Compilerbau

Martin Plümicke

WS 2018/19

Agenda

I. Überblick Vorlesung
  Literatur

II. Compiler Überblick

III. Überblick Funktionale Programmierung
  Einleitung
  Haskell-Grundlagen

IV. Compiler
  Scanner

Literatur

- Bauer and Höllerer.
  Übersetzung objektorientierter Programmiersprachen.
  Springer-Verlag, 1998, (in german).
- Alfred V. Aho, Ravi Lam, Monica S.and Sethi, and Jeffrey D. Ullman.
  Compiler: Prinzipien, Techniken und Werkzeuge.
  Pearson Studium Informatik. Pearson Education Deutschland, 2. edition, 2008.
  (in german).
- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
  Compilers Principles, Techniques and Tools.
  Addison Wesley, 1986.
- Reinhard Wilhelm and Dieter Maurer.
  Übersetzerbau.
  Springer-Verlag, 2. edition, 1992.
  (in german).

Literatur II

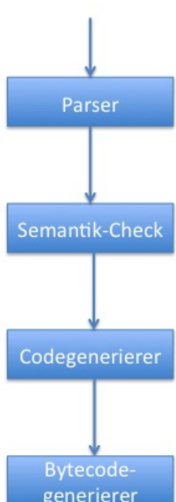
James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java® Virtual Machine Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.

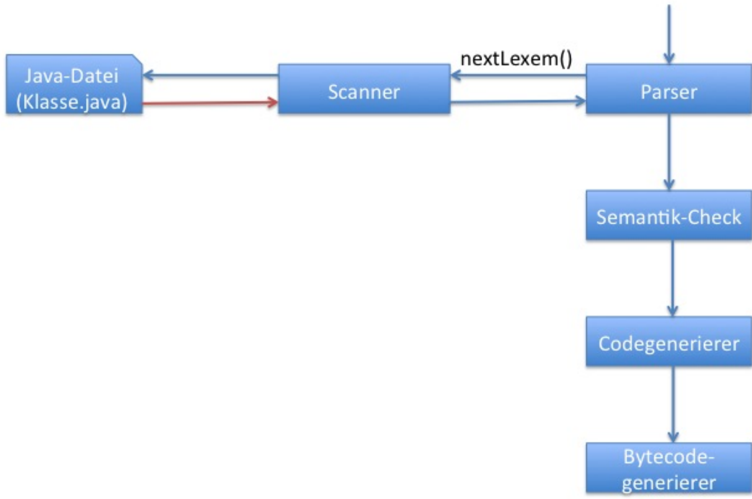
Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.

Peter Thiemann.
Grundlagen der funktionalen Programmierung.
Teubner, 1994.

