma}, \overline{f : \tau}}$$

$$\frac{\begin{array}{c} def(m, D) \text{ implies} \\ returnT(m, C) \prec returnT(m, D), \quad paramsT(m, C) = paramsT(m, D) \end{array}}{override(m, C, D)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \{\overline{\tau\, f};\ \overline{m}\,\} \qquad m \text{ defined in } \overline{m}}{def(m, C)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, f};\ \overline{m}\,\} \qquad m \text{ defined in } \overline{m}}{def(m, C)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, f};\ \overline{m}\,\} \qquad def(m, D)}{def(m, C)}$$

Supplementary definitions (optional)

$$\frac{def(m_i, C) \qquad m_i = \texttt{public}\ \tau\ m\ (\overline{\tau\, p})\ \{\overline{\sigma\, x};\ \overline{s};\ \texttt{return}\ e; \}}{paramsT(m, C) = (\overline{\tau})}$$

$$\frac{def(m_i, C) \qquad m_i = \texttt{public}\ \tau\ m\ (\overline{\tau\, p})\ \{\overline{\sigma\, x};\ \overline{s};\ \texttt{return}\ e; \}}{returnT(m, C) = \tau}$$