# The MiniJava Type System

Jens Palsberg

October 2014

## 1 What is MiniJava?

MiniJava is a subset of Java. The meaning of a MiniJava program is given by its meaning as a Java program. Overloading is not allowed in MiniJava. The MiniJava statement `System.out.println(...);` can only print integers. The MiniJava expression `e.length` only applies to expressions of type `int[]`.

We will now specify MiniJava's syntax and type system. A MiniJava program will type check with the MiniJava type system (as specified below) if and only if it will type check with the Java type system (as specified in the Java Language Specification).

## 2 Syntax

The grammar below uses the following metanotation:

- Nonterminal symbols are words written in *this font*.
- Terminal symbols are written in `this font`, except ⟨IDENTIFIER⟩ and ⟨INTEGER_LITERAL⟩.
- A production is of the form $lhs ::= rhs$, where $lhs$ is a nonterminal symbol and $rhs$ is a sequence of nonterminal and terminal symbols, with choices separated by |, and some times using "..." to denote a possibly empty list.
- We will use superscripts and subscripts to distinguish metavariables.

| | | | | |
|---|---|---|---|---|
| (*Goal*) | $g$ | ::= | $mc\ d_1\ \ldots\ d_n$ | main class followed by all other classes in program |
| (*MainClass*) | $mc$ | ::= | `class` $id$ `{ public static void main (String [] ` $id^S$`){` $t_1\ id_1$`; ...;` $t_r\ id_r$`;` $s_1 \ldots\ s_q$`}}` | |
| (*TypeDeclaration*) | $d$ | ::= | `class` $id$ `{` $t_1\ id_1$`; ...;` $t_f\ id_f$`;` $m_1\ \ldots\ m_k$ `}` | all fields declared before all methods |
| | | \| | `class` $id$ `extends` $id^P$ `{` $t_1\ id_1$`; ...;` $t_f\ id_f$`;` $m_1\ \ldots\ m_k$ `}` | |
| (*MethodDeclaration*) | $m$ | ::= | `public` $t\ id^M$ `(`$t_1^F\ id_1^F$`, ...,` $t_n^F\ id_n^F$`) {` $t_1\ id_1$`; ...;` $t_r\ id_r$`;` $s_1\ \ldots\ s_q$ `return` $e$`; }` | all local variables are declared at the beginning of the method body |
| (*Type*) | $t$ | ::= | `int[]` \| `boolean` \| `int` \| $id$ | only supports ints, int arrays, bools, and custom classes |
| (*Statement*) | $s$ | ::= | `{` $s_1\ \ldots\ s_q$ `}` \| $id$ `=` $e$`;` \| $id$ `[` $e_1$ `]` `=` $e_2$`;` | |
| | | \| | `if (` $e$ `)` $s_1$ `else` $s_2$ \| `while (` $e$ `)` $s$ \| `System.out.println(` $e$ `);` | |
| (*Expression*) | $e$ | ::= | $p_1$ `&&` $p_2$ \| $p_1$ `<` $p_2$ \| $p_1$ `+` $p_2$ \| $p_1$ `-` $p_2$ \| $p_1$ `*` $p_2$ \| $p_1$ `[` $p_2$ `]` | |
| | | \| | $p$ `.length` \| $p$ `.`$id$ `(`$e_1$`, ...,` $e_n$`)` \| $p$ | |
| (*PrimaryExpression*) | $p$ | ::= | $c$ \| `true` \| `false` \| $id$ \| `this` \| `new int[`$e$`]` \| `new` $id$`()` \| `!`$e$ \| `(`$e$`)` | primary expressions dont require runtime evaluation to know how to handle |
| (*IntegerLiteral*) | $c$ | ::= | ⟨INTEGER_LITERAL⟩ | |
| (*Identifier*) | $id$ | ::= | ⟨IDENTIFIER⟩ | |

# 3 Notation for Rules

We will use the following notation:

$$\frac{hypothesis_1 \qquad hypothesis_2 \quad \ldots \quad hypothesis_n}{conclusion}$$

This is a *rule* that says that if we can derive all of $hypothesis_1$, $hypothesis_2$, ..., $hypothesis_n$, then we can also derive *conclusion*.

