第十八章

类型推断

各种编译时分析都需要对未知的类型进行推理。其中最主要的是泛型方法适用性测试(§18.5.1)和泛型方法调用类型推断(§18.5.2)。通常，我们将关于未知类型的推理过程称为类型推断。

在较高的层次上，类型推断可以分解为三个过程:

- 规约接受关于表达式或类型的兼容性断言(称为约束公式)，并将其简化为推断变量的一组边界。通常，一个约束公式可以简化为其他约束公式，这些约束公式必须递归地简化。遵循一个过程来识别这些附加的约束公式，并最终通过一个边界集来表示条件，在这些条件下，推断类型的选择将使每个约束公式为真。
- 合并维护一组推断变量边界，确保在添加新边界时这些边界是一致的。 因为一个变量的边界有时会影响另一个变量的可能选择，这个过程传播这些相互依赖的变量之间的边界。
- 解析检查推断变量的边界，并确定与这些边界兼容的实例化。它还决定相互依赖的推断变量被解析的顺序。

这些过程相互作用密切:减少可以触发合并;合并可能导致进一步减少;解决可能导致进一步合并。

- §18.1 更准确地定义用作中间结果的概念和用来表示它们的符号。
- §18.2 详细描述了规约。
- §18.3 详细描述了合并。
- §18.4 详细描述了解析。
- §18.5 定义如何使用这些推断工具来解决某些编译时分析问题。

与 Java 语言规范的 Java SE 7 版相比，推断的重要更改包括:

- 添加对 lambda 表达式和方法引用作为方法调用参数的支持。
- 一般化，以多元表达式定义推断，在推断完成之前，可能不会有定义良好的类型。这对于改进嵌套泛型方法和菱形构造函数调用的推理具有显著的效果。
- 描述如何使用推断来处理通配符参数化的函数接口目标类型和最具体的方法分析。
- 明确调用适用性测试(只涉及调用参数)和调用类型推断(包含目标类型)之间的区别。

- 为了获得更好的结果，延迟所有推断变量的解析，即使是那些有下界的变量，直到调用类型推断。
- 改进相互依赖(或自依赖)变量的推断行为。
- 消除 bug 和潜在的混淆来源。这个修订更加仔细和精确地处理了特定转换上下文和子类型之间的区别，并通过并行处理相应的非推断关系来描述 reduction。当有意偏离非推断关系时，这些关系会被明确地标识出来。
- 为未来的发展奠定基础:推断的增强或新的应用程序将更容易集成到规范中。

18.1 概念和符号

本节定义了推断变量、约束公式和边界，这些术语将在本章中使用。它还提供了符号。

18.1.1 推断变量

推断变量是类型的元变量—也就是说，它们是允许对类型进行抽象推理的特殊名称。为了区别于类型变量，推断变量用希腊字母表示，主要是α。

术语“类型”在本章中被松散地用于包括包含推断变量的类类型语法。术语“正确类型”不包括提及推断变量的“类型”。 涉及到推断变量的断言是关于可以通过将每个推断变量替换为适当类型来生成的每个适当类型的断言。

18.1.2 约束公式

约束公式是兼容性或可能涉及推断变量的子类型的断言。公式可以采用下列形式之一:

- <Expression -> T>: 表达式在松散调用上下文中与类型 T 兼容(§5.3)。
- <S->T>：在松散的调用上下文中，类型 S 与类型 T 兼容(§5.3)。
- <S<:T>: 引用类型 S 是引用类型 T 的子类型(§4.10)。
- <S<=T>:类型参数 S 包含在类型参数 T 中(§4.5.1)。
- <S=T>: 类型 S 与类型 T 相同(§4.3.4)，或者类型参数 S 与类型参数 T 相同。
- <LambdaExpression->throws T>: LambdaExpression 体抛出的检查异常是由派生自 T 的函数类型的 throws 子句声明的。
- <MethodReference->throws T>：由引用方法抛出的检查异常是由派生自 T 的函数类型的 throws 子句声明的。

  约束公式示例:

  - 从 Collections.singleton("hi"), 我们有约束公式< "hi"->α>。通过 reduction, 这会变成约束公式: <String <：α>。
  - 从 Arrays.asList(1, 2.0), 我们有约束公式<1->α>和<2.0->α>。通过 reduction, 这会变成约束公式<int-> α> 和< double->α>, 和< Integer<：α>和<Double <：α>。
  - 从构造函数调用的目标类型 List<Thread> lt = new ArrayList<>(), 我们有约束公式<ArrayList< α>->List<Thread>>。通过 reduction, 这会变成约束公式<α<= Thread>，and then <α = Thread>.

18.1.3 边界

在推断过程中，维护推断变量的一组边界。边界有以下形式之一:

- S = T, 其中 S 或 T 中至少有一个是推断变量:S 与 T 相同。
- S <: T, S 或 T 中至少有一个是推断变量:S 是 T 的子类型。
- false: 不存在有效的推断变量选择。
- $G<\alpha_1,\cdots,\alpha_n> = capture(G<A_1,\cdots,A_n>)$: 变量$\alpha_1,\cdots,\alpha_n$ 表示对 $G<A_1,\cdots,A_n>$应用捕获转换的结果(§5.1.10) ($A_1,\cdots,A_n$ 可以是类型或通配符，可能提到推断变量)。
- throws α: 推断变量 α 出现在 throws 子句中。

如果应用替换后断言为真，则推断变量替换满足边界。假的边界永远不会被满足。

有些边界将推断变量与合适的类型联系起来。设 T 是一个合适的类型。给定 α = T 或 T = α 形式的边界，我们说 T 是 α 的实例化。类似地，给定形式为 α <： T 的一个边界，我们说 T 是 α 的适当上界，并且给定形式为 T <： α 的一个边界，我们说 T 是 α 的适当下界。

其他边界将两个推理变量或一个推理变量与包含推理变量的类型相关联。这种形式为 S=T 或 S <： T 的边界称为依赖关系。

形式为 $G<\alpha_1, ..., \alpha_n> = capture(G<A_1, ..., A_n>)$的边界表明 $\alpha_1, ..., \alpha_n$ 是捕获转换结果的占位符。这是必要的，因为捕获转换只能在适当的类型上执行，并且 $A_1, ..., A_n$ 中的推断变量可能尚未解析。

形式为 throws α 的边界纯粹是信息性的：它指示解析优化 α 的实例化，以便在可能的情况下，它不是检查的异常类型。

推断的一个重要中间结果是有界集。有时使用符号 true 来引用空边界集是很方便的；这只是出于方便，两者可以互换。

边界集的例子:

- { α = String } 包含单个边界, 实例化 α 为 String。
- { Integer <: α, Double <: α, α <: Object } 描述 α 的两个合适下界和一个合适上界。
- { α <： Iterable<?>, β <： Object, a <： List<β> } 描述了 a 和 β 各自的适当上限，以及它们之间的依赖关系。
- { } 不包含边界或依赖项，可以称为 true。
- { false } 表示不存在令人满意的实例化的事实。

当推断开始时，通常从类型参数声明 P_1，…，P_p 和相关推断变量 α_1，…，α_p 的列表中生成边界集。这样的边界集生成如下。对于每个 I (1 ≤ I ≤p):

- 如果 P_I 没有 TypeBound, 边界 α_I <: Object 出现在集合中。
- 否则，对于 TypeBound 中由&分隔的每个类型 T，边界 α_I <: $T[P_1:= \alpha_1, ..., P_p:= \alpha_p]$出现在

集合中; 如果这导致 α_i 没有合适的上界(只有依赖关系)，那么边界 α_i <: Object 也出现在集合中。

18.2 规约

规约是一个过程，通过该过程，一组约束公式（§18.1.2）被简化以产生边界集（§18.1.3）。

依次考虑每个约束公式。本节中的规则规定了如何将公式简化为以下一项或两项:

- 一个边界或边界集，它将与“当前”边界集合并。最初，当前边界集是空的。
- 进一步的约束公式，将递归简化。

当没有进一步的约束公式需要简化时，简化完成。

> 归约步骤的结果总是稳健的：如果推断变量实例化满足归约限制和边界，它也将满足原始限制。另一方面，归约不是完备性保持：可能存在满足原始约束但不满足归约约束或边界的推理变量实例化。这是由于算法的固有限制，以及避免过度复杂的愿望。一个影响是，有些表达式的类型参数推断无法找到解决方案，但如果程序员显式插入适当的类型，则可以很好地将其类型化。