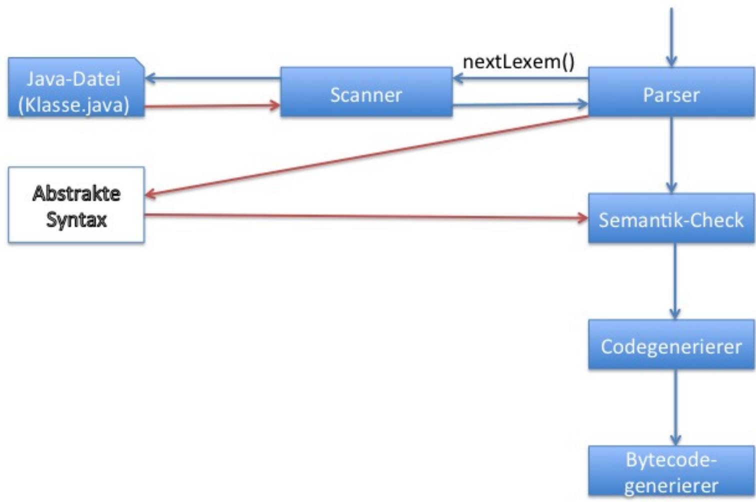
Compiler Überblick



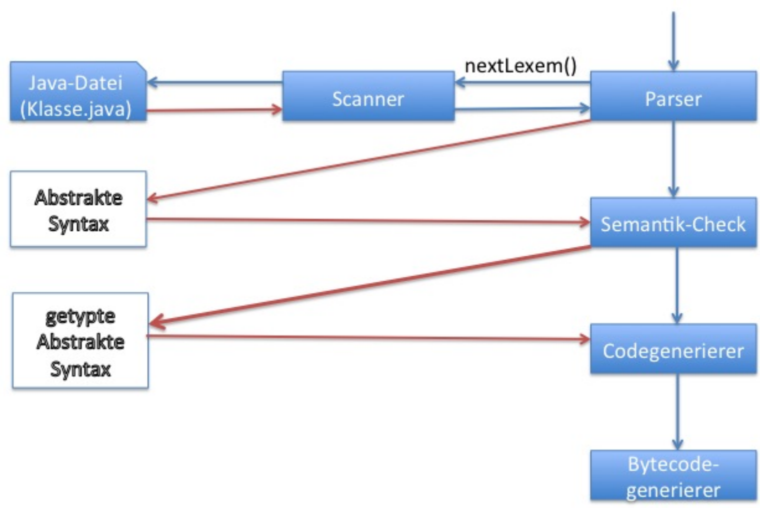
Compiler Überblick



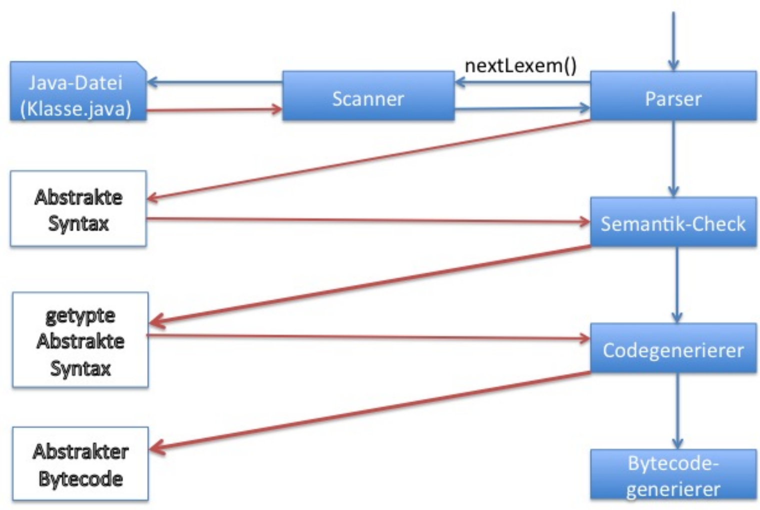
Compiler Überblick



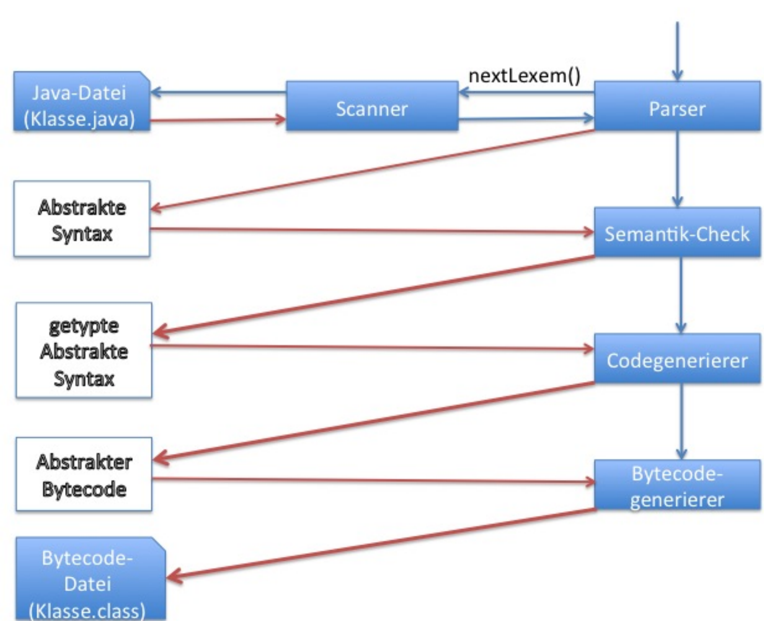
Compiler Überblick



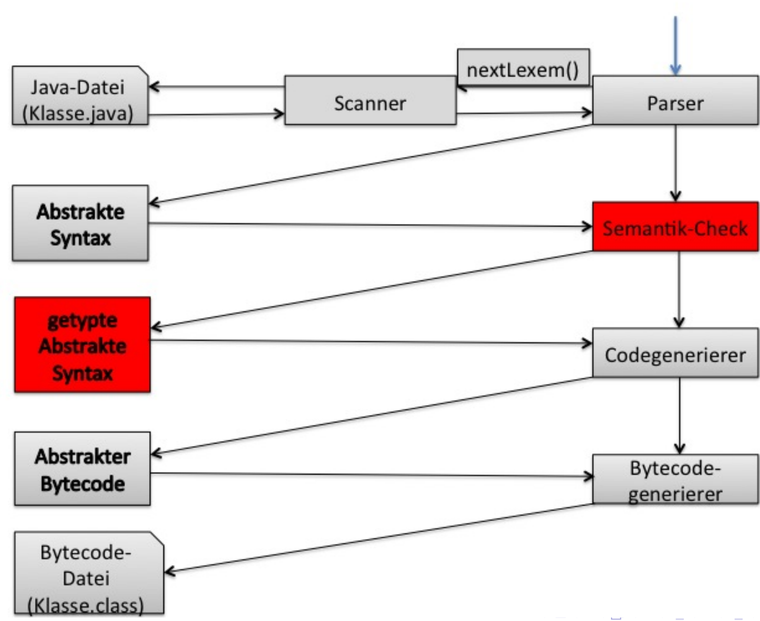
Compiler Überblick



Compiler Überblick



Semantische Analyse/Typecheck



Semantische Analyse/Typecheck

- Überprüfen der Kontextsensitiven Nebenbedingungen:
  - alle Variablen/Methoden deklariert/sichtbar?
  - Typen korrekt?
- Typisierung aller Sub–Terme