A special case arises when $n = 0$: we can write this case as:

$$\frac{}{conclusion}$$

or we can even omit the horizontal bar and write:

$$conclusion$$

We can say that this case is a rule with no hypotheses, or we can call it an *axiom*.

A *derivation* happens when we begin with one or more axioms, then perhaps apply some rules, and finally arrive at a conclusion. Notice that we can organize a derivation as a tree that has the axioms as leaves and the conclusion as the root. We can refer to such as tree as a *derivation tree*.

# 4 Subtyping

We now define a *subtype* relation on class names. We use $\leq$ to denote the subtype relation. We $t_1 \leq t_2$, we say that $t_1$ is a subtype of $t_2$. We define that $\leq$ is reflexive and transitive, and generated by the "extends" relation among classes.

all classes are subtypes of themselves

$$t \leq t \tag{1}$$

A extends B; B extends C; A extends C;

$$\frac{t_1 \leq t_2 \qquad t_2 \leq t_3}{t_1 \leq t_3} \tag{2}$$

$$\frac{\texttt{class } C \texttt{ extends } D \texttt{ ...} \text{ is in the program}}{C \leq D} \tag{3}$$

# 5 Type Environments

A type environment is a finite mapping from identifiers to types. We use $A$ to range over type environments. We use $dom(A)$ to denote the domain of $A$. If $id_1, \ldots, id_r$ are pairwise distinct identifiers, then the notation $[id_1 : t_1, \ldots, id_r : t_r]$ denotes a type environment that maps $id_i$ to $t_i$, for $i \in 1..r$. If $A_1, A_2$ are type environments, then $A_1 \cdot A_2$ is a type environment defined in the following way:

$$(A_1 \cdot A_2)(id) = \begin{cases} A_2(id) & \text{if } id \in dom(A_2) \\ A_1(id) & \text{otherwise} \end{cases} \tag{4}$$

Notice that $A_2$ takes precedence over $A_1$.

# 6 Helper Functions

## 6.1 The *classname* Helper Function

The function *classname* returns the name of a class. The definition of *classname* is:

$$classname(\texttt{class}\ id\ \{\ \texttt{public static void main (String [] }\ id^S\texttt{) }\ \{\ \ldots\ \}\}) = id \tag{5}$$

$$classname(\texttt{class}\ id\ \{\ t_1\ id_1\texttt{;}\ \ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ \ldots\ \ m_k\ \}) = id \tag{6}$$

$$classname(\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1\texttt{;}\ \ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ \ldots\ \ m_k\ \}) = id \tag{7}$$

## 6.2 The *linkset* Helper Function

The function *linkset* returns the connection between a class and its superclass (represented as a singleton set with one pair of class names), or the emptyset if a class has no superclass. The definition of *linkset* is:

$$linkset(\texttt{class}\ id\ \{\ \texttt{public static void main (String [] }\ id^S\texttt{) }\ \{\ \ldots\ \}\}) = \emptyset \tag{8}$$

$$linkset(\texttt{class}\ id\ \{\ t_1\ id_1\texttt{;}\ \ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ \ldots\ \ m_k\ \}) = \emptyset \tag{9}$$

$$linkset(\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1\texttt{;}\ \ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ \ldots\ \ m_k\ \}) = \{(id, id^P)\} \tag{10}$$

## 6.3 The *methodname* Helper Function

The function *methodname* returns the name of a method definition. The definition of *methodname* is:

$$methodname(\texttt{public}\ t\ \ id^M\ \ \texttt{(...)}\ \ \ldots) = id^M \tag{11}$$

## 6.4 The *distinct* Helper Function

The *distinct* function checks that the identifiers in a list are pairwise distinct.

$$\frac{\forall i \in 1..n : \forall j \in 1..n : id_i = id_j \Rightarrow i = j}{distinct(id_1, \ldots, id_n)} \tag{12}$$

## 6.5 The *acyclic* Helper Function

The *acyclic* function checks that a set of pairs contains no cycles.