18.2.1 表达式兼容性约束

形式为<Expression-> T>的约束公式规约如下:

- 如果 T 是合适的类型，则如果表达式在松散调用上下文中与 T 兼容（§5.3），则约束将规约为 true，否则为 false。
- 否则，如果表达式是 S 类型的独立表达式（§15.2），则约束规约为<S->T>。
- 否则，表达式为多元表达式（§15.2）。结果取决于表达式的形式:
  - 如果表达式是形式为(Expression')的带括号表达式，则约束规约为<Expression'->T>。
  - 如果表达式是类实例创建表达式或方法调用表达式，则约束规约为边界集 B_3，该边界集将用于确定表达式与目标类型 T 的兼容性，如§18.5.2.1 所定义。(对于类实例创建表达式，§15.9.3 中定义了用于推断的相应“方法”。)

    该边界集可能包含新的推断变量，以及这些新变量与 T 中的推断变量之间的依赖关系。
  - 如果表达式形式为 e_1?e_2:e_3 的条件表达式，约束规约为两个约束公式，< e_2->T> 和 <e_3->T>。
  - 如果表达式是 lambda 表达式或方法引用表达式，则在下面指定结果。
  - 如果表达式是具有结果表达式 e_1，…，e_n 的 switch 表达式，则约束将规约为 n 个约束公式，<e_1->T>，…，<e_n->T>。

> 通过将嵌套泛型方法调用作为多元表达式处理，我们改进了嵌套调用的推断行为。例如，以下内容在 Java SE 7 中是非法的，但在 Java SE 8 中是合法的:

```
ProcessBuilder b = new ProcessBuilder(Collections.emptyList());
  // ProcessBuilder's constructor expects a List<String>
```

当外部调用和嵌套调用都需要推断时，问题就更困难了。例如:

```
List<String> ls = new ArrayList<>(Collections.emptyList());
```

我们的方法是"提升"嵌套调用的推断边界(在 emptyList 的情况下是{α <: Object})到外部推断过程(在这种情况下，试图推断 β，其中构造函数用于类型 ArrayList<β>)。我们还推断出嵌套推断变量和外部推断变量之间的依赖关系(约束<List<α>->Collection<β>将规约为依赖 a=β)。这样，嵌套调用中的推断变量的解析可以等待，直到可以从外部调用推断出额外的信息(基于赋值目标，β=String)。

形式为<LambdaExpression->T>的约束公式, 其中 T 提到至少一个推断变量，规约为:

- 如果 T 不是一个函数式的接口类型(§9.8)，约束会规约为 false。
- 否则，设 T'为由 T 派生而来的地面目标类型，如§15.27.3 所述。如果§18.5.3 用于推导出参数化的函数式接口类型，则不执行 $F<A'_1, ..., A'_m>$是 $F<A_1, ..., A_m>$的子类型的测试（相反，它是用下面的约束公式断言的）。假定 lambda 表达式的目标函数类型为 T'的函数类型。则:
  - 如果找不到有效的函数类型，则约束规约为 false。
  - 否则，将如下断言 LambdaExpression 与目标函数类型的一致性:
    > 如果 lambda 参数的数量与函数类型的参数类型的数量不同，则约束规约为 false。
    > 如果 lambda 表达式是隐式类型的，并且函数类型的一个或多个参数类型不是合适的类型，则约束规约为 false。

      由于§18.5.1 中对隐式类型 lambda 表达式的处理以及§18.5.2.2 中对目标类型的替换，实际中从未出现这种情况。

    > 如果函数类型的结果为 void，并且 lambda 体既不是语句表达式，也不是 void 兼容块，则约束规约为 false。
    > 如果函数类型的结果不是 void，并且 lambda 体是一个不是值兼容的块，则约束规约为 false。
    > 否则，约束规约为以下所有约束公式:
      » 如果 lambda 参数有显式声明的类型 $F_1, ..., F_n$ 并且函数类型有参数类型 $G_1, \cdots, G_n$, 那么 (i) 对于所有 i ($1 \leqslant i \leqslant n$), $<F_i = G_i>$, 并且 (ii) <T' <: T>。
      » 如果函数类型的返回类型是(非 void)类型 R，则假定 lambda 的参数类型与函数类型的参数类型相同。则:
        • 如果 R 是合适的类型，并且 lambda 体或 lambda 体中的某些结果表达式在赋值上下文中与 R 不兼容，则为 false。
        • 否则，如果 R 不是合适的类型，那么 lambda 体的形式为 Expression，约束为 <Expression->R>; 或者 lambda 体是带有结果表达式 $e_1, ..., e_m$ 的块, 对所有 i ($1 \leqslant$

$i \leqslant m$), $<e_i$ ->R>。

从涉及 lambda 表达式的兼容性约束派生的关键信息是出现在目标函数类型的返回类型中的推断变量的边界集。这是至关重要的，因为函数式接口通常是泛型的，并且在这些类型上操作的许多方法也是泛型的。

在最简单的情况下，lambda 表达式可能只是为推断变量提供一个下界:

```
<T> List<T> makeThree(Factory<T> factory) { ... }
String s = makeThree(() -> "abc").get(2);
```

在更复杂的情况下，结果表达式可能是一个多元表达式——甚至可能是另一个 lambda 表达式——因此，在产生一个边界之前，推断变量可能要通过多个具有不同目标类型的约束公式传递。

本节中描述的大部分工作都在断言结果表达式之前;它的目的是派生 lambda 表达式的函数类型，并检查明显不符合兼容性的表达式。

我们不试图对出现在目标函数类型的 throws 子句中的推断变量产生边界。这是因为异常包含不是兼容性的一部分(§15.27.3)--尤其是它不能影响方法的适用性(§18.5.1)。然而，我们稍后确实得到了这些变量的边界，因为调用类型推断(§18.5.2.2)产生了异常包含约束公式(§18.2.5)。

注意，如果目标类型是一个推断变量，或者目标类型的参数类型包含推断变量，则生成 false。在调用类型推断(§18.5.2.2)期间，为了实例化这些推断变量，需要执行额外的替换，从而避免了这种情况。(换句话说，实际上，永远不会用这些形式之一的目标类型“调用”规约。)

最后，注意到§15.27.3 要求 lambda 表达式的结果表达式在赋值上下文中与目标类型的返回类型 R 兼容。如果 R 是一个合适的类型，例如 Byte 派生自 Function<α,Byte>，那么可分配性很容易测试，上面的规约就是这样做的。如果 R 不是一个合适的类型，比如派生自 Function<string, α>，那么我们做上面的简化假设，即松散调用兼容性将足够。赋值兼容和松散调用兼容之间的区别在于，只有赋值允许常量表达式的收窄，例如 Byte b = 100;。因此，我们的简化假设并不能保持完整性:给定目标返回类型α和整数字面量结果表达式 100，可以想象α可以实例化为 Byte，但规约实际上不会产生这样的边界。

形式为<MethodReference ->T>的约束公式, 其中 T 提到至少一个推断变量，规约为:

• 如果 T 不是一个函数式接口类型，或者 T 是一个没有函数类型的函数式接口类型(§9.9)，约束就会规约为 false。

• 否则，当目标为 T 时，如果方法引用不存在潜在的适用方法，则约束规约为 false。

• 否则，如果方法引用是精确的(§15.13.1), 则设 P_1, ..., P_n 为 T 的函数类型的参数类型, 设 F_1, ..., F_k 为可能适用的方法的参数类型。约束被规约为一组新的约束，如下:

- 在 n = k+1 的特例中, 类型 P_i 的参数充当调用的目标引用。方法引用表达式必须具有如下形式 ReferenceType :: [TypeArguments] Identifier。约束规约到 $<P_1$ <: ReferenceType>并且, 对所有 $i(2 \leqslant i \leqslant n)$, $<P_i$ ->F_{i-1}>。

