Expressions

Expressions

```
Expr         ::=  (Bindings | id | ‘_’) ‘=>’ Expr
               |  Expr1
Expr1        ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
               |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
               |  ‘try’ Expr [‘catch’ Expr] [‘finally’ Expr]
               |  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ‘)’
          | ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’) {nl} [‘yield’] Expr
               |  ‘throw’ Expr
               |  ‘return’ [Expr]
               |  [SimpleExpr ‘.’] id ‘=’ Expr
               |  SimpleExpr1 ArgumentExprs ‘=’ Expr
               |  PostfixExpr
               |  PostfixExpr Ascription
               |  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
PostfixExpr  ::=  InfixExpr [id [nl]]
InfixExpr    ::=  PrefixExpr
               |  InfixExpr id [nl] InfixExpr
PrefixExpr   ::=  [‘-’ | ‘+’ | ‘~’ | ‘!’] SimpleExpr
SimpleExpr   ::=  ‘new’ (ClassTemplate | TemplateBody)
               |  BlockExpr
               |  SimpleExpr1 [‘_’]
SimpleExpr1  ::=  Literal
               |  Path
               |  ‘_’
               |  ‘(’ [Exprs] ‘)’
               |  SimpleExpr ‘.’ id
               |  SimpleExpr TypeArgs
               |  SimpleExpr1 ArgumentExprs
               |  XmlExpr
Exprs        ::=  Expr {‘,’ Expr}
BlockExpr    ::=  ‘{’ CaseClauses ‘}’
               |  ‘{’ Block ‘}’
Block        ::=  BlockStat {semi BlockStat} [ResultExpr]
```

```
ResultExpr   ::=  Expr1
           |  (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block
Ascription   ::=  ‘:’ InfixType
                |  ‘:’ Annotation {Annotation}
                |  ‘:’ ‘_’ ‘*’
```

Expressions are composed of operators and operands. Expression forms are discussed subsequently in decreasing order of precedence.

Expression Typing

The typing of expressions is often relative to some *expected type* (which might be undefined). When we write "expression e is expected to conform to type $T$", we mean: 1. the expected type of e is $T$, and 2. the type of expression e must conform to $T$.

The following skolemization rule is applied universally for every expression: If the type of an expression would be an existential type $T$, then the type of the expression is assumed instead to be a skolemization of $T$.

Skolemization is reversed by type packing. Assume an expression e of type $T$ and let $t_1[\mathit{tps\_1}] >: L_1 <: U_1, \ldots, t_n[\mathit{tps\_n}] >: L_n <: U_n$ be all the type variables created by skolemization of some part of e which are free in $T$. Then the *packed type* of e is

```
$T$ forSome { type $t_1[\mathit{tps}\_1] >: L_1 <: U_1$; $\ldots$; type $t_n[\mathit{tps}\_
```

Literals

```
SimpleExpr    ::=  Literal
```

Typing of literals is described along with their lexical syntax; their evaluation is immediate.

The *Null* Value

The `null` value is of type `scala.Null`, and thus conforms to every reference type. It denotes a reference value which refers to a special `null` object. This object implements methods in class `scala.AnyRef` as follows:

- `eq($x\,$)` and `==($x\,$)` return `true` iff the argument x is also the "null" object.
- `ne($x\,$)` and `!=($x\,$)` return true iff the argument x is not also the "null" object.
- `isInstanceOf[$T\,$]` always returns `false`.
- `asInstanceOf[$T\,$]` returns the default value of type $T$.
- `##` returns `0`.

A reference to any other member of the "null" object causes a `NullPointerException` to be thrown.

Designators

```
SimpleExpr  ::=  Path
              |  SimpleExpr ‘.’ id
```

A designator refers to a named term. It can be a *simple name* or a *selection*.

A simple name x refers to a value as specified here. If x is bound by a definition or declaration in an enclosing class or object C, it is taken to be equivalent to the selection `$C$.this.$x$` where C is taken to refer to the class containing x even if the type name C is shadowed at the occurrence of x.

If r is a stable identifier of type T, the selection $r.x$ refers statically to a term member m of r that is identified in T by the name x.

For other expressions e, $e.x$ is typed as if it was `{ val $y$ = $e$; $y$.$x$ }`, for some fresh name y.

The expected type of a designator's prefix is always undefined. The type of a designator is the type T of the entity it refers to, with the following exception: The type of a path p which occurs in a context where a stable type is required is the singleton type `$p$.type`.

The contexts where a stable type is required are those that satisfy one of the following conditions:

1. The path p occurs as the prefix of a selection and it does not designate a constant, or
2. The expected type *pt* is a stable type, or
3. The expected type *pt* is an abstract type with a stable type as lower bound, and the type T of the entity referred to by p does not conform to *pt*, or
4. The path p designates a module.

The selection $e.x$ is evaluated by first evaluating the qualifier expression e, which yields an object r, say. The selection's result is then the member of r that is either defined by m or defined by a definition overriding m.

This and Super

```
SimpleExpr  ::=  [id ‘.’] ‘this’
              |  [id ‘.’] ‘super’ [ClassQualifier] ‘.’ id
```

The expression `this` can appear in the statement part of a template or compound type. It stands for the object being defined by the innermost template or compound type enclosing the reference. If this is a compound type, the type of `this` is that compound type. If it is a template of a class or object definition with simple name C, the type of this is the same as the type of `$C$.this`.

The expression `$C$.this` is legal in the statement part of an enclosing class or object definition with simple name C. It stands for the object being defined by the innermost such definition. If the expression's expected type is a stable type, or `$C$.this` occurs as the prefix of a selection, its type is `$C$.this.type`, otherwise it is the self type of class C.

A reference `super.$m$` refers statically to a method or type m in the least proper supertype of the innermost template containing the reference. It evaluates to the member m' in the actual supertype of that template which is equal to m or which overrides m. The statically referenced member m must be a type or a method.

If it is a method, it must be concrete, or the template containing the reference must have a member m' which overrides m and which is labeled `abstract override`.

A reference `$C$.super.$m$` refers statically to a method or type m in the least proper supertype of the innermost enclosing class or object definition named C which encloses the reference. It evaluates to the member m' in the actual supertype of that class or object which is equal to m or which overrides m. The statically referenced member m must be a type or a method. If the statically referenced member m is a method, it must be concrete, or the innermost enclosing class or object definition named C must have a member m' which overrides m and which is labeled `abstract override`.

The `super` prefix may be followed by a trait qualifier `[$T\,$]`, as in `$C$.super[$T\,$].$x$`. This is called a *static super reference*. In this case, the reference is to the type or method of x in the parent trait of C whose simple name is T. That member must be uniquely defined. If it is a method, it must be concrete.

Example

Consider the following class definitions

```
class Root { def x = "Root" }
class A extends Root { override def x = "A" ; def superA = super.x }
trait B extends Root { override def x = "B" ; def superB = super.x }
class C extends Root with B {
  override def x = "C" ; def superC = super.x
}
class D extends A with B {
  override def x = "D" ; def superD = super.x
}
```

The linearization of class `C` is `{C, B, Root}` and the linearization of class `D` is `{D, B, A, Root}`. Then we have:

```
(new A).superA == "Root"

(new C).superB == "Root"
(new C).superC == "B"

(new D).superA == "Root"
(new D).superB == "A"
(new D).superD == "B"
```

Note that the `superB` method returns different results depending on whether `B` is mixed in with class `Root` or `A`.

Function Applications

```
SimpleExpr    ::=  SimpleExpr1 ArgumentExprs
ArgumentExprs ::=  ‘(’ [Exprs] ‘)’
                |  ‘(’ [Exprs ‘,’] PostfixExpr ‘:’ ‘_’ ‘*’ ‘)’
                |  [nl] BlockExpr
Exprs         ::=  Expr {‘,’ Expr}
```

An application `$f(e_1 , \ldots , e_m)$` applies the function `$f$` to the argument expressions `$e_1, \ldots , e_m$`. For this expression to be well-typed, the function must be *applicable* to its arguments, which is defined next by case analysis on f's type.