Ungetypte abstrakte Syntax für *Mini*–Java I

```
data Class = Class(Type, [FieldDecl], [MethodDecl])

data FieldDecl = FieldDecl(Type, String)

data MethodDecl = Method(Type, String,[(Type,String)], Stmt)

data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl(Type, String)
          | If(Expr, Stmt , Maybe Stmt)
          | StmtExprStmt(StmtExpr)

data StmtExpr = Assign(String, Expr)
              | New(Type, [Expr])
              | MethodCall(Expr, String, [Expr])
```

Ungetypte abstrakte Syntax für *Mini*–Java II

```
data Expr = This
          | Super
          | LocalOrFieldVar(String)
          | InstVar(Expr, String)
          | Unary(String, Expr)
          | Binary(String, Expr, Expr)
          | Integer(Integer)
          | Bool(Bool)
          | Char(Char)
          | String(String)
          | Jnull
          | StmtExprExpr(StmtExpr)

type Prg = [Class]
```

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{\, id_1 : ty_1, \ldots, id_n : ty_n \,\}$

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \ldots, id_n : ty_n \}$

$$O \ \triangleright_{Id} \ id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{\, id_1 : ty_1, \ldots, id_n : ty_n \,\}$

$$O \;\triangleright_{Id}\; id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

$$O \;\triangleright_{Expr}\; e : ty$$

Aus der Menge der Typannahmen O ist für den Ausdruck e der Typ ty ableitbar.

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \ldots, id_n : ty_n \}$

$$O \ \triangleright_{Id} \ id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

$$O \ \triangleright_{Expr} \ e : ty$$

Aus der Menge der Typannahmen O ist für den Ausdruck e der Typ ty ableitbar.

$$O \ \triangleright_{Stmt} \ s : ty$$

Aus der Menge der Typannahmen O ist für das Statement s der Typ ty ableitbar.

Formale Definitionen II

Typurteile

$$[\textbf{Regelname}] \; \frac{O1 \; \triangleright \; x : ty1}{O2 \; \triangleright \; y : ty2}$$

Aus der Regel *Regelname* folgt, wenn man aus $O1$ ableiten kann, dass x den Typ ty1 hat, dann kann man aus $O2$ ableiten dass y den Typ $ty2$ hat.

Ident–Rule

$$[\textbf{Ident}] \; \frac{(f : ty) \in O_\tau}{O_\tau \; \triangleright_{Id} \; f : ty}$$

O_τ: Menge aller in der Klasse τ sichtbaren Methoden und Attribute

Beispiel Ident–Rule

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

Beispiel Ident–Rule

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

- $O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A} \,\}$

$$[\textbf{Ident}]\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}}\ \triangleright_{Id}\ \mathtt{attr} : \mathtt{Integer}}$$

Beispiel Ident–Rule

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

- $O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A} \,\}$

$$[\textbf{Ident}]\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}}\ \rhd_{Id}\ \mathtt{attr} : \mathtt{Integer}}$$

$$[\textbf{Ident}]\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}}\ \rhd_{Id}\ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A}}$$

Literal–Regeln

$$[\textbf{IntLiteral}] \quad O \triangleright_{Expr} \mathsf{Int}(\, n\,) : \texttt{int}$$

$$[\textbf{BoolLiteral}] \quad O \triangleright_{Expr} \mathsf{Bool}(\, b\,) : \texttt{boolean}$$

$$[\textbf{CharLiteral}] \quad O \triangleright_{Expr} \mathsf{Char}(\, c\,) : \texttt{char}$$

$$[\textbf{NullLiteral}] \quad O \triangleright_{Expr} \mathsf{Null} : \theta'$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}] \quad \frac{O \ \triangleright_{Expr} \ e : \texttt{int}}{O \ \triangleright_{Expr} \ \text{Unary}(\text{"+"} / \text{"-"}, e) : \texttt{int}}$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\quad \frac{O \ \triangleright_{Expr} \ e : \texttt{int}}{O \ \triangleright_{Expr} \ \text{Unary}(\text{"+"} / \text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}]\quad \frac{O \ \triangleright_{Expr} \ e : \texttt{boolean}}{O \ \triangleright_{Expr} \ \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\ \frac{O \ \triangleright_{Expr}\ e : \texttt{int}}{O \ \triangleright_{Expr}\ \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}]\ \frac{O \ \triangleright_{Expr}\ e : \texttt{boolean}}{O \ \triangleright_{Expr}\ \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

$$[\textbf{Binary1}]\ \frac{O \ \triangleright_{Expr}\ e1 : \texttt{int}, O \ \triangleright_{Expr}\ e2 : \texttt{int}}{O \ \triangleright_{Expr}\ \text{Binary}(\text{"+"}/\text{"-"}/\text{"*"}/\text{"\%"}, e1, e2) : \texttt{int}}$$

Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}]\ \frac{O\ \triangleright_{Expr}\ e : \texttt{int}}{O\ \triangleright_{Expr}\ \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}]\ \frac{O\ \triangleright_{Expr}\ e : \texttt{boolean}}{O\ \triangleright_{Expr}\ \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

$$[\textbf{Binary1}]\ \frac{O\ \triangleright_{Expr}\ e1 : \texttt{int},\ O\ \triangleright_{Expr}\ e2 : \texttt{int}}{O\ \triangleright_{Expr}\ \text{Binary}(\text{"+"}/\text{"-"}/\text{"*"}/\text{"\%"}, e1, e2) : \texttt{int}}$$

$$[\textbf{Binary2}]\ \frac{O\ \triangleright_{Expr}\ e1 : \texttt{boolean},\ O\ \triangleright_{Expr}\ e2 : \texttt{boolean}}{O\ \triangleright_{Expr}\ \text{Binary}(\text{"\&\&"}/\text{"||"}, e1, e2) : \texttt{boolean}}$$

Expression–Regel: Variablen

$$[\textbf{LocalOrFieldVar}] \quad \frac{O \;\triangleright_{Id}\; v : \theta}{O \;\triangleright_{Expr}\; \mathrm{LocalOrFieldVar}(\, v \,) : \theta}$$

Expression–Regel: Variablen

$$[\textbf{LocalOrFieldVar}] \quad \frac{O \;\triangleright_{Id}\; v : \theta}{O \;\triangleright_{Expr}\; \text{LocalOrFieldVar}(\, v\,) : \theta}$$

$$[\textbf{InstVar}] \quad \frac{O \;\triangleright_{Expr}\; re : \overline{\tau}, \qquad O_{\overline{\tau}} \;\triangleright_{Id}\; v : \theta}{O \;\triangleright_{Expr}\; \text{InstVar}(\, re, v\,) : \theta}$$

Beispiel `InstVar`

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(LocalOrFieldVar(*a*), *attr*).

Beispiel InstVar

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(LocalOrFieldVar(*a*), *attr*).

$$[\textbf{Ident}] \quad \frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id} \mathtt{a} : \mathtt{A}}$$

Beispiel InstVar

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(LocalOrFieldVar(*a*), *attr*).

$$[\textbf{Ident}]\ \frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}$$

$$[\textbf{LocalOrFieldVar}]\ \frac{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}{\{\, a : A \,\} \rhd_{Expr}\ \text{LocalOrFieldVar}(\, a \,) : \mathtt{A}}$$

$$\frac{O_{\mathtt{A}}}{\quad}$$

Beispiel `InstVar`

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(LocalOrFieldVar(*a*), *attr*).

$$[\textbf{LocalOrFieldVar}]\ \dfrac{[\textbf{Ident}]\ \dfrac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}}{\{\, a : A \,\} \rhd_{Expr}\ \text{LocalOrFieldVar}(\, a \,) : \mathtt{A},} \qquad [\textbf{Ident}]\ \dfrac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id}\ \mathtt{attr} : \mathtt{Integer}}$$

Beispiel InstVar

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(LocalOrFieldVar(*a*), *attr*).

$$\dfrac{[\textbf{Ident}]\ \dfrac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}}{[\textbf{LocalOrFieldVar}]\ \{\, a : A \,\} \rhd_{Expr}\ \text{LocalOrFieldVar}(\, a\,) : \mathtt{A},} \quad [\textbf{Ident}]\ \dfrac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id}\ \mathtt{attr} : \mathtt{Integer}}$$

$$[\textbf{InstVar}]\ \overline{\{\, a : A \,\} \rhd_{Expr}\ \textbf{InstVar}(\, \text{LocalOrFieldVar}(\, a\,), attr\,) : \mathtt{Integer}}$$

Expression–Regel: Statement–Expressions

[**New**] $\quad O \ \triangleright_{Expr} \ \mathrm{New}(\,\theta\,) : \theta$

[1] $\leq^*$ ist die Subtypen–Relation

Expression–Regel: Statement–Expressions

[New] $\quad O \ \rhd_{Expr} \ \text{New}(\ \theta\ ) : \theta$

[Assign] $\quad \dfrac{O \ \rhd_{Expr} \ ve : \theta', O \ \rhd_{Expr} \ e : \theta}{O \ \rhd_{Expr} \ \text{Assign}(\ ve, e\ ) : \theta'} \ \theta \leq^* \theta'$[1]

[1] $\leq^*$ ist die Subtypen–Relation

Expression–Regel: Statement–Expressions

[New] $O \ \triangleright_{Expr} \ \mathrm{New}(\, \theta \,) : \theta$

[Assign]
$$\frac{O \ \triangleright_{Expr} \ ve : \theta',\ O \ \triangleright_{Expr} \ e : \theta}{O \ \triangleright_{Expr} \ \mathrm{Assign}(\, ve, e \,) : \theta'} \quad \theta \leq^* \theta'^{1}$$