  在其他情况下, n = k, 约束规约到, 对所有 $i (1 \leqslant i \leqslant n)$,< P_i -> F_i >。

- 如果函数类型的结果不是 void，则让 R 为其返回类型。然后，如果可能适用的编译时声明的结果是 void，则约束规约为 false。否则, 约束规约为< R'->R>, 其中，R'是将捕获转换(§5.1.10)应用于可能适用的编译时声明的返回类型的结果。

• 否则，如果方法引用是不精确的, 并且:

- 如果函数类型的一个或多个参数类型不是合适的类型，则约束规约为 false。

  由于§18.5.1 对方法引用的处理不精确，以及§18.5.2.2 对目标类型的替换，这种情况在实践中从未出现过。

- 否则，将执行对编译时声明的搜索，如§15.13.1 所述。如果没有方法引用的编译时声明，则约束会规约为 false。否则，将有一个编译时声明，并且:(让 R 是函数类型的结果)

  > 如果 R 是 void，约束就会规约为 true。

  > 否则，如果方法引用表达式省略了 TypeArguments，并且编译时声明是泛型方法，并且编译时声明的返回类型提到了至少一个方法的类型参数，那么:

  » 如果 R 提到函数类型的一个类型参数，约束就会规约为 false。

  在这种情况下，关于 R 的约束可能会导致推断变量被作用域外的类型变量绑定。由于使用超出作用域的类型变量实例化推断变量是没有意义的，所以我们宁愿避免这种情况，只要出现这种可能性就立即放弃。这种简化是不完整的。

  »如果 R 未提及函数类型的一个类型参数，则约束规约到边界集 B_3，该边界集将用于确定方法引用在以函数类型的返回类型为目标时的兼容性，如§18.5.2.1 所定义。 B_3 可以包含新的推断变量以及这些新变量与 T 中的推理变量之间的依赖关系。

  用于确定泛型引用方法的返回类型的策略遵循本节前面用于泛型方法调用的模式。这可能涉及将边界“提升”到外部上下文中，并推断两组推断变量之间的依赖关系。

  > 否则，假设 R'是将捕获转换（§5.1.10）应用于编译时声明的调用类型（§15.12.2.6）的返回类型的结果。如果 R'为 void，则约束规约为 false；否则，约束将规约为 <R'->R>。

18.2.2 类型兼容性约束

形式为 < S->T > 的约束公式规约如下:

• 如果 S 和 T 是合适的类型，则如果 S 与 T 在松散调用上下文中兼容（§5.3），则约束将规约为 true，否则为 false。

• 否则，如果 S 是一个原生类型，则 S'是将装箱转换（§5.1.7）应用于 S 的结果。然后约束规约为<S'->T>。

• 否则，如果 T 是原生类型，则将 T'设为对 T 应用装箱转换（§5.1.7）的结果。然后约束规约为<S = T'>。

• 否则, 如果 T 是形式为 G<T_1, ..., T_n>的参数化类型, 不存在形式为 G<...>的类型是 S 的超类型, 但是 G 的原始类型是 S 的超类型，然后约束规约为 true。

• 否则, 如果 T 是形式为 G<T_1, ..., T_n>[]k 的数组类型，不存在形式为 G<...>$[]^k$ 的类型是 S 的超类型, 但是原始类型 G$[]^k$ 是 S 的超类型, 然后约束规约为 true。(符号$[]^k$ 表示 k 维的数组类型。)

- 否则, 约束规约为<S <: T>。

第四种和第五种情况是未经检查的转换的隐含使用（§5.1.9）。这些，以及在第一种情况下使用未检查的转换，可能会导致编译时未检查的警告，并可能影响方法的调用类型（§15.12.2.6）。

T 到 T'的装箱不是完全保持的;例如，如果 T 是 long, S 可能被实例化为 Integer，它不是 Long 的子类型，但可以拆箱，然后扩展为 long。在大多数情况下，我们通过对推断变量返回类型进行特殊处理来避免这个问题，我们知道这些类型已经被限制为某些装箱的原生类型；参见§18.5.2.1。

类似地，在 T 不是参数化类型的情况下（例如，如果 T 是推断变量），未检查转换的处理牺牲了完整性。在这种情况下，通常不清楚未经检查的转换是否必要。由于未检查的转换会引入未检查的警告，因此推断倾向于避免这些警告，除非这是明显必要的。

18.2.3 子类型约束

形式为< S <: T> 的约束公式规约如下:

- 如果 S 和 T 是合适的类型，则如果 S 是 T 的子类型（§4.10），则该约束规约为 true，否则为 false。
- 否则, 如果 S 是 null 类型，约束规约为 true。
- 否则, 如果 T 是 null 类型，约束规约为 false。
- 否则, 如果 S 是推断变量α, 约束规约为边界 α <: T。
- 否则, 如果 T 是推断变量α, 约束规约为 边界 S <: α.
- 否则, 约束根据 T 的形式规约:
  - 如果 T 是参数化类或接口类型，或参数化类或接口类型的内部类类型（直接或间接）, 设 A_1, ..., A_n 为 T 的类型参数。在 S 的超类型中，用类型参数 B_1，…，B_n 标识对应的类或接口类型。如果不存在这种类型，约束规约为 false。否则, 约束规约为以下新的约束: 对所有 i (1 ⩽ i ⩽n), <B_i <= A_i>。
  - 如果 T 是任何其他类或接口类型，则如果 T 在 S 的超类型中，则约束规约为 true，否则为 false。
  - 如果 T 是数组类型 T' [], 然后，在作为数组类型的 S 的超类型中，确定了一个最具体的类型，S'[]（这可能是 S 本身）。如果不存在这样的数组类型, 约束规约为 false。否则:
    > 如果 S' 或 T' 都不是原生类型, 约束规约为<S' <: T'>。
    > 否则, 如果 S' 或 T' 是相同的原生类型，约束规约为 true，否则规约为 false。
  - 如果 T 是类型变量，有三种情况:
    > 如果 S 是 T 为元素的交集类型，则约束规约为 true。
    > 否则, 如果 T 有下界 B，约束规约为<S <: B>。

> 否则，约束规约为 false

- 如果 T 是交集类型 I_1 & ... & I_n, 约束规约为以下新的约束: 对所有 i ($1 \leqslant i \leqslant n$), < S <: I_i >。

形式为<S <= T>的约束公式, 其中 S 和 T 是类型参数 (§4.5.1), 规约如下:

- 如果 T 是一个类型:
  - 如果 S 是一个类型, 约束规约为<S = T>。
  - 如果 S 是一个通配符, 约束规约为 false。
- 如果 T 是形式为?的通配符，约束规约为 true。
- 如果 T 是形式为? extends T'的通配符:
  - 如果 S 是一个类型, 约束规约为< S <: T' >。
  - 如果 S 是形式为?的通配符, 约束规约为<Object <: T'>。
  - 如果 S 是形式为? extends S'的通配符, 约束规约为<S' <: T'>。
  - 如果 S 是形式为? super S'的通配符, 约束规约为<Object=T' >。
- 如果 T 是形式为? super T'的通配符:
  - 如果 S 是一个类型, 约束规约为< T' <: S>。
  - 如果 S 是形式为? super S'的通配符, 约束规约为< T' <: S' >。
  - 否则, 约束规约为 false。

18.2.4 类型相等约束

形式为 <S = T>的约束公式, 其中 S 和 T 是类型, 规约如下:

- 如果 S 和 T 是合适的类型，如果 S 与 T 相同，则约束规约为 true(§4.3.4)，否则为 false。
- 否则, 如果 S 或 T 是 null 类型, 约束规约为 false。
- 否则, 如果 S 是推断变量 α, 并且 T 不是原生类型, 约束规约为边界 α = T。
- 否则, 如果 T 是推断变量 α, 并且 S 不是原生类型, 约束规约为边界 S = α。
- 否则, 如果 S 和 T 是具有相同擦除的类或接口类型, 其中 S 有类型参数 B_1, ..., B_n，并且 T 有类型参数 A_1, ..., A_n, 约束规约为以下新的约束: 对所有 i ($1 \leqslant i \leqslant n$), <$B_i$ = A_i>。
- 否则, 如果 S 和 T 是数组类型 S'[]和 T'[], 约束规约为< S' = T' >。
- 否则, 如果 S 和 T 是交集类型, 建立了 S 的元素和 T 的元素之间的对应关系。如果 S_i 和 T_j 是相同的类型，或者都是相同泛型类或接口的参数化，或者都是数组类型，S 的元素 S_i 对应于 T 的元素 T_i。