If f has a method type `($p_1$:$T_1 , \ldots , p_n$:$T_n$)$U$`, each argument expression e_i is typed with the corresponding parameter type T_i as expected type. Let S_i be the type of argument e_i $(i = 1, \ldots , m)$. The method f must be *applicable* to its arguments $e_1, \ldots , e_n$ of types $S_1, \ldots , S_n$. We say that an argument expression e_i is a *named* argument if it has the form `$x_i=e'_i$` and `$x_i$` is one of the parameter names `$p_1, \ldots, p_n$`.

Once the types S_i have been determined, the method f of the above method type is said to be applicable if all of the following conditions hold: - for every named argument $p_j = e'_i$ the type S_i is compatible with the parameter type T_j; - for every positional argument e_i the type S_i is compatible with T_i; - if the expected type is defined, the result type U is compatible to it.

If f is a polymorphic method, local type inference is used to instantiate f's type parameters. The polymorphic method is applicable if type inference can determine type arguments so that the instantiated method is applicable.

If f has some value type, the application is taken to be equivalent to `$f$.apply($e_1 , \ldots , e_m$)`, i.e. the application of an `apply` method defined by f. The value `$f$` is applicable to the given arguments if `$f$.apply` is applicable.

Evaluation of `$f$($e_1 , \ldots , e_n$)` usually entails evaluation of f and $e_1, \ldots , e_n$ in that order. Each argument expression is converted to the type of its corresponding formal parameter. After that, the application is rewritten to the function's right hand side, with actual arguments substituted for formal parameters. The result of evaluating the rewritten right-hand side is finally converted to the function's declared result type, if one is given.

The case of a formal parameter with a parameterless method type `=> $T$` is treated specially. In this case, the corresponding actual argument expression e is not evaluated before the application. Instead, every use of the formal parameter on the right-hand side of the rewrite rule entails a re-evaluation of e. In other words, the evaluation order for `=>`-parameters is *call-by-name* whereas the evaluation order for normal parameters is *call-by-value*. Furthermore, it is required that e's packed type conforms to the parameter type T. The behavior of by-name parameters is preserved if the application is transformed into a block due to named or default arguments. In this case, the local value for that parameter has the form `val $y_i$ = () => $e$` and the argument passed to the function is `$y_i$()`.

The last argument in an application may be marked as a sequence argument, e.g. `$e$: _*`. Such an argument must correspond to a repeated parameter of type `$S$*` and it must be the only argument matching this parameter (i.e. the number of formal parameters and actual arguments must be the same). Furthermore, the type of e must conform to `scala.Seq[$T$]`, for some type T which conforms to S. In this case, the argument list is transformed by replacing the sequence e with its elements. When the application uses named arguments, the vararg parameter has to be specified exactly once.

A function application usually allocates a new frame on the program's run-time stack. However, if a local method or a final method calls itself as its last action, the call is executed using the stack-frame of the caller.

Example

Assume the following method which computes the sum of a variable number of arguments:

```scala
def sum(xs: Int*) = (0 /: xs) ((x, y) => x + y)
```

Then

```scala
sum(1, 2, 3, 4)
sum(List(1, 2, 3, 4): _*)
```

both yield `10` as result. On the other hand,

```scala
sum(List(1, 2, 3, 4))
```

would not typecheck.

Named and Default Arguments

If an application is to use named arguments $p = e$ or default arguments, the following conditions must hold.

- For every named argument $p_i = e_i$ which appears left of a positional argument in the argument list $e_1 \ldots e_m$, the argument position i coincides with the position of parameter p_i in the parameter list of the applied method.
- The names x_i of all named arguments are pairwise distinct and no named argument defines a parameter which is already specified by a positional argument.
- Every formal parameter $p_j : T_j$ which is not specified by either a positional or named argument has a default argument.

If the application uses named or default arguments the following transformation is applied to convert it into an application without named or default arguments.

If the method f has the form `$p.m$[$\mathit{targs}$]` it is transformed into the block

```
{ val q = $p$
  q.$m$[$\mathit{targs}$]
}
```

If the method f is itself an application expression the transformation is applied recursively on f. The result of transforming f is a block of the form

```
{ val q = $p$
  val $x_1$ = expr$_1$
  $\ldots$
  val $x_k$ = expr$_k$
  q.$m$[$\mathit{targs}$]($\mathit{args}_1$)$, \ldots ,$($\mathit{args}_l$)
}
```

where every argument in $(args\_1), \ldots, (args\_l)$ is a reference to one of the values $x_1, \ldots, x_k$. To integrate the current application into the block, first a value definition using a fresh name y_i is created for every argument in $e_1, \ldots, e_m$, which is initialised to e_i for positional arguments and to e'_i for named arguments of the form `$x_i=e'_i$`. Then, for every parameter which is not specified by the argument list, a value definition using a fresh name z_i is created, which is initialized using the method computing the default argument of this parameter.

Let *args* be a permutation of the generated names y_i and z_i such such that the position of each name matches the position of its corresponding parameter in the method type `($p_1:T_1 , \ldots , p_n:T_n$)$U$`. The final result of the transformation is a block of the form

```
{ val q = $p$
  val $x_1$ = expr$_1$
  $\ldots$
  val $x_l$ = expr$_k$
  val $y_1$ = $e_1$
  $\ldots$
  val $y_m$ = $e_m$
  val $z_1$ = $q.m\$default\$i[\mathit{targs}](\mathit{args}_1), \ldots ,(\mathit{args}_l)$
  $\ldots$
  val $z_d$ = $q.m\$default\$j[\mathit{targs}](\mathit{args}_1), \ldots ,(\mathit{args}_l)$
  q.$m$[$\mathit{targs}$]($\mathit{args}_1$)$, \ldots ,$($\mathit{args}_l$)($\mathit{args}$)
}
```

Signature Polymorphic Methods

For invocations of signature polymorphic methods of the target platform `$f$($e_1 , \ldots , e_m$)`, the invoked method has a different method type `($p_1$:$T_1 , \ldots , p_n$:$T_n$)$U$` at each call site. The parameter types `$T_ , \ldots , T_n$` are the types of the argument expressions `$e_1 , \ldots , e_m$` and `$U$` is the expected type at the call site. If the expected type is undefined then `$U$` is `scala.AnyRef`. The parameter names `$p_1 , \ldots , p_n$` are fresh.

Note

On the Java platform version 7 and later, the methods `invoke` and `invokeExact` in class `java.lang.invoke.MethodHandle` are signature polymorphic.

Method Values

```
SimpleExpr    ::=  SimpleExpr1 ‘_’
```

The expression `$e$ _` is well-formed if *e* is of method type or if *e* is a call-by-name parameter. If *e* is a method with parameters, `$e$ _` represents *e* converted to a function type by eta expansion. If *e* is a parameterless method or call-by-name parameter of type `=> $T$`, `$e$ _` represents the function of type `() => $T$`, which evaluates *e* when it is applied to the empty parameter list `()`.

Example

The method values in the left column are each equivalent to the eta-expanded expressions on the right.

placeholder syntax	eta-expansion
`math.sin _`	`x => math.sin(x)`
`math.pow _`	`(x1, x2) => math.pow(x1, x2)`
`val vs = 1 to 9; vs.fold _`	`(z) => (op) => vs.fold(z)(op)`
`(1 to 9).fold(z)_`	`{ val eta1 = 1 to 9; val eta2 = z; op => eta1.fold(eta2)(op) }`

placeholder syntax	eta-expansion
`Some(1).fold(??? : Int)_`	`{ val eta1 = Some(1); val eta2 = () => ???; op => eta1.fold(eta2())(op) }`

Note that a space is necessary between a method name and the trailing underscore because otherwise the underscore would be considered part of the name.