[Method-Call]
$$\frac{\begin{array}{c} O \ \triangleright_{Expr} \ re : \overline{\tau} \\ O_{\overline{\tau}} \ \triangleright_{Id} \ m : \theta'_1 \times \ldots \times \theta'_n \rightarrow \theta \\ \forall 1 \leqslant i \leqslant n : \ O \ \triangleright_{Expr} \ e_i : \theta_i \end{array}}{O \ \triangleright_{Expr} \ \mathrm{MethodCall}(\, re, m, (e_1, \ldots, e_n) \,) : \theta} \quad \theta_i \leq^* \theta'_i$$

[1] $\leq^*$ ist die Subtypen–Relation

Beispiel MethodCall

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

Beispiel MethodCall

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(a), *meth*, Bool(`true`)).

$$[\textbf{Ident}] \quad \frac{\{\, a : A \,\}}{\{\, a : A \,\} \ \rhd_{Id} \ \mathtt{a} : \mathtt{A}}$$

Beispiel MethodCall

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{ \mathtt{attr} : \mathtt{Integer}, \ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A} \}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(a), $meth$, Bool(`true`)).

$$[\textbf{LocalOrFieldVar}] \frac{[\textbf{Ident}] \dfrac{\{ a : A \}}{\{ a : A \} \rhd_{Id} \ \mathtt{a} : \mathtt{A}}}{\{ a : A \} \rhd_{Expr} \ \mathrm{LocalOrFieldVar}(a) : \mathtt{A}}$$

$$\frac{O_{\mathtt{A}}}{\quad}$$

Beispiel MethodCall

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(a), $meth$, Bool(`true`)).

$$[\textbf{Ident}]\ \dfrac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}$$

$$[\textbf{LocalOrFieldVar}]\ \dfrac{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}{\{\, a : A \,\} \rhd_{Expr}\ \mathrm{LocalOrFieldVar}(\, a \,) : \mathtt{A},}$$

$$[\textbf{Ident}]\ \dfrac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id}\ \mathtt{meth} : \mathtt{Boolean} \to \mathtt{A}}$$

Beispiel MethodCall

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_A = \{ \texttt{attr} : \texttt{Integer}, \ \texttt{meth} : \texttt{Boolean} \rightarrow \texttt{A} \}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

$$\textbf{MethodCall}(\ \textbf{LocalOrFieldVar}(\ a\ ),\ \mathit{meth},\ \text{Bool}(\ \texttt{true}\ )\ ).$$

$$[\textbf{LocalOrFieldVar}]\ \dfrac{[\textbf{Ident}]\ \dfrac{\{\ a : A\ \}}{\{\ a : A\ \} \rhd_{Id}\ \texttt{a} : \texttt{A}}}{\{\ a : A\ \} \rhd_{Expr}\ \text{LocalOrFieldVar}(\ a\ ) : \texttt{A},}$$

$$[\textbf{Ident}]\ \dfrac{O_A}{O_A \rhd_{Id}\ \texttt{meth} : \texttt{Boolean} \rightarrow \texttt{A}}$$

$$[\textbf{BoolLiteral}]\ \text{Bool}(\ \texttt{true}\ ) : \texttt{boolean}$$

Beispiel `MethodCall`

```
class A {
    Integer attr;
    A meth(Boolean x) { ... }
}
```