  如果 S 中的每个元素恰好对应 T 中的一个元素，反之亦然，则约束规约为以下新的约束:

对于 S 的每个元素 S_i和对应的 T 的元素 T_j，$< S_i = T_j >$。如果不是这样, 约束规约为 false。

该规则不允许推断变量直接作为交集类型的元素出现(而不是嵌套在参数化类型中)。由于对类型参数声明的限制(§4.4)，这样的交集类型在实践中不会出现。

- 否则, 约束规约为 false。

形式为< S = T>的约束公式, 其中 S 和 T 是类型参数 (§4.5.1), 规约如下:

- 如果 S 和 T 是类型, 如上所述规约了约束。
- 如果 S 的形式为?，并且 T 的形式为 ?，约束规约为 true。
- 如果 S 的形式为?，并且 T 的形式为? extends T', 约束规约为 <Object = T' >。
- 如果 S 的形式为? extends S' ，并且 T 的形式为?, 约束规约为< S' = Object>。
- 如果 S 的形式为? extends S' ，并且 T 的形式为? extends T', 约束规约为 < S' = T' >。
- 如果 S 的形式为? super S' ，并且 T 的形式为? super T', 约束规约为< S' = T' >。
- 否则, 约束规约为 false。

18.2.5 检查异常的约束

形式为<*LambdaExpression* ->*throws T*>的约束公式规约如下:

- 如果 T 不是函数式接口类型(§9.8), 约束规约为 *false*。
- 否则，让 lambda 表达式的目标函数类型按照§15.27.3 中的规定确定。如果找不到有效的函数类型，则约束规约为 false。
- 否则，如果 lambda 表达式是隐式类型的，并且函数类型的一个或多个参数类型不是正确类型，则约束会规约为 false。

  由于§18.5.2.2 中对目标类型进行了替换，这种情况在实践中从未出现过。

- 否则，如果函数类型的返回类型既不是 void 也不是正确类型，则约束会规约为 false。

  由于§18.5.2.2 中对目标类型进行了替换，这种情况在实践中从未出现过。

- 否则，设 E_1，…， E_n是函数类型的 throws 子句中的不正确类型。如果 lambda 表达式是隐式类型的，则让其参数类型为函数类型的参数类型。如果 lambda 体是多元表达式或包含多元结果表达式的块，则让目标返回类型为函数类型的返回类型。让 X_1,…， X_m 是 lambda 体可以抛出的检查异常类型(§11.2)。然后有两种情况:
  - 如果 n = 0 (函数类型的 throws 子句只包含合适的类型), 那么如果存在某个 i (1 ≤ i ≤m) 使得 Xi 不是 throws 子句中任何适当类型的子类型, 约束规约为 false; 否则, 约束规约为 true。
  - 如果 n > 0, 约束规约为一组子类型约束: 对所有 i (1 ≤ i ≤ m), 如果 X_i不是 throws 子句中任何适当类型的子类型, 则约束包括, 对所有 j (1 ≤ j ≤n), $<X_i <: E_j>$。此外, 对

所有 j (1 ⩽ j ⩽n), 约束规约为抛出 E_j 的边界。

形式为<MethodReference ->throws T>的约束公式规约如下:

• 如果 T 不是一个函数式接口类型，或者 T 是一个函数式接口类型，但没有函数类型(§9.9), 约束会规约为 false。

• 否则，让方法引用表达式的目标函数类型为 T 的函数类型。如果方法引用是不精确的(§15.13.1)，并且函数类型的一个或多个参数类型不是正确类型，则约束规约为 false。

• 否则，如果方法引用不精确，并且函数类型的结果既不是 void 也不是正确类型，约束就会规约为 false。

• 否则，设 E_1，…， E_n 是函数类型的 throws 子句中的不正确类型。让 X_1,…， X_m 是方法引用的编译时声明(§15.13.2)的调用类型的 throws 子句中的检查异常(从函数类型的参数类型和返回类型派生)。然后有两种情况:

- 如果 n = 0 (函数类型的 throws 子句只包含合适的类型), 那么如果存在某个 i (1 ⩽ i ⩽ m)使得 X_i 不是 throws 子句中任何适当类型的子类型, 约束规约为 false; 否则, 约束规约为 true。
- 如果 n > 0, 约束规约为一组子类型约束: 对所有 i (1 ⩽ i ⩽ m), 如果 X_i 不是 throws 子句中任何适当类型的子类型, 则约束包括, 对所有 j (1 ⩽ j ⩽ n), $< X_i <: E_j >$。此外, 对所有 j (1 ⩽ j ⩽ n), 约束规约为抛出 E_j 的边界。

检查异常的约束与返回类型的约束分开处理，因为返回类型的兼容性会影响方法的适用性(§18.5.1)，而异常只会在重载解析完成后影响调用类型(§18.5.2)。这可以通过在 lambda 表达式兼容性的定义中包含异常兼容性来简化(§15.27.3)，但这可能会导致令人惊讶的情况，即由显式类型的 lambda 体引发的异常会改变重载解析。

在(i)知道 lambda 的参数类型和(ii)知道主体中结果表达式的目标类型之前，无法确定 lambda 体引发的异常。(第二个要求是考虑泛型方法调用，例如，在返回类型和 throws 子句中出现相同的类型参数。) 因此，我们要求从目标类型 T 派生的这两个类型都是合适的类型。

一个结果是，从其他 lambda 表达式返回的 lambda 表达式不能从它们抛出的异常生成约束。这些约束只能从顶级 lambda 表达式生成。

请注意，对于函数类型的 throws 子句中出现多个推断变量的情况，处理是不完整的。 任何一个变量本身都可以满足声明每个检查异常的约束，但我们不能确定是哪个变量。所以，为了可预测性，我们对它们都进行了限制。

18.3 合并

在推断过程中，随着边界集的生成和增长，有可能根据原边界的断言推断出新的边界。合并的过程识别这些新的边界并将它们添加到边界集。

合并可以在两种情况下发生。一种情况是，边界集包含互补的边界对;这意味着新的约束公式，如§18.3.1 所述。另一种情况是，边界集包含一个涉及捕获转换的边界;这意味着新的边界，也可能意味着新的约束公式，如§18.3.2 所述。在这两种情况下，任何新的约束公式

都被规约，任何新的边界都被添加到边界集。这可能会引发进一步合并;最终，集合将到达一个固定的点，并且无法推断出进一步的边界。

如果合并一个边界集已经达到一个固定的点，并且该集合不包含边界 false，那么该边界集具有以下属性:

- 对于每一个推断变量的适当下界 L 和适当上界 U 的组合, *L* <: *U*.
- 如果一个边界所提到的每个推断变量都有一个实例化，那么相应的替换就满足了这个边界。
- 给定一个依赖 α= β, α的每一个边界都匹配 β 的一个边界，反之亦然。
- 给定一个依赖 α <: β，α的每个下界是 β 的一个下界，β 的每个上界是 α 的一个上界。

> 合并达到一个固定点的断言稍微简化了问题。基于 Kennedy 和 Pierce 的著作《关于带有方差的名义子类型的可判性》，这一性质可以通过论证可能出现在有界集合中的类型集是有限的来证明。这一论点基于两个假设:
>
> - 规约子类型约束时，不会生成新的捕获变量(§18.2.3)。
> - 不追求扩展的继承路径。
>
> 该规范目前不能保证这些属性(在规约子类型约束时，它对通配符的处理是不精确的，并且不检测可扩展的继承路径)，但在未来的版本中可能会这样做。(这不是一个新问题:Java 子类型算法也有不终止的风险。)

18.3.1 互补边界对

(在本节中，S 和 T 是推断变量或类型，U 是适当的类型。为简洁起见，α = T 形式的边界也可以匹配 T = α 形式的边界。)

当一个边界集包含一对匹配以下规则之一的边界时，就隐含了一个新的约束公式:

- α = S 和α = T 意味着 < S = T>
- α = S 和 α <: T 意味着< S <: T>
- α = S 和 T <: α 意味着 < T <: S>
- S <: α 和 α<: T 意味着 < S <: T >
- α = U 和 S = T 意味着 <S [α:=U] =T[α:=U]>
- α = U 和 S <: T 意味着 < S [α:=U]<:T[α:=U]>

当边界集包含一对边界 α <: S 和 α <: T, 存在形式为 $G<S_1, ..., S_n>$ 的 S 的超类型，以及形式为 $G<T_1, ..., T_n>$ 的 T 的超类型(对于 G 的一些泛型类或接口), 那么对于所有 i ($1 \leqslant i \leqslant n$), 如果 S_i 和 T_i 是类型 (不是通配符), 隐含约束公式$< S_i = T_i >$。

18.3.2 涉及捕获转换的边界

当一个边界集包含形式为 $G <\alpha_1, \cdots, \alpha_n > = capture(G < A_1, \cdots, A_n>)$的边界, 隐含了新的

边界和新的约束公式，如下所示。

让 P_1,…， P_n 表示 G 的类型参数，让 B_1，…， B_n 表示这些类型参数的边界。让 θ 代表替换 $[P_1:=\alpha_1$，…， $P_n:=\alpha_n]$。设 R 不是推断变量(但不一定是正确类型)。

α_1, ..., α_n 上一组隐含的边界, 它是根据§18.1.3 中规定的 P_1，…， P_n 的声明边界生成的。

此外, 对所有 i ($1 \leqslant i \leqslant n$):

- 如果 A_i 不是通配符, 则隐含边界 $\alpha_i = A_i$。
- 如果 A_i 是形式为? 的通配符:
  - α_i = R 隐含边界为 false
  - α_i <: R 隐含约束公式 <B_i θ <: R>
  - R<: α_i 隐含边界为 false
- 如果 A_i 是形式为? extends T 的通配符:
  - α_i = R 隐含边界为 false
  - 如果 B_i 是 Object, 那么 α_i <: R 隐含约束公式 <T <: R>
  - 如果 T 是 Object, 那么 α_i <: R 隐含约束公式 <B_i θ <: R>
  - R <: α_i 隐含边界为 false
- 如果 A_i 是形式为? super T 的通配符:
  - α_i = R 隐含边界为 false
  - α_i <: R 隐含约束公式<B_i θ <: R>
  - R <: α_i 隐含约束公式 <R <: T>

18.4 解析

给定一个不包含边界 false 的边界集，可以解析边界集中提到的推断变量的一个子集。这意味着可以为每个推断变量添加一个满意的实例化，直到所有请求的变量都有实例化。

边界集中的依赖关系可能要求按特定顺序解析变量，或者解析其他变量。依赖项指定如下:

- 给定以下形式之一的边界，其中 T 是推断变量β或提及β的类型:
  - α = T
  - α <: T
  - T = α
  - T<: α

如果 α 出现在另一个形式为 G<..., α, ...> = capture(G<...>)的边界的左边, 则 β 依赖于 α 的解析。否则，α 依赖于 β 的解析。

- 推断变量 α 出现在形式为 G<..., α, ...> = capture(G<...>)的边界的左边，取决于该边界中提到的每个其他推断变量的解析（在=号的两侧）。
- 如果存在一个推断变量 γ，使得 α 取决于 γ 的解析，γ 取决于 β 的解析，则推断变量 α 取决于推断变量 β 的解析。
- 推断变量 α 取决于其自身的解析。

给定一组要解析的推断变量，设 V 是该集合和该集合中至少一个变量的解析所依赖的所有变量的并集。

如果 V 中的每个变量都有实例化，则解析成功，此过程终止。

否则, 设{ α1, ..., αn } 为 V 中未实例化变量的非空子集以至于 (i) 对所有 i (1 ⩽ i ⩽ n), 如果 αi 取决于变量 β 的解析, 则要么 β 有一个实例化或者存在某个 j 使得 β = αj; 并且(ii) 不存在具有这个属性的{ α1, ..., αn }的合适的非空子集。解析通过基于边界集中的边界为 α1, ..., αn 中的每一个生成实例化来进行:

- 如果对于所有 i (1 ⩽ i ⩽ n)，边界集不包含形式为 G<..., α_i,...> = capture(G<...>)的边界，然后，为每个 α_i 定义候选实例化 T_i:
  - 如果 α_i 有一个或多个合适的下界 L_1，⋯,L_k, 那么 T_i = lub(L_1,..., L_k) (§4.10.4)。
  - 否则, 如果边界集包含 throws α_i, 并且 α_i 的每一个合适的上界是 RuntimeException 的子类型, 那么 Ti = RuntimeException。
  - 否则, 当 α_i 有合适的上界 U_1, ..., U_k, T_i = glb(U_1, ..., U_k) (§5.1.10)。

  边界 $\alpha_1 = T_1$, ..., $\alpha_n = T_n$ 与当前边界集合并。

  如果结果不包含边界 false, 则结果变成新的边界集，并且如上所述，通过选择一组新的变量来实例化(如果需要)来进行解析。

  否则, 结果包含边界 false, 所以通过执行以下步骤进行第二次实例化{ a_1, ..., a_n }的尝试。
- 如果对某些 i (1 ⩽ i ⩽ n)，边界集包含形式为 G<..., a_i, ...> = capture(G<...>)的边界，或者;

  如果在上述步骤中产生的边界集包含边界 false;

  然后设 Y_1，...，Y_n 是边界如下的新类型变量:
  - 对所有 i (1 ⩽ i ⩽n), 如果 α_i 有一个或多个合适的下界 L_1, ..., L_k, 则设 Y_i 的下界为 lub(L_1, ..., L_k); 如果没有, 那么 Y_i 没有下界。
  - 对所有 i (1 ⩽ i ⩽ n), 如果 α_i 有上界 U_1，⋯，U_k, 则设 Y_i 的上界为 glb(U_1 θ, ..., U_k θ), 其中 θ 是替换 [α_1:=Y_1，⋯，α_n:=Y_n]。

如果类型变量 Y_1, ..., Y_n 没有格式良好的边界 (也就是说，下界不是上界的子类型，或者交集类型不一致), 那么解析失败了。

否则, 对所有 i ($1 \leqslant i \leqslant n$), 所有形式为 G<..., α_i, ...> = capture(G<...>) 的边界从当前边界集中被移除，并且合并边界 $\alpha_1 = Y_1$, ..., $\alpha_n = Y_n$。

如果结果不包含边界 false, 则结果变成新的边界集，并且如上所述，通过选择要实例化的一组新变量（如果需要）进行解析。

否则，结果包含边界 false，解析失败。

> 实例化推断变量的第一种方法是从该变量的边界导出实例化。然而，有时，复杂的依赖性意味着结果不在变量的边界内。在这种情况下，执行不同的实例化方法，类似于捕获转换（§5.1.10）：引入新类型变量，边界从推断变量的边界导出。请注意，这些"捕获"变量的下限仅使用适当的类型计算：这对于避免对未实例化的类型变量执行类型计算非常重要。

18.5 推断的使用

使用上面定义的推断过程，在编译时执行以下分析。

18.5.1 适用性推断的调用

给定不提供显式类型参数的方法调用，确定潜在适用的泛型方法 m 是否适用的过程如下：

- $P_1,\cdots,P_p$ ($p \geqslant 1$) 是 m 的类型参数, 设 $\alpha_1,\cdots,\alpha_p$ 为推断变量, 设 θ 为替换 [P_1:=α_1，⋯，P_p:=α_p].
- 初始边界集 B_0 是由 P_1, ..., P_p 声明的边界产生的，如 §18.1.3 所述。
- 对所有 i ($1 \leqslant i \leqslant p$), 如果 P_i 出现在 m 的 throws 子句, 则隐含着边界 throws α_i。这些边界，如果有的话，与 B_0 合并产生一个新的边界集 B_1。
- 得到一组约束公式 C:

  设 F_1, ..., F_n 为 m 的形式参数类型，设 e_1, ..., e_k 为调用的实际参数表达式。则：

  - 通过严格调用来测试适用性:

    如果 $k \neq n$, 或者如果存在 i ($1 \leqslant i \leqslant n$) 使得 e_i 与适用性相关(§15.12.2.2)并且要么(i) e_i 是一个原生类型的单独的表达式，但是 F_i 是引用类型, 或者(ii) F_i 是原生类型但是 e_i 不是一个原生类型的单独的表达式; 那么该方法就不适用，不需要进行推理。

    否则, C 包含, 对所有 i ($1 \leqslant i \leqslant k$) 其中 e_i 与适用性有关, <e_i ->F_i θ>。

  - 通过松散调用测试适用性:

    如果 $k \neq n$, 这种方法是不适用的，没有必要进行推断。

    否则, C 包含, 对所有 i ($1 \leqslant i \leqslant k$) 其中 e_i 与适用性有关,<e_i -> Fi θ>.