Type Applications

```
SimpleExpr    ::=  SimpleExpr TypeArgs
```

A *type application* `$e$[$T_1 , \ldots , T_n$]` instantiates a polymorphic value $e$ of type `[$a_1$ >: $L_1$ <: $U_1, \ldots , a_n$ >: $L_n$ <: $U_n$]$S$` with argument types `$T_1 , \ldots , T_n$`. Every argument type T_i must obey the corresponding bounds L_i and U_i. That is, for each $i = 1, \ldots, n$, we must have $\sigma L_i <: T_i <: \sigma U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$. The type of the application is σS.

If the function part $e$ is of some value type, the type application is taken to be equivalent to `$e$.apply[$T_1 , \ldots ,$ T$_n$]`, i.e. the application of an `apply` method defined by $e$.

Type applications can be omitted if local type inference can infer best type parameters for a polymorphic method from the types of the actual method arguments and the expected result type.

Tuples

```
SimpleExpr   ::=  ‘(’ [Exprs] ‘)’
```

A *tuple expression* `($e_1 , \ldots , e_n$)` is an alias for the class instance creation `scala.Tuple$n$($e_1 , \ldots , e_n$)`, where $n \geq 2$. The empty tuple `()` is the unique value of type `scala.Unit`.

Instance Creation Expressions

```
SimpleExpr     ::=  ‘new’ (ClassTemplate | TemplateBody)
```

A *simple instance creation expression* is of the form `new $c$` where $c$ is a constructor invocation. Let $T$ be the type of $c$. Then $T$ must denote a (a type instance of) a non-abstract subclass of `scala.AnyRef`. Furthermore, the *concrete self type* of the expression must conform to the self type of the class denoted by $T$. The concrete self type is normally $T$, except if the expression `new $c$` appears as the right hand side of a value definition

```
val $x$: $S$ = new $c$
```

(where the type annotation `: $S$` may be missing). In the latter case, the concrete self type of the expression is the compound type `$T$ with $x$.type`.

The expression is evaluated by creating a fresh object of type T which is initialized by evaluating c. The type of the expression is T.

A *general instance creation expression* is of the form `new $t$` for some class template t. Such an expression is equivalent to the block

```
{ class $a$ extends $t$; new $a$ }
```

where a is a fresh name of an *anonymous class* which is inaccessible to user programs.

There is also a shorthand form for creating values of structural types: If `{$D$}` is a class body, then `new {$D$}` is equivalent to the general instance creation expression `new AnyRef{$D$}`.

Example

Consider the following structural instance creation expression:

```
new { def getName() = "aaron" }
```

This is a shorthand for the general instance creation expression

```
new AnyRef{ def getName() = "aaron" }
```

The latter is in turn a shorthand for the block

```
{ class anon\$X extends AnyRef{ def getName() = "aaron" }; new anon\$X }
```

where `anon\$X` is some freshly created name.

Blocks

```
BlockExpr  ::=  '{' CaseClauses '}'
             |  '{' Block '}'
Block      ::=  BlockStat {semi BlockStat} [ResultExpr]
```

A *block expression* `{$s_1$; $\ldots$; $s_n$; $e\,$}` is constructed from a sequence of block statements $s_1, \ldots, s_n$ and a final expression e. The statement sequence may not contain two definitions or declarations that bind the same name in the same namespace. The final expression can be omitted, in which case the unit value `()` is assumed.

The expected type of the final expression e is the expected type of the block. The expected type of all preceding statements is undefined.

The type of a block `$s_1$; $\ldots$; $s_n$; $e$` is `$T$ forSome {$\,Q\,$}`, where T is the type of e and Q contains existential clauses for every value or type name which is free in T and which is defined locally in one of the statements $s_1, \ldots, s_n$. We say the existential clause *binds* the occurrence of the value or type name. Specifically,

- A locally defined type definition `type$\;t = T$` is bound by the existential clause `type$\;t >: T <: T$`. It is an error if t carries type parameters.
- A locally defined value definition `val$\;x: T = e$` is bound by the existential clause `val$\;x: T$`.
- A locally defined class definition `class$\;c$ extends$\;t$` is bound by the existential clause `type$\;c <: T$` where T is the least class type or refinement type which is a proper supertype of the type c. It is an error if c carries type parameters.
- A locally defined object definition `object$\;x\;$extends$\;t$` is bound by the existential clause `val$\;x: T$` where T is the least class type or refinement type which is a proper supertype of the type `$x$.type`.

Evaluation of the block entails evaluation of its statement sequence, followed by an evaluation of the final expression e, which defines the result of the block.

Example

Assuming a class `Ref[T](x: T)`, the block

```
{ class C extends B {$\ldots$} ; new Ref(new C) }
```

has the type `Ref[_1] forSome { type _1 <: B }`. The block

```
{ class C extends B {$\ldots$} ; new C }
```

simply has type `B`, because with the rules here the existentially quantified type `_1 forSome { type _1 <: B }` can be simplified to `B`.

Prefix, Infix, and Postfix Operations

```
PostfixExpr     ::=  InfixExpr [id [nl]]
InfixExpr       ::=  PrefixExpr
                  |  InfixExpr id [nl] InfixExpr
PrefixExpr      ::=  [‘-’ | ‘+’ | ‘!’ | ‘~’] SimpleExpr
```

Expressions can be constructed from operands and operators.

Prefix Operations

A prefix operation *op*; e consists of a prefix operator *op*, which must be one of the identifiers ‘`+`’, ‘`-`’, ‘`!`’ or ‘`~`’. The expression *op*; e is equivalent to the postfix method application `e.unary_$\mathit{op}$`.

Prefix operators are different from normal method applications in that their operand expression need not be atomic. For instance, the input sequence `-sin(x)` is read as `-(sin(x))`, whereas the method application `negate sin(x)` would be parsed as the application of the infix operator `sin` to the operands `negate` and `(x)`.

Postfix Operations

A postfix operator can be an arbitrary identifier. The postfix operation $e; op$ is interpreted as $e.op$.

Infix Operations

An infix operator can be an arbitrary identifier. Infix operators have precedence and associativity defined as follows:

The *precedence* of an infix operator is determined by the operator's first character. Characters are listed below in increasing order of precedence, with characters on the same line having the same precedence.

```
(all letters)
|
^
&
= !
< >
:
+ -
* / %
(all other special characters)
```

That is, operators starting with a letter have lowest precedence, followed by operators starting with '`|`', etc.

There's one exception to this rule, which concerns *assignment operators*. The precedence of an assignment operator is the same as the one of simple assignment `(=)`. That is, it is lower than the precedence of any other operator.

The *associativity* of an operator is determined by the operator's last character. Operators ending in a colon '`:`' are right-associative. All other operators are left-associative.

Precedence and associativity of operators determine the grouping of parts of an expression as follows.

- If there are several infix operations in an expression, then operators with higher precedence bind more closely than operators with lower precedence.
- If there are consecutive infix operations $e_0; op\_1; e_1; op\_2 \ldots op\_n; e_n$ with operators $op\_1, \ldots, op\_n$ of the same precedence, then all these operators must have the same associativity. If all operators are left-associative, the sequence is interpreted as $(\ldots(e_0; op\_1; e_1); op\_2 \ldots); op\_n; e_n$. Otherwise, if all operators are right-associative, the sequence is interpreted as $e_0; op\_1; (e_1; op\_2; (\ldots op\_n; e_n)\ldots)$.
- Postfix operators always have lower precedence than infix operators. E.g. $e_1; op\_1; e_2; op\_2$ is always equivalent to $(e_1; op\_1; e_2); op\_2$.