no subclass cycles

$$\frac{\neg\ [\ \exists j_1, \ldots, j_h \in 1..n : (\forall i \in 1..(h-1) : id^P_{j_i} = id_{j_{i+1}})\ \wedge\ (id^P_{j_h} = id_{j_1})\ ]}{acyclic(\{\ (id_1, id^P_1), \ldots, (id_n, id^P_n)\ \})} \tag{13}$$

## 6.6 The *fields* Helper Function

We use the notation $fields(C)$ to denote a type environment constructed from the fields of $C$ and the fields of the superclasses of $C$. The fields in $C$ take precedence over the fields in the superclasses of $C$. The definition of *fields* is:

fields(id) returns a list of all fields in the 'id' class and their associated types

$$\frac{\texttt{class}\ id\ \{\ t_1\ id_1\texttt{;}\ \ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ \ldots\ \ m_k\ \}\quad \text{is in the program}}{fields(id) = [id_1 : t_1, \ldots, id_f : t_f]} \tag{14}$$

$$\frac{\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1\texttt{;}\ \ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ \ldots\ \ m_k\ \}\quad \text{is in the program}}{fields(id) = fields(id^P) \cdot [id_1 : t_1, \ldots, id_f : t_f]} \tag{15}$$

this notation denotes that [id1:t1,...,idf:tf] takes precedence over fields(id^p).

### 6.7 The *methodtype* Helper Function

We use the notation $methodtype(id, id^M)$ to denote the list of argument types of the method with name $id^M$ in class $id$ (or a superclass of $id$) together with the return type (or $\perp$ if no such method exists). The result of *methodtype* is of the form $(t_1, \ldots, t_n) \to t$, or $\perp$. The definition of *methodtype* is:

describes how we keep track of function signatures: methodtype(class, method name)

$$\frac{\begin{array}{l}\texttt{class}\ id\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for some } j \in 1..k : methodname(m_j) = id^M \\ m_j \text{ is of the form} \\ \quad \texttt{public}\ t\ id^M\ \texttt{(}t_1^F\ id_1^F\texttt{, }\ldots\texttt{, }t_n^F\ id_n^F\texttt{)}\ \{t_1\ id_1\texttt{; }\ldots\texttt{; }t_r\ id_r\texttt{; }s_1\ \ldots\ s_q\ \texttt{return}\ e\texttt{; }\}\end{array}}{methodtype(id, id^M) = (id_1^F : t_1^F, \ldots, id_n^F : t_n^F) \to t} \quad (16)$$

method is not in current class (may still be in a superclass)

$$\frac{\begin{array}{l}\texttt{class}\ id\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for all } j \in 1..k : methodname(m_j) \neq id^M\end{array}}{methodtype(id, id^M) = \perp} \quad (17)$$

if a function is in the current class, we get the function signature from the current class, not any super class of the current class

$$\frac{\begin{array}{l}\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for some } j \in 1..k : methodname(m_j) = id^M \\ m_j \text{ is of the form} \\ \quad \texttt{public}\ t\ id^M\ \texttt{(}t_1^F\ id_1^F\texttt{, }\ldots\texttt{, }t_n^F\ id_n^F\texttt{)}\ \{t_1\ id_1\texttt{; }\ldots\texttt{; }t_r\ id_r\texttt{; }s_1\ \ldots\ s_q\ \texttt{return}\ e\texttt{; }\}\end{array}}{methodtype(id, id^M) = (id_1^F : t_1^F, \ldots, id_n^F : t_n^F) \to t} \quad (18)$$

if a function isn't found in the current class, we check the current class's parent for the function signature

$$\frac{\begin{array}{l}\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for all } j \in 1..k : methodname(m_j) \neq id^M\end{array}}{methodtype(id, id^M) = methodtype(id^P, id^M)} \quad (19)$$

### 6.8 The *noOverloading* Helper function

$$\frac{methodtype(id^P, id^M) \neq \perp \ \Rightarrow\ methodtype(id^P, id^M) = methodtype(id, id^M)}{noOverloading(id, id^P, id^M)} \quad (20)$$

if a method of the same name exists in a superclass, then the function signature must be the same