  - 通过可变调用来测试适用性:

    设 F'_1, ..., F'_k 是 m 的前 k 个可变参数类型(§15.12.2.4)。C 包括, 对所有 i ($1 \leqslant i \leqslant k$) 其中 e_i

与适用性有关，$<e_i -> F'_i\ \theta>$。

- C 被规约(§18.2)，并且所得到的边界与 B_1 合并，以产生新的边界集 B_2。
- 最后，方法 m 适用于 B2 不包含边界 false 和 B2 中所有推断变量的解析成功(§18.4)。

考虑下面的方法调用和赋值:

```
List<Number> ln = Arrays.asList(1, 2.0);
```

如§15.12 所述，必须确定调用的最具体适用方法。唯一可能适用的方法（§15.12.2.1）声明如下:

```
public static <T> List<T> asList(T... a)
```

简单地说（由于其算术性），该方法既不适用于严格调用（§15.12.2.2），也不适用于松散调用（§14.12.3）。但由于没有其他候选者，在第三阶段，通过可变参数调用检查该方法的适用性。

初始边界集 B 是单个推断变量 α 的平凡上界:

```
{α <： Object}
```

初始约束公式集如下:

$\{ <1 -> \alpha>, <2.0 -> \alpha> \}$

这些被规约为一个新的边界集 B_1:

$\{ \alpha <: \text{Object}, \text{Integer} <: \alpha, \text{Double} <: \alpha \}$

然后，为了测试该方法是否适用，我们尝试解析这些边界。我们成功生成了相当复杂的实例化。

$\alpha = \text{Number \& Comparable<? extends Number \& Comparable<?>>}$

因此，我们证明了该方法是适用的；由于不存在其他候选方法，因此它是最具体的适用方法。尽管如此，方法调用的类型及其与赋值中目标类型的兼容性直到可以发生进一步的推断时才确定，如下一节所述。

18.5.2 调用类型推断

给定一个不提供显式类型参数的方法调用表达式，以及对应的最具体适用的泛型方法 m，推断所选方法的调用类型（§15.12.2.6）的过程可能需要解决额外的约束，以断言与目标类型的兼容性，并断言方法调用参数表达式的有效性。

重要的是要注意，在查找方法调用的类型时涉及多“轮”推断。例如，这是必要的，以允许目标类型影响调用的类型，而不允许其影响适用方法的选择。第一轮(§18.5.1)产生一个边界集，并测试一个解析的存在，但不承诺该解析。随后的轮询规约了额外的约束，直到最后的解析步骤确定表达式的“真实”类型。

18.5.2.1 多元方法调用兼容性

如果方法调用表达式是一个多元表达式(§15.12)，它与目标类型 T 的兼容性如下所示。

如果方法调用表达式出现在严格调用上下文中，并且 T 是原生类型，则表达式与 T 不兼容。

否则:

- 为了证明 m 在§18.5.1 中是适用的，让 B_2 成为规约产生的边界集。

(虽然在§18.5.1 中有必要证明 B_2 中的推断变量可以被解析，为了建立适用性，这个解析

步骤产生的实例不被认为是 B_2 的一部分。)

- 设 B_3 是由 B_2 导出的边界集，如下所示。

  设 R 为 m 的返回类型，设 θ 为定义在§18.5.1 中的替换[P_1:=α_1，…，P_p:=α_p]，用推断变量替换 m 的类型参数，并设 T 为调用的目标类型。则:

  - 如果在§18.5.1 的约束集规约过程中，未检查的转换是该方法适用的必要条件，约束公式<|R|->T> 被规约并与 B2 合并。
  - 否则，如果 R θ 是一个参数化类型 G<A1, ..., An>, A1, ..., An 之一是通配符，那么，对于新推断变量 β1，…，βn，约束公式<G<β1，…，βn>-> T>被规约和合并，与边界 G<β1，…， βn> = capture(G<A1, ..., An>)和 B2 一起。
  - 否则,如果 R θ 是推断变量 α，下面其中一个是正确的:

    > T 是一个引用类型，但不是通配符参数化类型，并且 (i) B_2 包含形式为 α = S 或 S <: α 之一的边界，在这里 S 是通配符参数化类型，或 (ii) B_2 包含形式为 S_1 <: α 和 S_2 <: α 的两个边界，其中 S_1 和 S_2 有超类型，是同一泛型类或接口的两个不同参数。

    > T 是泛型类或接口 G 的参数化，并且 B_2 包含形式为α = S 或 S <: α之一的边界，不存在形式为 G<...>的类型是 S 的超类型，但是原始类型|G<...>| 是 S 的超类型。

    > T 是原生类型，§5.1.7 中提到的一个原生包装类是 B_2 中α的实例化、上界或下界。

    则 α 在 B_2 中被解析，所得到的 α 实例的捕获是 U，约束公式< U ->T> 被规约并与 B_2 合并。

  - 否则，约束公式<R θ ->T>被规约并与 B_2 合并。

- 如果 B_3 不包含边界 false，并且 B_3 中所有推断变量的解析成功，那么方法调用表达式与 T 兼容(§18.4)。

  考虑上一节的例子:

  ```
  List<Number> ln = Arrays.asList(1, 2.0);
  ```

  最具体的适用方法是:

  ```
  public static <T> List<T> asList(T... a)
  ```

  为了完成方法调用的类型检查，我们必须确定它是否与它的目标类型 List<Number>兼容。

  在上一节 B_2 中用来证明适用性的边界集是:

  { α <: Object, Integer <: α, Double <: α}

  新的约束公式集如下:

  { <List<α> ->List<Number>> }

  该兼容性约束产生了 α 的相等边界，该相等边界包含在新的边界集 B_3 中:

  { α <: Object, Integer <: α, Double <: α, α = Number }

  这些边界被简单地解析:

α= Number

最后，我们对 asList 的声明返回类型执行替换，以确定方法调用具有类型 List < Number > ； 显然，这与目标类型兼容。

这种推断策略不同于 Java 语言规范的 Java SE 7 版本，后者会根据其下界（甚至在考虑调用的目标类型之前）实例化α，正如我们在上一节中所做的那样。这将导致类型错误，因为生成的类型不是 List < Number > 的子类型。

在各种特殊情况下，基于 B2 中出现的边界，我们急切地解析一个作为调用返回类型出现的推断变量。这是为了避免不幸的情况，在这种情况下，通常的约束<R θ->T>不是完整性保持。不幸的是，通过急切地解析变量，我们可能无法利用稍后推断的边界。在某些情况下，稍后从调用参数推断的边界(例如隐式类型的 lambda 表达式)也可能会导致不同的结果，如果它们出现在 B_2 中的话。尽管有这些限制，但该策略允许在典型用例中产生合理的结果，并且向后兼容 Java 语言规范的 Java SE 7 版中的算法。

18.5.2.2 额外的参数约束

所选方法的调用类型是在考虑方法调用表达式的参数表达式可能隐含的附加约束之后确定的，如下所示：

- 如果方法调用表达式是多元表达式，假设 B_3 是在§18.5.2.1 中生成的边界集，以演示与方法调用的实际目标类型的兼容性。

  如果方法调用表达式不是多元表达式，设 B_3 与规约产生的边界集相同，以证明 m 在§18.5.1 中是适用的。

  (尽管在§18.5.1 和§18.5.2.1 中有必要证明边界集合中的推断变量是可以解析的，但这些解析步骤产生的实例化不被视为 B_3 的一部分。)