The right-hand operand of a left-associative operator may consist of several arguments enclosed in parentheses, e.g. $e; op; (e_1, \ldots, e_n)$. This expression is then interpreted as $e.op(e_1, \ldots, e_n)$.

A left-associative binary operation $e_1; op; e_2$ is interpreted as $e_1.op(e_2)$. If *op* is right-associative and its parameter is passed by name, the same operation is interpreted as $e_2.op(e_1)$. If *op* is right-associative and its parameter is passed by value, it is interpreted as `{ val $x$=$e_1$; $e_2$.$\mathit{op}$($x\,$) }`, where x is a fresh name.

Assignment Operators

An *assignment operator* is an operator symbol (syntax category `op` in Identifiers) that ends in an equals character "`=`", with the exception of operators for which one of the following conditions holds:

1. the operator also starts with an equals character, or
2. the operator is one of `(<=)`, `(>=)`, `(!=)`.

Assignment operators are treated specially in that they can be expanded to assignments if no other interpretation is valid.

Let's consider an assignment operator such as `+=` in an infix operation `$l$ += $r$`, where l, r are expressions. This operation can be re-interpreted as an operation which corresponds to the assignment

```
$l$ = $l$ + $r$
```

except that the operation's left-hand-side l is evaluated only once.

The re-interpretation occurs if the following two conditions are fulfilled.

1. The left-hand-side l does not have a member named `+=`, and also cannot be converted by an implicit conversion to a value with a member named `+=`.
2. The assignment `$l$ = $l$ + $r$` is type-correct. In particular this implies that l refers to a variable or object that can be assigned to, and that is convertible to a value with a member named `+`.

Typed Expressions

```
Expr1              ::=  PostfixExpr ':' CompoundType
```

The *typed expression* $e : T$ has type T. The type of expression e is expected to conform to T. The result of the expression is the value of e converted to type T.

Example

Here are examples of well-typed and ill-typed expressions.

```
1: Int               // legal, of type Int
1: Long              // legal, of type Long
// 1: string         // ***** illegal
```

Annotated Expressions

```
Expr1              ::=  PostfixExpr ‘:’ Annotation {Annotation}
```

An *annotated expression* `$e$: @$a_1$ $\ldots$ @$a_n$` attaches annotations $a_1, \ldots, a_n$ to the expression e.

Assignments

```
Expr1        ::=  [SimpleExpr ‘.’] id ‘=’ Expr
               |  SimpleExpr1 ArgumentExprs ‘=’ Expr
```

The interpretation of an assignment to a simple variable `$x$ = $e$` depends on the definition of x. If x denotes a mutable variable, then the assignment changes the current value of x to be the result of evaluating the expression e. The type of e is expected to conform to the type of x. If x is a parameterless method defined in some template, and the same template contains a setter method `$x$_=` as member, then the assignment `$x$ = $e$` is interpreted as the invocation `$x$_=($e\,$)` of that setter method. Analogously, an assignment `$f.x$ = $e$` to a parameterless method x is interpreted as the invocation `$f.x$_=($e\,$)`.

An assignment `$f$($\mathit{args}\,$) = $e$` with a method application to the left of the ‘=’ operator is interpreted as `$f.$update($\mathit{args}$, $e\,$)`, i.e. the invocation of an `update` method defined by f.

Example

Here are some assignment expressions and their equivalent expansions.

assignment	expansion
`x.f = e`	`x.f_=(e)`
`x.f() = e`	`x.f.update(e)`
`x.f(i) = e`	`x.f.update(i, e)`
`x.f(i, j) = e`	`x.f.update(i, j, e)`

Example Imperative Matrix Multiplication

Here is the usual imperative code for matrix multiplication.

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length, yss(0).length)
  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss(i)(k) * yss(k)(j)
        k += 1
      }
      zss(i)(j) = acc
      j += 1
    }
    i += 1
  }
  zss
}
```

Desugaring the array accesses and assignments yields the following expanded version:

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length, yss.apply(0).length)
  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss.apply(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss.apply(i).apply(k) * yss.apply(k).apply(j)
        k += 1
      }
      zss.apply(i).update(j, acc)
      j += 1
    }
    i += 1
  }
  zss
}
```

Conditional Expressions

```
Expr1         ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
```

The *conditional expression* `if ($e_1$) $e_2$ else $e_3$` chooses one of the values of e_2 and e_3, depending on the value of e_1. The condition e_1 is expected to conform to type `Boolean`. The then-part e_2 and the else-part e_3 are both expected to conform to the expected type of the conditional expression. The type of the conditional expression is the weak least upper bound of the types of e_2 and e_3. A semicolon preceding the `else` symbol of a conditional expression is ignored.

The conditional expression is evaluated by evaluating first e_1. If this evaluates to `true`, the result of evaluating e_2 is returned, otherwise the result of evaluating e_3 is returned.

A short form of the conditional expression eliminates the else-part. The conditional expression `if ($e_1$) $e_2$` is evaluated as if it was `if ($e_1$) $e_2$ else ()`.

While Loop Expressions

```
Expr1          ::=  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
```

The *while loop expression* `while ($e_1$) $e_2$` is typed and evaluated as if it was an application of `whileLoop ($e_1$) ($e_2$)` where the hypothetical method `whileLoop` is defined as follows.

```
def whileLoop(cond: => Boolean)(body: => Unit): Unit  =
  if (cond) { body ; whileLoop(cond)(body) } else {}
```

Do Loop Expressions

```
Expr1          ::=  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ‘)’
```

The *do loop expression* `do $e_1$ while ($e_2$)` is typed and evaluated as if it was the expression `($e_1$ ; while ($e_2$) $e_1$)`. A semicolon preceding the `while` symbol of a do loop expression is ignored.

For Comprehensions and For Loops

```
Expr1         ::=  ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’)
                       {nl} [‘yield’] Expr
Enumerators    ::=  Generator {semi Generator}
Generator    ::= Pattern1 ‘<-’ Expr {[semi] Guard | semi Pattern1 ‘=’ Expr}
Guard          ::=  ‘if’ PostfixExpr
```

A *for loop* `for ($\mathit{enums}\,$) $e$` executes expression e for each binding generated by the enumerators *enums*. A *for comprehension* `for ($\mathit{enums}\,$) yield $e$` evaluates expression e for each binding generated by the enumerators *enums* and collects the results. An enumerator sequence always starts with a generator; this can be followed by further generators, value definitions, or guards. A *generator* `$p$ <- $e$` produces bindings from an expression e which is matched in some way against pattern

p. A *value definition* `$p$ = $e$` binds the value name p (or several names in a pattern p) to the result of evaluating the expression e. A *guard* `if $e$` contains a boolean expression which restricts enumerated bindings. The precise meaning of generators and guards is defined by translation to invocations of four methods: `map`, `withFilter`, `flatMap`, and `foreach`. These methods can be implemented in different ways for different carrier types.

The translation scheme is as follows. In a first step, every generator `$p$ <- $e$`, where p is not irrefutable for the type of e, and p is some pattern other than a simple name or a name followed by a colon and a type, is replaced by

```
$p$ <- $e$.withFilter { case $p$ => true; case _ => false }
```

Then, the following rules are applied repeatedly until all comprehensions have been eliminated.