# 7 Type Rules

## 7.1 Type Judgments

We will use the following seven forms of type judgments:

$$\begin{array}{rcll} & \vdash & g & \text{goal type checks} \\ & \vdash & mc & \text{main class type checks} \\ & \vdash & d & \text{type declaration d type checks} \\ C & \vdash & m & \text{method declaration m type checks in class C} \\ A, C & \vdash & s & \text{statement s type checks} \\ A, C & \vdash & e : t & \text{expression type checks to t} \\ A, C & \vdash & p : t & \text{primary expression type checks to t} \end{array}$$

We can read a judgment as follows. The judgment $\vdash g$ means "the goal $g$ type checks." The judgment $\vdash mc$ means "the main class $mc$ type checks." The judgment $\vdash d$ means "the type declaration $d$ type checks." The judgment $C \vdash m$ means "if defined in class $C$, the method declaration $m$ type checks." The judgment $A, C \vdash s$ means "in a type environment $A$, if written in class $C$, the statement $s$ type checks." The judgment $A, C \vdash e : t$ means "in a type environment $A$, if written in class $C$, the expression $e$ has type $t$." The judgment $A, C \vdash p : t$ means "in a type environment $A$, if written in class $C$, the primary expression $p$ has type $t$."

## 7.2 Goal

$$\frac{\begin{array}{l} distinct(classname(mc), classname(d_1), \ldots, classname(d_n)) \quad \text{all classes have distinct names} \\ acyclic(linkset(d_1) \cup \ldots \cup linkset(d_n)) \quad \text{no subclass cycles} \\ \vdash mc \quad \text{main class type checks} \\ \vdash d_i \qquad i \in 1..n \quad \text{all non main classes type check} \end{array}}{\vdash mc\ d_1\ \ldots\ d_n \quad \text{all classes type check}} \tag{21}$$

## 7.3 Main Class

$$\frac{distinct(id_1, \ldots, id_r) \quad \text{all fields in main class are distinct} \qquad [id_1 : t_1,\ \ldots, id_r : t_r], \bot \vdash s_i \quad \text{all statements type check in the environment consisting of only the main function's local variables} \qquad i \in 1..q}{\vdash \texttt{class}\ id\ \{\ \texttt{public static void main (String [] }\ id^S\texttt{)}\{t_1\ id_1;\ \ldots;\ t_r\ id_r;\ s_1 \ldots\ s_q\}\}} \tag{22}$$

## 7.4 Type Declarations

$$\frac{\begin{array}{l} distinct(id_1, \ldots, id_f) \\ distinct(methodname(m_1), \ldots, methodname(m_k)) \\ id \vdash m_i \qquad i \in 1..k \end{array}}{\vdash \texttt{class}\ id\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \}} \quad (23)$$

if you have all distinct field names and all distinct method names and all of your methods type check in the current class, then the whole class type checks

if you have distinct fields and method names, there is no overloading, and all of your methods type check in the current class, then the entire class type checks

$$\frac{\begin{array}{l} distinct(id_1, \ldots, id_f) \\ distinct(methodname(m_1), \ldots, methodname(m_k)) \\ noOverloading(id, id^P, methodname(m_i)) \qquad id \vdash m_i \qquad i \in 1..k \end{array}}{\vdash \texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \}} \quad (24)$$

## 7.5 Method Declarations

cannot have a local variable named the same thing as a parameter in a method declaration. Also all parameters and local variables must have distinct identifiers

parameters and local variables take precedence over class and superclass fields

all statements type check in the current class

return statement type = method type

$$\frac{\begin{array}{l} distinct(id_1^F, \ldots, id_n^F, id_1, \ldots, id_r) \\ A = fields(C) \cdot [id_1^F : t_1^F, \ldots, id_n^F : t_n^F, id_1 : t_1, \ldots, id_r : t_r] \\ A, C \vdash s_i \qquad i \in 1..q \qquad A, C \vdash e : t \end{array}}{C \vdash \texttt{public}\ t\ id^M\ (t_1^F\ id_1^F,\ \ldots,\ t_n^F\ id_n^F)\ \{t_1\ id_1;\ \ldots;\ t_r\ id_r;\ s_1\ \ldots\ s_q\ \texttt{return}\ e;\ \}} \quad (25)$$