$O_{\mathtt{A}} = \{\, \mathtt{attr} : \mathtt{Integer},\ \mathtt{meth} : \mathtt{Boolean} \rightarrow \mathtt{A} \,\}$

```
...
A a = new A()
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(`true`)).

$$\text{[Ident]}\ \frac{\{\, a : A \,\}}{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}$$

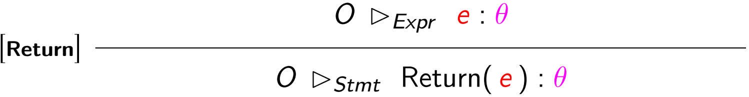
$$\text{[LocalOrFieldVar]}\ \frac{\{\, a : A \,\} \rhd_{Id}\ \mathtt{a} : \mathtt{A}}{\{\, a : A \,\} \rhd_{Expr}\ \text{LocalOrFieldVar}(\, a \,) : \mathtt{A},}$$

$$\text{[Ident]}\ \frac{O_{\mathtt{A}}}{O_{\mathtt{A}} \rhd_{Id}\ \mathtt{meth} : \text{Boolean} \rightarrow \mathtt{A}}$$

$$\text{[BoolLiteral]}\ \text{Bool}(\, \mathtt{true} \,) : \mathtt{boolean}$$

$$\text{[MethodCall]}\ \frac{\{\, a : A \,\} \rhd_{Expr}\ \text{LocalOrFieldVar}(\, a \,) : \mathtt{A}, \quad O_{\mathtt{A}} \rhd_{Id}\ \mathtt{meth} : \text{Boolean} \rightarrow \mathtt{A}, \quad \text{Bool}(\, \mathtt{true} \,) : \mathtt{boolean}}{\{\, a : A \,\} \rhd_{Expr}\ \textbf{MethodCall}(\, \textbf{LocalOrFieldVar}(\, a \,), \mathtt{meth}, \text{Bool}(\mathtt{true}) \,) : \mathtt{A}}$$

Statement–Regeln



$$[\textbf{Return}]\ \frac{O \ \rhd_{Expr}\ e : \theta}{O \ \rhd_{Stmt}\ \text{Return}(\ e\ ) : \theta}$$

[2] upper bound

Statement–Regeln

$$[\textbf{Return}] \quad \frac{O \ \triangleright_{Expr} \ e : \theta}{O \ \triangleright_{Stmt} \ \text{Return}(\ e\ ) : \theta}$$

$$[\textbf{If}] \quad \frac{\begin{array}{c} O \ \triangleright_{Stmt} \ s_1 : \theta_1, O \ \triangleright_{Stmt} \ s_2 : \theta_2 \\ O \ \triangleright_{Expr} \ e : \texttt{boolean} \end{array}}{O \ \triangleright_{Stmt} \ \text{If}(\ e, s_1, s_2\ ) : \overline{\theta}, \textit{wobei } \overline{\theta} \in \textbf{UB}^2(\ \theta_1, \theta_2\ )}$$

[2] upper bound

Statement–Regeln

$$[\textbf{Return}] \quad \frac{O \ \triangleright_{Expr} \ e : \theta}{O \ \triangleright_{Stmt} \ \text{Return}(\ e\ ) : \theta}$$

$$[\textbf{If}] \quad \frac{O \ \triangleright_{Stmt} \ s_1 : \theta_1, O \ \triangleright_{Stmt} \ s_2 : \theta_2 \qquad O \ \triangleright_{Expr} \ e : \texttt{boolean}}{O \ \triangleright_{Stmt} \ \text{If}(\ e, s_1, s_2\ ) : \overline{\theta}, \textit{wobei } \overline{\theta} \in \textbf{UB}^2(\ \theta_1, \theta_2\ )}$$

$$[\textbf{While}] \quad \frac{O \ \triangleright_{Expr} \ e : \texttt{boolean}, \ O \ \triangleright_{Stmt} \ \text{Block}(\ B\ ) : \theta}{O \ \triangleright_{Stmt} \ \text{While}(\ e, \text{Block}(\ B\ )\ ) : \theta}$$

[2] upper bound

Statement–Regeln: Statement–Expressions

[**New**] $\qquad O \;\triangleright_{Stmt}\; \mathrm{New}(\,\theta\,) : \texttt{void}$

Statement–Regeln: Statement–Expressions

[New] $\quad O \ \triangleright_{Stmt} \ \text{New}(\ \theta\ ) : \texttt{void}$

[Assign]

$$\frac{O \ \triangleright_{Expr} \ ve : \theta', O \ \triangleright_{Expr} \ e : \theta}{O \ \triangleright_{Stmt} \ \text{Assign}(\ ve, e\ ) : \texttt{void}} \quad \theta \leq^{*} \theta'$$

Statement–Regeln: Statement–Expressions

[New] $O \ \triangleright_{Stmt} \ \text{New}(\,\theta\,) : \texttt{void}$

[Assign]

$$\frac{O \ \triangleright_{Expr} \ ve : \theta', O \ \triangleright_{Expr} \ e : \theta}{O \ \triangleright_{Stmt} \ \text{Assign}(\,ve, e\,) : \texttt{void}} \quad \theta \leq^* \theta'$$

[Method-Call]

$$\frac{\begin{array}{c} O \ \triangleright_{Expr} \ re : \overline{\tau} \\ O_{\overline{\tau}} \ \triangleright_{Id} \ m : \theta'_1 \times \ldots \times \theta'_n \to \theta \\ \forall 1 \leqslant i \leqslant n : \ O \ \triangleright_{Expr} \ e_i : \theta_i \end{array}}{O \ \triangleright_{Stmt} \ \text{MethodCall}(\,re, m, (e_1, \ldots, e_n)\,) : \texttt{void}} \quad \theta_i \leq^* \theta'_i$$

Block–Statement Regeln

$$[\textbf{BlockInit}] \quad \frac{O \ \rhd_{Stmt} \ stmt : \theta}{O \ \rhd_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

Block–Statement Regeln

[BlockInit]

$$\frac{O \ \triangleright_{Stmt} \ stmt : \theta}{O \ \triangleright_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

[Block]

$$\frac{O \ \triangleright_{Stmt} \ s_1 : \theta, \ \ O \ \triangleright_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta'}{O \ \triangleright_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \overline{\theta}, \textit{wobei } \overline{\theta} \in \mathbf{UB}(\ \theta, \theta'\ )}$$

Block–Statement Regeln

[BlockInit]

$$\frac{O \;\triangleright_{Stmt}\; stmt : \theta}{O \;\triangleright_{Stmt}\; \text{Block}(\, stmt \,) : \theta}$$

[Block]

$$\frac{O \;\triangleright_{Stmt}\; s_1 : \theta,\;\; O \;\triangleright_{Stmt}\; \text{Block}(\, s_2; \ldots; s_n; \,) : \theta'}{O \;\triangleright_{Stmt}\; \text{Block}(\, s_1; s_2; \ldots; s_n; \,) : \overline{\theta},\; \textit{wobei}\; \overline{\theta} \in \mathbf{UB}(\, \theta, \theta' \,)}$$

[Blockvoid]

$$\frac{\begin{array}{c} O \;\triangleright_{Stmt}\; s_1 : \texttt{void}, \\ O \;\triangleright_{Stmt}\; \text{Block}(\, s_2; \ldots; s_n; \,) : \theta \end{array}}{O \;\triangleright_{Stmt}\; \text{Block}(\, s_1; s_2; \ldots; s_n; \,) : \theta}$$

Block–Statement Regeln

[BlockInit]

$$\frac{O \ \triangleright_{Stmt} \ stmt : \theta}{O \ \triangleright_{Stmt} \ \mathrm{Block}(\ stmt\ ) : \theta}$$

[Block]

$$\frac{O \ \triangleright_{Stmt} \ s_1 : \theta, \ O \ \triangleright_{Stmt} \ \mathrm{Block}(\ s_2; \ldots; s_n;\ ) : \theta'}{O \ \triangleright_{Stmt} \ \mathrm{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \overline{\theta}, \mathit{wobei}\ \overline{\theta} \in \mathbf{UB}(\ \theta, \theta'\ )}$$

[Blockvoid]

$$\frac{\begin{array}{c} O \ \triangleright_{Stmt} \ s_1 : \mathtt{void}, \\ O \ \triangleright_{Stmt} \ \mathrm{Block}(\ s_2; \ldots; s_n;\ ) : \theta \end{array}}{O \ \triangleright_{Stmt} \ \mathrm{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \theta}$$

[Block-Local-VarDecl]

$$\frac{O \backslash \{\ v : \theta'\ \} \cup \{\ v : \overline{\theta}\ \} \ \triangleright_{Stmt} \ \mathrm{Block}(\ s_2; \ldots; s_n;\ ) : \theta}{O \ \triangleright_{Stmt} \ \mathrm{Block}(\ \mathrm{LocalVarDecl}(\ v, \overline{\theta}\ ); s_2; \ldots; s_n;\ ) : \theta}$$

Beispiel