- A for comprehension `for ($p$ <- $e\,$) yield $e'$` is translated to `$e$.map { case $p$ => $e'$ }`.
- A for loop `for ($p$ <- $e\,$) $e'$` is translated to `$e$.foreach { case $p$ => $e'$ }`.
- A for comprehension

  ```
  for ($p$ <- $e$; $p'$ <- $e'; \ldots$) yield $e''$
  ```

  where `$\ldots$` is a (possibly empty) sequence of generators, definitions, or guards, is translated to

  ```
  $e$.flatMap { case $p$ => for ($p'$ <- $e'; \ldots$) yield $e''$ }
  ```

- A for loop

  ```
  for ($p$ <- $e$; $p'$ <- $e'; \ldots$) $e''$
  ```

  where `$\ldots$` is a (possibly empty) sequence of generators, definitions, or guards, is translated to

  ```
  $e$.foreach { case $p$ => for ($p'$ <- $e'; \ldots$) $e''$ }
  ```

- A generator `$p$ <- $e$` followed by a guard `if $g$` is translated to a single generator `$p$ <- $e$.withFilter(($x_1 , \ldots , x_n$) => $g\,$)` where $x_1, \ldots, x_n$ are the free variables of p.
- A generator `$p$ <- $e$` followed by a value definition `$p'$ = $e'$` is translated to the following generator of pairs of values, where x and x' are fresh names:

  ```
  ($p$, $p'$) <- for ($x @ p$ <- $e$) yield { val $x' @ p'$ = $e'$; ($x$, $x'$) }
  ```

Example

The following code produces all pairs of numbers between 1 and $n-1$ whose sums are prime.

```
for  { i <- 1 until n
       j <- 1 until i
```

```
       if isPrime(i+j)
} yield (i, j)
```

The for comprehension is translated to:

```
(1 until n)
  .flatMap {
     case i => (1 until i)
       .withFilter { j => isPrime(i+j) }
       .map { case j => (i, j) } }
```

Example

For comprehensions can be used to express vector and matrix algorithms concisely. For instance, here is a method to compute the transpose of a given matrix:

```
def transpose[A](xss: Array[Array[A]]) = {
  for (i <- Array.range(0, xss(0).length)) yield
    for (xs <- xss) yield xs(i)
}
```

Here is a method to compute the scalar product of two vectors:

```
def scalprod(xs: Array[Double], ys: Array[Double]) = {
  var acc = 0.0
  for ((x, y) <- xs zip ys) acc = acc + x * y
  acc
}
```

Finally, here is a method to compute the product of two matrices. Compare with the imperative version.

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val ysst = transpose(yss)
  for (xs <- xss) yield
    for (yst <- ysst) yield
      scalprod(xs, yst)
}
```

The code above makes use of the fact that `map`, `flatMap`, `withFilter`, and `foreach` are defined for instances of class `scala.Array`.

Return Expressions

```
Expr1      ::=  'return' [Expr]
```

A *return expression* `return $e$` must occur inside the body of some enclosing user defined method. The innermost enclosing method in a source program, m, must have an explicitly declared result type, and the type of e must conform to it.

The return expression evaluates the expression e and returns its value as the result of m. The evaluation of any statements or expressions following the return expression is omitted. The type of a return expression is `scala.Nothing`.

The expression e may be omitted. The return expression `return` is type-checked and evaluated as if it were `return ()`.

Returning from the method from within a nested function may be implemented by throwing and catching a `scala.runtime.NonLocalReturnException`. Any exception catches between the point of return and the enclosing methods might see and catch that exception. A key comparison makes sure that this exception is only caught by the method instance which is terminated by the return.

If the return expression is itself part of an anonymous function, it is possible that the enclosing method m has already returned before the return expression is executed. In that case, the thrown `scala.runtime.NonLocalReturnException` will not be caught, and will propagate up the call stack.

Throw Expressions

```
Expr1       ::=  ‘throw’ Expr
```

A *throw expression* `throw $e$` evaluates the expression e. The type of this expression must conform to `Throwable`. If e evaluates to an exception reference, evaluation is aborted with the thrown exception. If e evaluates to `null`, evaluation is instead aborted with a `NullPointerException`. If there is an active `try` expression which handles the thrown exception, evaluation resumes with the handler; otherwise the thread executing the `throw` is aborted. The type of a throw expression is `scala.Nothing`.

Try Expressions

```
Expr1 ::=  ‘try’ Expr [‘catch’ Expr] [‘finally’ Expr]
```

A *try expression* is of the form `try { $b$ } catch $h$` where the handler h is a pattern matching anonymous function

```
{ case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$ }
```

This expression is evaluated by evaluating the block b. If evaluation of b does not cause an exception to be thrown, the result of b is returned. Otherwise the handler h is applied to the thrown exception. If the handler contains a case matching the thrown exception, the first such case is invoked. If the handler contains no case matching the thrown exception, the exception is re-thrown.

Let pt be the expected type of the try expression. The block b is expected to conform to pt. The handler h is expected conform to type `scala.PartialFunction[scala.Throwable, $\mathit{pt}\,$]`. The type of the try expression is the weak least upper bound of the type of b and the result type of h.

A try expression `try { $b$ } finally $e$` evaluates the block b. If evaluation of b does not cause an exception to be thrown, the expression e is evaluated. If an exception is thrown during evaluation of e, the evaluation of the try expression is aborted with the thrown exception. If no exception is thrown during evaluation of e, the result of b is returned as the result of the try expression.

If an exception is thrown during evaluation of b, the finally block e is also evaluated. If another exception e is thrown during evaluation of e, evaluation of the try expression is aborted with the thrown exception. If no exception is thrown during evaluation of e, the original exception thrown in b is re-thrown once evaluation of e has completed. The block b is expected to conform to the expected type of the try expression. The finally expression e is expected to conform to type `Unit`.

A try expression `try { $b$ } catch $e_1$ finally $e_2$` is a shorthand for `try { try { $b$ } catch $e_1$ } finally $e_2$`.

Anonymous Functions

```
Expr            ::=  (Bindings | [‘implicit’] id | ‘_’) ‘=>’ Expr
ResultExpr    ::= (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block
Bindings        ::=  ‘(’ Binding {‘,’ Binding} ‘)’
Binding         ::=  (id | ‘_’) [‘:’ Type]
```

The anonymous function of arity n, `($x_1$: $T_1 , \ldots , x_n$: $T_n$) => e` maps parameters x_i of types T_i to a result given by expression e. The scope of each formal parameter x_i is e. Formal parameters must have pairwise distinct names.

In the case of a single untyped formal parameter, `($x\,$) => $e$` can be abbreviated to `$x$ => $e$`. If an anonymous function `($x$: $T\,$) => $e$` with a single typed parameter appears as the result expression of a block, it can be abbreviated to `$x$: $T$ => e`.

A formal parameter may also be a wildcard represented by an underscore `_`. In that case, a fresh name for the parameter is chosen arbitrarily.

A named parameter of an anonymous function may be optionally preceded by an `implicit` modifier. In that case the parameter is labeled `implicit`; however the parameter section itself does not count as an implicit parameter section. Hence, arguments to anonymous functions always have to be given explicitly.

Translation

If the expected type of the anonymous function is of the shape `scala.Function$n$[$S_1 , \ldots , S_n$, $R\,$]`, or can be SAM-converted to such a function type, the type `$T_i$` of a parameter `$x_i$` can be omitted, as far as `$S_i$` is defined in the expected type, and `$T_i$ = $S_i$` is assumed. Furthermore, the expected type when type checking $e$ is R.

If there is no expected type for the function literal, all formal parameter types `$T_i$` must be specified explicitly, and the expected type of $e$ is undefined. The type of the anonymous function is `scala.Function$n$[$T_1 , \ldots , T_n$, $R\,$]`, where R is the packed type of $e$. R must be equivalent to a type which does not refer to any of the formal parameters $x_i$.

The eventual run-time value of an anonymous function is determined by the expected type: - a subclass of one of the builtin function types, `scala.Function$n$[$S_1 , \ldots , S_n$, $R\,$]` (with $S_i$ and R fully defined), - a single-abstract-method (SAM) type; - `PartialFunction[$T$, $U$]` - some other type.

The standard anonymous function evaluates in the same way as the following instance creation expression:

```
new scala.Function$n$[$T_1 , \ldots , T_n$, $T$] {
  def apply($x_1$: $T_1 , \ldots , x_n$: $T_n$): $T$ = $e$
}
```

The same evaluation holds for a SAM type, except that the instantiated type is given by the SAM type, and the implemented method is the single abstract method member of this type.

The underlying platform may provide more efficient ways of constructing these instances, such as Java 8's `invokedynamic` bytecode and `LambdaMetaFactory` class.

When a `PartialFunction` is required, an additional member `isDefinedAt` is synthesized, which simply returns `true`. However, if the function literal has the shape `x => x match { $\ldots$ }`, then `isDefinedAt` is derived from the pattern match in the following way: each case from the match expression evaluates to `true`, and if there is no default case, a default case is added that evalutes to `false`. For more details on how that is implemented see "Pattern Matching Anonymous Functions".

Example

Examples of anonymous functions:

```
x => x                             // The identity function

f => g => x => f(g(x))             // Curried function composition
```

```
(x: Int,y: Int) => x + y           // A summation function

() => { count += 1; count }        // The function which takes an
                                   // empty parameter list $()$,
                                   // increments a non-local variable
                                   // `count' and returns the new value.

_ => 5                             // The function that ignores its argument
                                   // and always returns 5.
```

Placeholder Syntax for Anonymous Functions

```
SimpleExpr1  ::=  '_'
```

An expression (of syntactic category `Expr`) may contain embedded underscore symbols `_` at places where identifiers are legal. Such an expression represents an anonymous function where subsequent occurrences of underscores denote successive parameters.

Define an *underscore section* to be an expression of the form `_:$T$` where $T$ is a type, or else of the form `_`, provided the underscore does not appear as the expression part of a type ascription `_:$T$`.

An expression e of syntactic category `Expr` *binds* an underscore section u, if the following two conditions hold: (1) e properly contains u, and (2) there is no other expression of syntactic category `Expr` which is properly contained in e and which itself properly contains u.

If an expression e binds underscore sections $u_1, \ldots, u_n$, in this order, it is equivalent to the anonymous function `($u'_1$, ... $u'_n$) => $e'$` where each u'_i results from u_i by replacing the underscore with a fresh identifier and $e'$ results from e by replacing each underscore section u_i by u'_i.

Example

The anonymous functions in the left column use placeholder syntax. Each of these is equivalent to the anonymous function on its right.

`_ + 1`	`x => x + 1`
`_ * _`	`(x1, x2) => x1 * x2`
`(_: Int) * 2`	`(x: Int) => (x: Int) * 2`
`if (_) x else y`	`z => if (z) x else y`
`_.map(f)`	`x => x.map(f)`
`_.map(_ + 1)`	`x => x.map(y => y + 1)`

Constant Expressions

Constant expressions are expressions that the Scala compiler can evaluate to a constant. The definition of “constant expression” depends on the platform, but they include at least the expressions of the following forms:

- A literal of a value class, such as an integer
- A string literal
- A class constructed with `Predef.classOf`
- An element of an enumeration from the underlying platform
- A literal array, of the form `Array$(c_1 , \ldots , c_n)$`, where all of the c_i's are themselves constant expressions
- An identifier defined by a constant value definition.

Statements