## 7.6 Statements

if all statements in a list type check, the whole list type checks

$$\frac{A, C \vdash s_i \qquad i \in 1..q}{A, C \vdash \{\ s_1\ \ldots\ s_q\ \}} \quad (26)$$

if Child extends Parent, then you can say Parent p = new Child();

$$\frac{A(id) = t_1 \qquad A, C \vdash e : t_2 \qquad t_2 \leq t_1}{A, C \vdash id\ \texttt{=}\ e;} \quad (27)$$

integer arrays can be indexed with integers and each index can be assigned an integer value

$$\frac{A(id) = \texttt{int[]} \qquad A, C \vdash e_1 : \texttt{int} \qquad A, C \vdash e_2 : \texttt{int}}{A, C \vdash id\ \texttt{[}\ e_1\ \texttt{]}\ \texttt{=}\ e_2;} \quad (28)$$

$$\frac{A, C \vdash e : \texttt{boolean} \qquad A, C \vdash s_1 \qquad A, C \vdash s_2}{A, C \vdash \texttt{if}\ \texttt{(}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2} \quad (29)$$

$$\frac{A, C \vdash e : \texttt{boolean} \qquad A, C \vdash s}{A, C \vdash \texttt{while}\ \texttt{(}\ e\ \texttt{)}\ s} \quad (30)$$

can only print ints

$$\frac{A, C \vdash e : \texttt{int}}{A, C \vdash \texttt{System.out.println(}\ e\ \texttt{);}} \quad (31)$$

## 7.7 Expressions and Primary Expressions

$$\frac{A, C \vdash p_1 : \texttt{boolean} \qquad A, C \vdash p_2 : \texttt{boolean}}{A, C \vdash p_1 \ \texttt{\&\&} \ p_2 : \texttt{boolean}} \tag{32}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{<} \ p_2 : \texttt{boolean}} \tag{33}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{+} \ p_2 : \texttt{int}} \tag{34}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{-} \ p_2 : \texttt{int}} \tag{35}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{*} \ p_2 : \texttt{int}} \tag{36}$$

$$\frac{A, C \vdash p_1 : \texttt{int[]} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{[} \ p_2 \ \texttt{]} : \texttt{int}} \tag{37}$$

$$\frac{A, C \vdash p : \texttt{int[]}}{A, C \vdash p \ \texttt{.length} : \texttt{int}} \tag{38}$$

when calling an object's member function, you can pass in subtypes of each of the parameters and the function call will still type check

$$\frac{\begin{array}{c} A, C \vdash p : D \qquad methodtype(D, id) = (t'_1, \ldots, t'_n) \to t \\ A, C \vdash e_i : t_i \qquad t_i \leq t'_i \qquad i \in 1..n \end{array}}{A, C \vdash p \ .id \ (e_1, \ \ldots, \ e_n) : t} \tag{39}$$

$A, C \vdash c : \texttt{int}$ c denotes an int (c...onstant) (40)

$A, C \vdash \texttt{true} : \texttt{boolean}$ (41)

$A, C \vdash \texttt{false} : \texttt{boolean}$ (42)

$$\frac{id \in dom(A)}{A, C \vdash id : A(id)}$$ if something is in the current symbol table, then it type checks (43)

$$\frac{C \neq \bot}{A, C \vdash \texttt{this} : C}$$ if we are in a valid class, then 'this' type checks to the current class type (44)

$$\frac{A, C \vdash e : \texttt{int}}{A, C \vdash \texttt{new int[}e\texttt{]} : \texttt{int[]}}$$ integer arrays must have length specified as an integer value upon instantiation (45)

$A, C \vdash \texttt{new} \ id\texttt{()} : id$ (46)

$$\frac{A, C \vdash e : \texttt{boolean}}{A, C \vdash \texttt{!}e : \texttt{boolean}} \tag{47}$$

$$\frac{A, C \vdash e : t}{A, C \vdash \texttt{(}e\texttt{)} : t} \tag{48}$$