- 如下所示生成一组约束公式 C。

  设 e_1，…，e_k 是方法调用表达式的实际参数表达式。

  如果 m 通过严格或松散调用来应用，设 F_1, ..., F_k 是 m 的形式参数类型; 如果 m 通过可变参数调用可用，设 F_1，…，F_k 为 m 的前 k 个可变参数类型(§15.12.2.4)。

  设 θ 为定义在§18.5.1 中的替换[P_1:=α_1，…，P_p:=α_p]，将 m 的类型参数替换为推断变量。

  那么，对所有 i (1 ⩽ i ⩽ k):

  - 如果 e_i 与适用性无关，C 包含< e_i->F_i θ>。
  - 根据 e_i 的形式，可能会包括其他约束：
    > 如果 e_i 是一个 LambdaExpression，C 包含<LambdaExpression ->throws F_i θ>，并且在 lambda 体中搜索附加约束：
    > 对于块 lambda 体，搜索递归地应用于其每个结果表达式。
    > 对于多元类实例创建表达式或多元方法调用表达式，当推断多元表达式的调用类型时，C 包含将出现在由§18.5.2 生成的集合 C 中的所有约束公式。
    > 对于带括号的表达式，搜索将递归地应用于包含的表达式。

> 对于条件表达式，搜索递归地应用于第二和第三操作数。

> 对于 lambda 表达式，搜索将递归地应用于 lambda 主体。

> 对于 switch 表达式, 搜索递归地应用于每个结果表达式。

> 如果 e_i 是一个 MethodReference, C 包含<MethodReference ->throws F_i θ>。

> 如果 e_i 是多元类实例创建表达式或多元方法调用表达式，C 包含了在推断多元表达式的调用类型时，§18.5.2 生成的集合 C 中会出现的所有约束公式。

> 如果 e_i 是一个括号表达式，这些规则将递归地应用于包含的表达式。

> 如果 e_i 是一个条件表达式，这些规则将递归地应用于第二个和第三个操作数。

> 如果 e_i 是一个 switch 表达式，这些规则将递归应用到它的每个结果表达式。

- 当 C 不是空的时候，下面的过程重复进行，从边界集 B_3 开始，积累新的边界到一个“当前”的边界集，最终产生一个新的边界集 B_4:

  1. 在 C 中选择一个约束的子集，满足在 C 中对于每个约束，没有任何输入变量可以影响另一个约束的输出变量的属性。输入变量和输出变量的定义如下。如果α依赖于β的解析，推断变量α可以影响推断变量β(§18.4)，反之亦然;或者如果存在第三个推断变量γ，使得α可以影响γ，γ 可以影响β。

     如果这个子集是空的，那么在约束之间的依赖关系图中存在一个(或多个)循环。在这种情况下，C 中参与依赖循环(或多个循环)并且不依赖于循环(或多个循环)之外的任何约束将被考虑。从这些考虑的约束中选择一个约束，如下所示:

     - 如果任何考虑的约束具有形式<Expression->T>，那么，所选择的约束就是考虑过的这种形式的约束，它包含了每一个考虑过的这种形式的约束的表达式的左边(§3.5)的表达式。
     - 如果没有考虑的约束具有形式<Expression->T>，然后，选择的约束是考虑的约束，它包含每个其他考虑的约束的表达式左边的表达式。

  2. 选中的约束从 C 中移除。
  3. 解析所有选定约束的输入变量 α_1，…，α_m。
  4. T_1,...,T_m 是 α_1, ..., α_m 的实例化，替换[α_1:=T_1, ..., α_m:=T_m]被应用到每个约束。
  5. 由替换产生的约束被规约并与当前的边界集合并。

- 最后，如果 B_4 不包含边界 false，则解析 B_4 中的推断变量。

  如果推断变量 α_1, ..., α_p 的实例化 T_1, ..., T_p 解析成功, 设 θ' 为替换[P_1:=T_1，…，P_p:=T_p]。那么:

  - 如果在§18.5.1 中的约束集规约过程中需要进行未检查的转换才能应用该方法，然后将 θ'应用于 m 的类型的参数类型，得到 m 的调用类型的参数类型，而 m 的调用类

型的返回类型和抛出类型是通过擦除 m 的类型的返回类型和抛出类型来给出的。

- 如果该方法不需要未经检查的转换即可应用，则通过将 θ'应用于 m 的类型来获得 m 的调用类型。

如果 B_4 包含边界 false, 或者如果解析失败, 则会发生编译时错误。

规约附加参数约束的过程可能需要仔细地对<Expression->T>, <LambdaExpression->throws T>, 和 <MethodReference ->throws T>形式的约束公式进行排序。为便于排序，这些约束的输入变量定义如下：

- 对于 <LambdaExpression->T>:
  - 如果 T 是一个推断变量，它就是(唯一的)输入变量。
  - 如果 T 是一个函数式接口类型，并且可以从 T 派生出一个函数类型(§15.27.3)，则输入变量包括(i) 如果 lambda 表达式是隐式类型化的，则函数类型的参数类型所提到的推断变量; 和 (ii) 如果函数类型的返回类型 R 不是 void，则对 lambda 体中的每个结果表达式 e (或者对于主体本身，如果它是一个表达式), < e->R>的输入变量。
  - 否则，就没有输入变量。
- 对于<LambdaExpression ->throws T>:
  - 如果 T 是一个推断变量，它就是(唯一的)输入变量。
  - 如果 T 是一个函数式接口类型，并且可以派生出一个函数类型，如§15.27.3 所述，输入变量包括(i) 如果 lambda 表达式是隐式类型化的，则函数类型的参数类型所提到的推断变量; 和 (ii) 函数类型的返回类型提到的推断变量。
  - 否则，就没有输入变量。
- 对于 <MethodReference ->T>:
  - 如果 T 是一个推断变量，它就是(唯一的)输入变量。
  - 如果 T 是带有函数类型的函数式接口类型，并且如果方法引用不精确(§15.13.1)，那么输入变量就是函数类型的参数类型所提到的推断变量。
  - 否则，就没有输入变量。
- 对于<MethodReference ->throws T>:
  - 如果 T 是一个推断变量，它就是(唯一的)输入变量。
  - 如果 T 是带有函数类型的函数式接口类型，并且如果方法引用不精确(§15.13.1)，那么输入变量就是函数类型的参数类型和函数类型的返回类型所提到的推断变量。
  - 否则，就没有输入变量。
- 对于<Expression ->T>, 如果 Expression 是括号表达式:

Expression 包含的表达式是 Expression', 输入变量是 < Expression' ->T>的输入变量。

- 对于 <ConditionalExpression ->T>:

  条件表达式形式为 e_1 ? e_2 : e_3,输入变量是<e_2->T> 和<e_3->T>的输入变量。

- 对于<SwitchExpression-> T>:

  switch 表达式有结果表达式 e_1, ..., e_n, 输入变量是, 对所有 i (1 ≤ i ≤ n), <e_i ->T>的输入变量。

- 对于所有其他约束公式，没有输入变量。

这些约束的输出变量都是约束 T 右边的类型所提到的推断变量，它们不是输入变量。The

18.5.3 函数接口参数化推断

带有显式参数类型 P_1, ..., P_n 的 lambda 表达式指向一个带有至少一个通配符类型参数的函数式接口类型 F<A_1, ..., A_m>，然后 F 的参数化可以导出为 lambda 表达式的地面目标类型如下。

设 Q_1，⋯，Q_k 为类型为 F<α_1，⋯，α_m>的函数类型的参数类型, 其中 α_1，⋯，α_m 是新推断变量。

如果 n ≠ k, 不存在有效的参数化。否则，将形成一组约束公式, 对所有 i (1 ≤ i ≤n), < P_i = Q_i>。该约束公式集被规约以形成边界集 B。

如果 B 包含边界 false, 不存在有效的参数化。否则, 函数式接口类型的新参数化 F<A'$_1$, ..., A'$_m$>, 构造如下：, 对于 1 ≤ i ≤ m:

- 如果 B 包含 α_i 的实例化（§18.1.3）T， 则 A'$_i$ = T。
- 否则, A'$_i$ = A_i。

如果 F<A'$_1$, ..., A'$_m$> 不是一个格式良好的类型 (也就是说，类型参数不在它们的边界内), 或者如果 F<A'$_1$, ..., A'$_m$> 不是 F<A_1, ..., A_m>的子类型, 不存在有效的参数。否则,如果所有的类型参数是类型，则推断的参数化是 F<A'$_1$, ..., A'$_m$>, 或者如果一个或多个类型参数还是通配符，则是 F<A'$_1$, ..., A'$_m$>的非通配符参数化(§9.9)。

为了确定通配符参数化的函数式接口的函数类型，我们必须用特定的类型“实例化”通配符类型参数。“默认”方法是简单地用通配符的边界替换通配符，如§9.8 所述，但在 lambda 表达式具有与通配符边界不对应的显式参数类型的情况下，这将产生伪错误。例如:

```
Predicate<? super Integer> p = (Number n) -> n.equals(23);
```

Lambda 表达式是 Predicate<Number>，它是 Predicate<? super Integer>的子类型，但不是 Predicate<Integer>的子类型。本节中的分析用于推断 Number 是 Predicate 的类型参数的合适选择。

也就是说，这里的分析虽然是用一般的类型推断来描述的，但有意地非常简单。唯一的约束是相等约束，这意味着规约等同于简单的模式匹配。更强大的策略还可以从 lambda 表达式体中推断约束。但是，考虑到对周围和/或嵌套的泛型方法调用可能与推断进行交互，这将引入大量额外的复杂性。

18.5.4 更具体的方法推理

当测试一个适用的方法比另一个更具体时(§15.12.2.5)，当第二个方法是泛型方法时，有必要测试是否可以推断出第二个方法的类型参数的某些实例化，使第一个方法比第二个方法更具体。

假设 m_1 是第一种方法，m_2 是第二种方法。其中 m_2 有类型参数 P_1, ..., P_p, 设 α_1, ..., α_p 为推断变量, 设 θ 为替换[P_1:=α_1，…，P_p:=α_p]。

设 e_1, ..., e_k 为相应调用的参数表达式。则:

- 如果 m_1 和 m_2 适用于严格或松散调用(§15.12.2.2, §15.12.2.3), 则设 S_1, ..., S_k 为 m_1 的形式参数类型, 设 T_1, ..., T_k 为应用到 m_2 的形式参数类型 θ 的结果。
- 如果 m_1 和 m_2 适用于可变参数调用 (§15.12.2.4), 则设 S_1, ..., S_k 为 m_1 的前 k 个可变参数类型, 设 T_1, ..., T_k 为θ的结果适用于 m_2 的前 k 个可变参数类型。

  注意，没有替换应用于 S_1, ..., S_k; 即使 m_1 是泛型, m_1 的类型参数被当做类型变量,而不是推断变量。

确定 m_1 是否比 m_2 更具体的过程如下:

- 首先, 初始边界集 B, 是从 P_1, ..., P_p 的声明的边界生成的,如 §18.1.3 中指定。
- 其次, 对所有 i (1 ⩽ i ⩽ k), 生成一组约束公式或边界。
- 如果 T_i 是合适的类型, 如果 S_i 对于 e_i 比 T_i 更具体(§15.12.2.5)，则结果为 true，否则为 false。(请注意，S_i 始终是合适的类型。)
- 否则,如果 S_i 和 T_i 不都是函数式接口类型, 约束公式<S_i <: T_i > 被产生。

  否则，如果 Si 的接口是 Ti 的接口的超接口或子接口 (或者, 如果 Si 或 Ti 是交集类型, Si 的一些接口是 Ti 的一些接口的超接口或子接口), 约束公式<Si<: Ti> 被产生。

  否则，设 MT_S 为 S_i 的捕获的函数类型，MT_S'为 S_i(无捕获)的函数类型，MT_T 为 T_i 的函数类型。如果 MT_S 和 MT_S 具有不同数量的形参或类型参数，或者如果 MT_S 和 MT_S'没有相同的类型参数(§8.4.4)，则结果为 false。否则，从 MT_S 和 MT_T 的类型参数、形式参数类型和返回类型生成以下约束公式或边界:

  - 设 A_1,⋯,A_n 为 MTs 的类型参数, 设 B_1，…，B_n 为 MT_T 的类型参数。

    设 θ' 为替换[B_1:= A_1，…，B_n:= A_n]. 则, 对所有 j (1 ⩽ j ⩽ n):

    > 如果 A_j 的边界提到 A_1,⋯,A_n 中的一个，并且 B_j 的边界不是适当的类型，则为 false。
    >
    > 否则, 如果 X 是 A_j 的边界并且 Y 是 B_j 的边界, 则<X = Yθ'>。

      如果边界 A_j 提到 A1, ..., A_n 中的一个，而 B_j 的边界不是正确的类型，那么产生相等约束将增加推断变量被作用域外类型变量约束的可能性。由于使用超出作用域的类型变量实例化推断变量是没有意义的，所以我们宁愿避免这种情况，只要出现这种可能性就立即放弃。这种简化是不完整的。(同样的注释适用于下面的形式参数类型和返回类型的处理。)

- 设 U_1，…，U_k 为 MTs 的形式参数类型，设 V_1，…，V_k 为 MT_T 的形式参数类型。则，对所有 j (1 ⩽ j ⩽ k):
  - > 如果 U_j 提到了 A_1, ..., A_n 之一，并且 V_j 不是合适的类型，则为 false。
  - > 否则，$<V_j\ \theta' <:\ U_j>$，并且，当 U_1', ..., U_k' 是 MTs'的形式参数类型，A_1', ..., A_k' 是 MTs'的形式参数类型，$<V_j\ [B_1 := A_1', \ldots, B_n := A_n'] = U_j'>$
- 设 R_S 为 MT_S 的返回类型，设 R_T 为 MT_T 的返回类型，则:
  - > 如果 R_S 提及 A_1, ..., A_n 之一，并且 R_T 不是合适的类型,则为 false。
  - > 否则，如果 e_i 是一个显式类型的 lambda 表达式:
    - » 如果 R_T 是 void，则为 true。
    - » 否则，如果 R_S 和 R_T 是函数式接口类型，并且 e_i 有至少一个结果表达式，则对于 e_i 中每个结果表达式，重复整个第二步，对给定的结果表达式推断出 R_S 比 R_T θ'更具体的约束。
    - » 否则，如果 R_S 是原生类型而 R_T 不是，并且 e_i 有至少一个结果表达式，e_i 的每个结果表达式都是一个独立的原生类型表达式(§15.2)，则为 true。
    - » 否则，如果 R_T 是原生类型而 R_S 不是，并且 e_i 有至少一个结果表达式，e_i 的每个结果表达式都是一个独立的引用类型表达式或多元表达式，则为 true。
    - » 否则，$<R_S <:\ R_T\ \theta'>$。
  - > 否则，如果 e_i 是一个精确的方法引用:
    - » 如果 R_T 为 void，则为 true。
    - » 否则，如果 R_S 是一个原生类型而 R_T 不是，e_i 的编译时声明有一个原生返回类型，则为 true。
    - » 否则如果 R_T 是一个原生类型而 R_S 不是，e_i 的编译时声明有一个引用返回类型，则为 true。
    - » 否则，$<R_S <:\ R_T\ \theta'>$。
  - > 否则，如果 e_i 是一个括号表达式，这些从 R_S 和 R_T 派生的约束规则递归地应用于所包含的表达式。
  - > 否则，如果 e_i 是一个条件表达式，这些来自 R_S 和 R_T 的约束规则递归地应用于第二个和第三个操作数。
  - > 否则，如果 e_i 是一个 switch 表达式，这些来自 R_S 和 R_T 的约束规则递归地应用于其每个结果表达式。
  - > 否则,则为 false。

- 第三, 如果 m_2 通过可变参数调用可用并且具有 k+1 个参数, 当 S_{k+1} 是 m_1 的第 k+1 个可变参数类型，T_{k+1} 是 θ 应用于 m_2 的第 k+1 个可变参数类型的结果, 约束 $<S_{k+1} <: T_{k+1}>$ 被产生。
- 第四, 产生的边界和约束公式被规约并与 B 合并以产生边界集 B'。

  如果 B' 不包含边界 false, B'中所有推断变量的解析都成功, 则 m_1 比 m_2 更具体。

  否则，m_1 并不比 m_2 更具体。