```
BlockStat    ::=  Import
               |  {Annotation} [‘implicit’] [‘lazy’] Def
               |  {Annotation} {LocalModifier} TmplDef
               |  Expr1
               |
TemplateStat ::=  Import
               |  {Annotation} {Modifier} Def
               |  {Annotation} {Modifier} Dcl
               |  Expr
               |
```

Statements occur as parts of blocks and templates. A *statement* can be an import, a definition or an expression, or it can be empty. Statements used in the template of a class definition can also be declarations. An expression that is used as a statement can have an arbitrary value type. An expression statement e is evaluated by evaluating e and discarding the result of the evaluation.

Block statements may be definitions which bind local names in the block. The only modifier allowed in all block-local definitions is `implicit`. When prefixing a class or object definition, modifiers `abstract`, `final`, and `sealed` are also permitted.

Evaluation of a statement sequence entails evaluation of the statements in the order they are written.

Implicit Conversions

Implicit conversions can be applied to expressions whose type does not match their expected type, to qualifiers in selections, and to unapplied methods. The available implicit conversions are given in the next two sub-sections.

Value Conversions

The following seven implicit conversions can be applied to an expression *e* which has some value type *T* and which is type-checked with some expected type *pt*.

Static Overloading Resolution

If an expression denotes several possible members of a class, overloading resolution is applied to pick a unique member.

Type Instantiation

An expression *e* of polymorphic type

`[$a_1$ >: $L_1$ <: $U_1 , \ldots , a_n$ >: $L_n$ <: $U_n$]$T$`

which does not appear as the function part of a type application is converted to a type instance of *T* by determining with local type inference instance types `$T_1 , \ldots , T_n$` for the type variables `$a_1 , \ldots , a_n$` and implicitly embedding *e* in the type application `$e$[$T_1 , \ldots , T_n$]`.

Numeric Widening

If *e* has a primitive number type which weakly conforms to the expected type, it is widened to the expected type using one of the numeric conversion methods `toShort`, `toChar`, `toInt`, `toLong`, `toFloat`, `toDouble` defined here.

Numeric Literal Narrowing

If the expected type is `Byte`, `Short` or `Char`, and the expression *e* is an integer literal fitting in the range of that type, it is converted to the same literal in that type.

Value Discarding

If *e* has some value type and the expected type is `Unit`, *e* is converted to the expected type by embedding it in the term `{ $e$; () }`.

SAM conversion

An expression `(p1, ..., pN) => body` of function type `(T1, ..., TN) => T` is sam-convertible to the expected type `S` if the following holds: - the class `C` of `S` declares an abstract method `m` with signature `(p1: A1, ..., pN: AN): R`; - besides `m`, `C` must not declare or inherit any other deferred value members; - the method `m` must have a single argument list; - there must be a type `U` that is a subtype of `S`, so that the expression `new U { final def m(p1: A1, ..., pN: AN): R = body }` is well-typed (conforming to the expected type `S`); - for the purpose of scoping, `m` should be considered a static member (`U`'s members are not in scope in `body`); - `(A1, ..., AN) => R` is a subtype of `(T1, ..., TN) => T` (satisfying this condition drives type inference of unknown type parameters in `S`);

Note that a function literal that targets a SAM is not necessarily compiled to the above instance creation expression. This is platform-dependent.

It follows that: - if class `C` defines a constructor, it must be accessible and must define exactly one, empty, argument list; - class `C` cannot be `final` or `sealed` (for simplicity we ignore the possibility of SAM conversion in the same compilation unit as the sealed class); - `m` cannot be polymorphic; - it must be possible to derive a fully-defined type `U` from `S` by inferring any unknown type parameters of `C`.

Finally, we impose some implementation restrictions (these may be lifted in future releases): - `C` must not be nested or local (it must not capture its environment, as that results in a nonzero-argument constructor) - `C`'s constructor must not have an implicit argument list (this simplifies type inference); - `C` must not declare a self type (this simplifies type inference); - `C` must not be `@specialized`.

View Application

If none of the previous conversions applies, and e's type does not conform to the expected type pt, it is attempted to convert e to the expected type with a view.

Selection on `Dynamic`

If none of the previous conversions applies, and e is a prefix of a selection $e.x$, and e's type conforms to class `scala.Dynamic`, then the selection is rewritten according to the rules for dynamic member selection.

Method Conversions

The following four implicit conversions can be applied to methods which are not applied to some argument list.

Evaluation

A parameterless method m of type `=> $T$` is always converted to type T by evaluating the expression to which m is bound.

Implicit Application

If the method takes only implicit parameters, implicit arguments are passed following the rules here.

Eta Expansion

Otherwise, if the method is not a constructor, and the expected type pt is a function type, or, for methods of non-zero arity, a type sam-convertible to a function type, $(Ts') \Rightarrow T'$, eta-expansion is performed on the expression e.

(The exception for zero-arity methods is to avoid surprises due to unexpected sam conversion.)

Empty Application

Otherwise, if e has method type $()T$, it is implicitly applied to the empty argument list, yielding $e()$.

Overloading Resolution

If an identifier or selection e references several members of a class, the context of the reference is used to identify a unique member. The way this is done depends on whether or not e is used as a function. Let $\mathcal{A}$ be the set of members referenced by e.

Assume first that e appears as a function in an application, as in `$e$($e_1 , \ldots , e_m$)`.

One first determines the set of functions that is potentially applicable based on the *shape* of the arguments.

The *shape* of an argument expression e, written $\mathit{shape}(e)$, is a type that is defined as follows: - For a function expression `($p_1$: $T_1 , \ldots , p_n$: $T_n$) => $b$: (Any $, \ldots ,$ Any) => $\mathit{shape}(b)$`, where `Any` occurs n times in the argument type. - For a pattern-matching anonymous function definition `{ case ... }`: `PartialFunction[Any, Nothing]`. - For a named argument `$n$ = $e$`: $\mathit{shape}(e)$. - For all other expressions: `Nothing`.

Let $\mathcal{B}$ be the set of alternatives in $\mathcal{A}$ that are *applicable* to expressions $(e_1, \ldots, e_n)$ of types $(\mathit{shape}(e_1), \ldots, \mathit{shape}(e_n))$. If there is precisely one alternative in $\mathcal{B}$, that alternative is chosen.

Otherwise, let $S_1, \ldots, S_m$ be the list of types obtained by typing each argument as follows.

Normally, an argument is typed without an expected type, except when all alternatives explicitly specify the same parameter type for this argument (a missing parameter type, due to e.g. arity differences, is taken as `NoType`, thus resorting to no expected type), or when trying to propagate more type information to aid inference of higher-order function parameter types, as explained next.

The intuition for higher-order function parameter type inference is that all arguments must be of a function-like type (`PartialFunction`, `FunctionN` or some equivalent SAM type), which in turn must define the same set of higher-order argument types, so that they can safely be used as the expected type of a given argument of the overloaded method, without unduly ruling out any alternatives. The intent is not to steer overloading resolution, but to preserve enough type information to steer type inference of the arguments (a function literal or eta-expanded method) to this overloaded method.

Note that the expected type drives eta-expansion (not performed unless a function-like type is expected), as well as inference of omitted parameter types of function literals.

More precisely, an argument `$e_i$` is typed with an expected type that is derived from the `$i$`th argument type found in each alternative (call these `$T_{ij}$` for alternative `$j$` and argument position `$i$`) when all `$T_{ij}$` are function types `$(A_{1j},..., A_{nj}) => ?$` (or the equivalent `PartialFunction`, or SAM) of some arity `$n$`, and their argument types `$A_{kj}$` are identical across all overloads `$j$` for a given `$k$`. Then, the expected type for `$e_i$` is derived as follows: - we use `$PartialFunction[A_{1j},..., A_{nj}, ?]$` if for some overload `$j$`, `$T_{ij}$`'s type symbol is `PartialFunction`; - else, if for some `$j$`, `$T_{ij}$` is `FunctionN`, the expected type is `$FunctionN[A_{1j},..., A_{nj}, ?]$`; - else, if for all `$j$`, `$T_{ij}$` is a SAM type of the same class, defining argument types `$A_{1j},..., A_{nj}$` (and a potentially varying result type), the expected type encodes these argument types and the SAM class.

For every member m in $\mathcal{B}$ one determines whether it is applicable to expressions $(e_1, \dots, e_m)$ of types $S_1, \dots, S_m$.

It is an error if none of the members in $\mathcal{B}$ is applicable. If there is one single applicable alternative, that alternative is chosen. Otherwise, let $\mathcal{CC}$ be the set of applicable alternatives which don't employ any default argument in the application to $e_1, \dots, e_m$.

It is again an error if $\mathcal{CC}$ is empty. Otherwise, one chooses the *most specific* alternative among the alternatives in $\mathcal{CC}$, according to the following definition of being "as specific as", and "more specific than":

- A parameterized method m of type `($p_1:T_1, \ldots , p_n:T_n$)$U$` is *as specific as* some other member m' of type S if m' is applicable to arguments `($p_1 , \ldots , p_n$)` of types $T_1, \dots, Tlast$; if T_n denotes a repeated parameter (it has shape $T*$), and so does m''s last parameter, $Tlast$ is taken as T, otherwise is T_n used directly.
- A polymorphic method of type `[$a_1$ >: $L_1$ <: $U_1 , \ldots , a_n$ >: $L_n$ <: $U_n$]$T$` is as specific as some other member of type S if T is as specific as S under the assumption that for $i = 1, \dots, n$ each a_i is an abstract type name bounded from below by L_i and from above by U_i.
- A member of any other type is always as specific as a parameterized method or a polymorphic method.
- Given two members of types T and U which are neither parameterized nor polymorphic method types, the member of type T is as specific as the member of type U if the existential dual of T conforms to the existential dual of U. Here, the existential dual of a polymorphic type `[$a_1$ >: $L_1$ <: $U_1 , \ldots , a_n$ >: $L_n$ <: $U_n$]$T$` is `$T$ forSome { type $a_1$ >: $L_1$ <: $U_1$ $, \ldots ,$ type $a_n$ >: $L_n$ <: $U_n$}`. The existential dual of every other type is the type itself.

The *relative weight* of an alternative A over an alternative B is a number from

0 to 2, defined as the sum of

- 1 if A is as specific as B, 0 otherwise, and
- 1 if A is defined in a class or object which is derived from the class or object defining B, 0 otherwise.

A class or object C is *derived* from a class or object D if one of the following holds:

- C is a subclass of D, or
- C is a companion object of a class derived from D, or
- D is a companion object of a class from which C is derived.

An alternative A is *more specific* than an alternative B if the relative weight of A over B is greater than the relative weight of B over A.

It is an error if there is no alternative in $\mathcal{CC}$ which is more specific than all other alternatives in $\mathcal{CC}$.

Assume next that e appears as a function in a type application, as in `$e$[$\mathit{targs}\,$]`. Then all alternatives in $\mathcal{A}$ which take the same number of type parameters as there are type arguments in *targs* are chosen. It is an error if no such alternative exists. If there are several such alternatives, overloading resolution is applied again to the whole expression `$e$[$\mathit{targs}\,$]`.

Assume finally that e does not appear as a function in either an application or a type application. If an expected type is given, let $\mathcal{B}$ be the set of those alternatives in $\mathcal{A}$ which are compatible to it. Otherwise, let $\mathcal{B}$ be the same as $\mathcal{A}$. In this last case we choose the most specific alternative among all alternatives in $\mathcal{B}$. It is an error if there is no alternative in $\mathcal{B}$ which is more specific than all other alternatives in $\mathcal{B}$.

Example

Consider the following definitions:

```
class A extends B {}
def f(x: B, y: B) = $\ldots$
def f(x: A, y: B) = $\ldots$
val a: A
val b: B
```

Then the application `f(b, b)` refers to the first definition of f whereas the application `f(a, a)` refers to the second. Assume now we add a third overloaded definition