```
while (true) {
    return 1;
}
```

Beispiel

```
while (true) {
    return 1;
}
```

$O = \emptyset$

Beispiel

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$[\textbf{IntLiteral}] \qquad O \rhd_{Expr} \text{Int}(1) : \texttt{int}$$

Beispiel

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$\textbf{[Return]}\ \frac{\textbf{[IntLiteral]}\ \ O \ \triangleright_{Expr}\ \mathrm{Int}(\,1\,) : \texttt{int}}{O \ \triangleright_{Stmt}\ \mathrm{Return}(\,\mathrm{Int}(\,1\,)\,) : \texttt{int}}$$

Beispiel

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$\dfrac{\dfrac{[\textbf{IntLiteral}] \quad O \triangleright_{Expr} \text{Int}(1) : \texttt{int}}{O \triangleright_{Stmt} \text{Return}(\text{Int}(1)) : \texttt{int}}\,[\textbf{Return}]}{O \triangleright_{Stmt} \text{Block}(\text{Return}(\text{Int}(1))) : \texttt{int}}\,[\textbf{Block-Init}]$$

Beispiel

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$[\textbf{Bool-Literal}]\; O \rhd_{Expr} \text{Bool}(\,\textit{True}\,) : \texttt{boolean}$

$$
[\textbf{Block-Init}]\;
\dfrac{[\textbf{Return}]\;\dfrac{[\textbf{IntLiteral}]\quad O \rhd_{Expr} \text{Int}(\,1\,) : \texttt{int}}{O \rhd_{Stmt} \text{Return}(\,\text{Int}(\,1\,)\,) : \texttt{int}}}{O \rhd_{Stmt} \text{Block}(\,\text{Return}(\,\text{Int}(\,1\,)\,)\,) : \texttt{int}}
$$

Beispiel

```
while (true) {
    return 1;
}
```

$O = \emptyset$

$$\dfrac{[\textbf{Bool-Literal}]\; O \rhd_{Expr} \text{Bool}(\,\mathit{True}\,) : \texttt{boolean} \qquad [\textbf{Block-Init}]\;\dfrac{[\textbf{Return}]\;\dfrac{[\textbf{IntLiteral}]\; O \rhd_{Expr} \text{Int}(\,1\,) : \texttt{int}}{O \rhd_{Stmt} \text{Return}(\,\text{Int}(\,1\,)\,) : \texttt{int}}}{O \rhd_{Stmt} \text{Block}(\,\text{Return}(\,\text{Int}(\,1\,)\,)\,) : \texttt{int}}}{O \rhd_{Stmt} \text{While}(\,\text{Bool}(\,\mathit{True}\,), \text{Block}(\,\text{Return}(\,\text{Int}(\,1\,)\,)\,)\,) : \texttt{int}}\;[\textbf{while}]$$

Datenstruktur typisierter Expressions

```
type Type = String

-- data Type = TVar(String)          -- Typvariable
--           | TC(String, [Type])    -- Typkonstruktor
--           | WC                    -- Wildscard
--           | WC_Super(Type)        -- Super-Wildscard
--           | WC_Extends(Type)      -- Extends-Wildcard
```

Datenstruktur typisierter Expressions