```
def f(x: B, y: A) = $\ldots$
```

Then the application `f(a, a)` is rejected for being ambiguous, since no most specific applicable signature exists.

Local Type Inference

Local type inference infers type arguments to be passed to expressions of polymorphic type. Say e is of type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n]T$ and no explicit type parameters are given.

Local type inference converts this expression to a type application `$e$[$T_1 , \ldots , T_n$]`. The choice of the type arguments $T_1, \ldots, T_n$ depends on the context in which the expression appears and on the expected type pt. There are three cases.

Case 1: Selections

If the expression appears as the prefix of a selection with a name x, then type inference is *deferred* to the whole expression $e.x$. That is, if $e.x$ has type S, it is now treated as having type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n]S$, and local type inference is applied in turn to infer type arguments for $a_1, \ldots, a_n$, using the context in which $e.x$ appears.

Case 2: Values

If the expression e appears as a value without being applied to value arguments, the type arguments are inferred by solving a constraint system which relates the expression's type T with the expected type pt. Without loss of generality we can assume that T is a value type; if it is a method type we apply eta-expansion to convert it to a function type. Solving means finding a substitution σ of types T_i for the type parameters a_i such that

- None of the inferred types T_i is a singleton type unless it is a singleton type corresponding to an object or a constant value definition or the corresponding bound U_i is a subtype of `scala.Singleton`.
- All type parameter bounds are respected, i.e. $\sigma L_i <: \sigma a_i$ and $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The expression's type conforms to the expected type, i.e. $\sigma T <: \sigma pt$.

It is a compile time error if no such substitution exists. If several substitutions exist, local-type inference will choose for each type variable a_i a minimal or maximal type T_i of the solution space. A *maximal* type T_i will be chosen if the type parameter a_i appears contravariantly in the type T of the expression. A *minimal* type T_i will be chosen in all other situations, i.e. if the variable appears covariantly, non-variantly or not at all in the type T. We call such a substitution an *optimal solution* of the given constraint system for the type T.

Case 3: Methods

The last case applies if the expression e appears in an application $e(d_1, \ldots, d_m)$. In that case T is a method type $(p_1 : R_1, \ldots, p_m : R_m)T'$. Without loss of generality we can assume that the result type T' is a value type; if it is a method type we apply eta-expansion to convert it to a function type. One computes first the types S_j of the argument expressions d_j, using two alternative schemes. Each argument expression d_j is typed first with the expected type R_j, in which the type parameters $a_1, \ldots, a_n$ are taken as type constants. If this fails, the argument d_j is typed instead with an expected type R'_j which results from R_j by replacing every type parameter in $a_1, \ldots, a_n$ with *undefined*.

In a second step, type arguments are inferred by solving a constraint system which relates the method's type with the expected type *pt* and the argument types $S_1, \ldots, S_m$. Solving the constraint system means finding a substitution σ of types T_i for the type parameters a_i such that

- None of the inferred types T_i is a singleton type unless it is a singleton type corresponding to an object or a constant value definition or the corresponding bound U_i is a subtype of `scala.Singleton`.
- All type parameter bounds are respected, i.e. $\sigma L_i <: \sigma a_i$ and $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The method's result type T' conforms to the expected type, i.e. $\sigma T' <: \sigma pt$.
- Each argument type weakly conforms to the corresponding formal parameter type, i.e. $\sigma S_j <:_w \sigma R_j$ for $j = 1, \ldots, m$.

It is a compile time error if no such substitution exists. If several solutions exist, an optimal one for the type T' is chosen.

All or parts of an expected type *pt* may be undefined. The rules for conformance are extended to this case by adding the rule that for any type T the following two statements are always true: $undefined <: T$ and $T <: undefined$

It is possible that no minimal or maximal solution for a type variable exists, in which case a compile-time error results. Because <: is a pre-order, it is also possible that a solution set has several optimal solutions for a type. In that case, a Scala compiler is free to pick any one of them.

Example

Consider the two methods:

```
def cons[A](x: A, xs: List[A]): List[A] = x :: xs
def nil[B]: List[B] = Nil
```

and the definition

```
val xs = cons(1, nil)
```

The application of `cons` is typed with an undefined expected type. This application is completed by local type inference to `cons[Int](1, nil)`. Here, one uses the following reasoning to infer the type argument `Int` for the type parameter `a`:

First, the argument expressions are typed. The first argument `1` has type `Int` whereas the second argument `nil` is itself polymorphic. One tries to type-check `nil` with an expected type `List[a]`. This leads to the constraint system

```
List[b?] <: List[a]
```

where we have labeled `b?` with a question mark to indicate that it is a variable in the constraint system. Because class `List` is covariant, the optimal solution of this constraint is

```
b = scala.Nothing
```

In a second step, one solves the following constraint system for the type parameter `a` of `cons`:

```
Int <: a?
List[scala.Nothing] <: List[a?]
List[a?] <: $\mathit{undefined}$
```

The optimal solution of this constraint system is

```
a = Int
```

so `Int` is the type inferred for `a`.

Example

Consider now the definition

```
val ys = cons("abc", xs)
```

where `xs` is defined of type `List[Int]` as before. In this case local type inference proceeds as follows.

First, the argument expressions are typed. The first argument `"abc"` has type `String`. The second argument `xs` is first tried to be typed with expected type `List[a]`. This fails, as `List[Int]` is not a subtype of `List[a]`. Therefore, the second strategy is tried; `xs` is now typed with expected type `List[$\mathit{undefined}$]`. This succeeds and yields the argument type `List[Int]`.

In a second step, one solves the following constraint system for the type parameter `a` of `cons`:

```
String <: a?
List[Int] <: List[a?]
List[a?] <: $\mathit{undefined}$
```

The optimal solution of this constraint system is

```
a = scala.Any
```

so `scala.Any` is the type inferred for `a`.

Eta Expansion

Eta-expansion converts an expression of method type to an equivalent expression of function type. It proceeds in two steps.

First, one identifies the maximal sub-expressions of e; let's say these are $e_1, \ldots, e_m$. For each of these, one creates a fresh name x_i. Let e' be the expression resulting from replacing every maximal subexpression e_i in e by the corresponding fresh name x_i. Second, one creates a fresh name y_i for every argument type T_i of the method ($i = 1, \ldots, n$). The result of eta-conversion is then:

```
{ val $x_1$ = $e_1$;
  $\ldots$
  val $x_m$ = $e_m$;
  ($y_1: T_1 , \ldots , y_n: T_n$) => $e'$($y_1 , \ldots , y_n$)
}
```

The behavior of call-by-name parameters is preserved under eta-expansion: the corresponding actual argument expression, a sub-expression of parameterless method type, is not evaluated in the expanded block.

Dynamic Member Selection

The standard Scala library defines a marker trait `scala.Dynamic`. Subclasses of this trait are able to intercept selections and applications on their instances by defining methods of the names `applyDynamic`, `applyDynamicNamed`, `selectDynamic`, and `updateDynamic`.

The following rewrites are performed, assuming e's type conforms to `scala.Dynamic`, and the original expression does not type check under the normal rules, as specified fully in the relevant subsection of implicit conversion:

- `e.m[Ti](xi)` becomes `e.applyDynamic[Ti]("m")(xi)`
- `e.m[Ti]` becomes `e.selectDynamic[Ti]("m")`
- `e.m = x` becomes `e.updateDynamic("m")(x)`

If any arguments are named in the application (one of the `xi` is of the shape `arg = x`), their name is preserved as the first component of the pair passed to `applyDynamicNamed` (for missing names, `""` is used):

- `e.m[Ti](argi = xi)` becomes `e.applyDynamicNamed[Ti]("m")(("argi", xi))`

Finally:

- `e.m(x) = y` becomes `e.selectDynamic("m").update(x, y)`

None of these methods are actually defined in the `scala.Dynamic`, so that users are free to define them with or without type parameters, or implicit arguments.