```
type Type = String

-- data Type = TVar(String)                -- Typvariable
--           | TC(String, [Type])          -- Typkonstruktor
--           | WC                          -- Wildscard
--           | WC_Super(Type)              -- Super-Wildscard
--           | WC_Extends(Type)            -- Extends-Wildcard

data Expr = This
          | Super
          | LocalOrFieldVar(String)
          | InstVar(Expr, String)
          | Unary(String, Expr)
          | Binary(String, Expr, Expr)
          | Integer(Integer)
          | Bool(Bool)
          | Char(Char)
          | String(String)
          | Jnull
          | StmtExprExpr(StmtExpr)
          | TypedExpr(Expr, Type)
```

Datenstruktur typisierter Statements

```
data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl(String)
          | If(Expr, Stmt , Maybe Stmt)
          | StmtExprStmt(StmtExpr)
          | TypedStmt(Stmt, Type)

data StmtExpr = Assign(String, Expr)
          | New(Type, [Expr])
          | MethodCall(Expr, String, [Expr])
          | TypedStmtExpr(StmtExpr, Type)
```

Typisierung von Simple–Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung von Simple–Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung:

```
TypedExpr(Binary("+",
                 TypedExpr(Integer(1), "int"),
                 TypedExpr(Integer(2), "int"))
          "int")
```

LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) {
    return v + x;
  }
}
```

LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) {
    return v + x;
  }
}
```

Erg. der Parsers:

```
Return(Binary("+",
              LocalOrFieldVar("v")      ← FieldVar
              LocalOrFieldVar("x")))    ← LocalVar
```

LocalOrFieldVar

```
class Klassenname {
  int v;
  int methode (int x) {
    return v + x;
  }
}
```

Erg. der Parsers:

```
Return(Binary("+",
                 LocalOrFieldVar("v")      ← FieldVar
                 LocalOrFieldVar("x")))    ← LocalVar
```

Typisierung:

```
Return(
  TypedExpr(
       Binary("+",
                 TypedExpr(LocalOrFieldVar("v"), "int"),
                 TypedExpr(LocalOrFieldVar("x"), "int")),
        "int"))
```

InstVar, MethodCall

```
class Cl1 {
   char m1 () {
     int b;
     Cl2 x = new Cl2 ()
     return x.m2(x.v, b);
   }
}
```

```
class Cl2 {
  Cl3 v;
  char m2(cl3 v, int w) { ...}
 }

class Cl3 { ...}
```

InstVar, MethodCall

```
class Cl1 {
   char m1 () {
     int b;
     Cl2 x = new Cl2 ()
     return x.m2(x.v, b);
   }
}
```

```
class Cl2 {
  Cl3 v;
  char m2(cl3 v, int w) { ...}
 }

class Cl3 { ...}
```

Erg. der Parsers:

```
Return(MethodCall(LocalOrFieldVar("x"), "m2",
                  [InstVar(LocalOrFieldVar("x"), "v"),
                   LocalOrFieldVar("b")]))
```

InstVar, MethodCall

```
class Cl1 {
   char m1 () {
     int b;
     Cl2 x = new Cl2 ()
     return x.m2(x.v, b);
   }
}
```

```
class Cl2 {
  Cl3 v;
  char m2(cl3 v, int w) { ...}
 }

class Cl3 { ...}
```

Erg. der Parsers:

```
Return(MethodCall(LocalOrFieldVar("x"), "m2",
                  [InstVar(LocalOrFieldVar("x"), "v"),
                   LocalOrFieldVar("b")]))
```

Typisierung:

```
Return(TypedExpr(
  MethodCall(TypedExpr(LocalOrFieldVar("x"), "Cl2"), "m2",
             [TypedExpr(
                InstVar(TypedExpr(LocalOrFieldVar("x"), "Cl2"),
                        "v"), "Cl3"),
              TypedExpr(LocalOrFieldVar("b"), "int")]), "char"))
```

Semantische Analyse/Algorithmus `typecheck`

```
typecheckExpr:: Expr -> [(String, Type)] -> [Class] -> Expr
typecheckStmt:: Stmt -> [(String, Type)] -> [Class] -> Stmt
```

1. Argument: Ungetypter Ausdruck/Statement
2. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
3. Argument: Alle sichtbaren Klassen mit Fields und Methoden

Ergebnis: getypeter Ausdruck/Statement

Semantische Analyse/Algorithmus `typecheck`

```
typecheckExpr:: Expr -> [(String, Type)] -> [Class] -> Expr
typecheckStmt:: Stmt -> [(String, Type)] -> [Class] -> Stmt
```

1. Argument: Ungetypter Ausdruck/Statement
2. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
3. Argument: Alle sichtbaren Klassen mit Fields und Methoden

Ergebnis: getypeter Ausdruck/Statement

Ablauf: Lauf über alle Methoden aller Klassen:

- Check ob Variablen/Methoden deklariert
- Check, ob die Typen der Metdodenaufrufe/Zuweisung korrekt sind
- Einfügen der Typisierungen