



下载APP



25 | 增强编译器前端功能第4步：综合运用多种语义分析技术

2021-10-11 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 16:32 大小 15.16M



你好，我是宫文学。

在上一节课，我们比较全面地分析了怎么用集合运算的算法思路实现类型计算。不过，在实际的语义分析过程中，我们往往需要综合运用多种技术。

不知道你还记不记得，我们上一节课举了一个例子，里面涉及了数据流分析和类型计算技术。不过这还不够，今天这节课，我们还要多举几个例子，来看看如何综合运用各种技术来达到语义分析的目的。在这个过程中，你还会加深对类型计算的理解、了解常量折叠^和常量传播技术，以及实现更精准的类型推导。



好，我们首先接着上一节课的思路，看一看怎么把数据流分析与类型计算结合起来。

在类型计算中使用数据流分析技术

我们再用一下上节课的示例程序 foo7。在这个程序中，age 的类型是 number|null，age1 的类型是 string|number。我们先让 age=18，这时候把 age 赋给 age1 是合法的。之后又给 age 赋值为 null，然后再把 age 赋给 age1，这时编译器就会报错。

[复制代码](#)

```
1 function foo7(age : number|null){
2     let age1 : string|number;
3     age = 18;      //age的值域现在变成了一个值类型：18
4     age1 = age;    //OK
5     age = null;    //age的值域现在变成了null
6     age1 = age;    //错误！
7     console.log(age1);
8 }
```

在这个过程中，age 的值域是动态变化的。在这里，我用了“值域”这个词。它其实跟类型是同一个意思。我这里用值域这个词，是强调动态变化的特征。毕竟，如果说到类型，你通常会觉得变量的类型是不变的。如果你愿意，也可以直接把它叫做类型。

你马上就会想到，数据流分析技术很擅长处理这种情况。具体来说，就是在扫描程序代码的过程中，某个值会不断地变化。

提到数据流分析，那自然我们就要先来识别它的 5 大关键要素了。我们来分析一下。

首先是分析方向。这个场景中，分析方向显然是自上而下的。

第二，是数据流分析针对的变量。在这个场景中，我们需要分析的是变量的值域。所以，我用了一个 varRanges 变量，来保存每个变量的值域。varRanges 是一个 map，每个变量在里面有一个 key。

[复制代码](#)

```
1 varRanges:Map<VarSymbol, Type> = new Map();
```

第三，我们要确定 varRanges 的初始值。在这个例子中，每个变量的值域的初始值就是它原来的类型。比如 age 一开始的值域就是 number|null。

第四，我们要确定转换函数，也就是在什么情况下，变量的值域会发生变化。在当前的例子中，我们只需要搞清楚变量赋值的情况就可以了。如果我们要在变量声明中进行初始化，那也可以看做是变量赋值。

在变量赋值时，如果 = 号右边的值是一个常量，那么变量的值域都会变成一个值对象，这种情况我们已经在前一节课分析过了。

那如果 = 号右边的值不是常量，而是另一个变量呢？比如下面一个例子 foo10，x 的类型是 number|string，y 的类型是 string。然后把 y 赋给 x。我相信你也看出来，现在 x 的值域就应该跟 y 的一样了，都是 string。

[复制代码](#)

```
1 function foo10(x : number|string, y : string){
2   x = y;    //x的值域变成了string
3   if (typeof x == 'string'){ //其实这个条件一定为true
4     println("x is string");
5   }
6 }
```

研究一下这个例子，你会发现通过赋值操作，我们把 x 的值域收窄了。在 TypeScript 的文档中，这被叫做“[Narrowing](#)”。翻译成汉语的话，我们姑且称之为“窄化”吧。

不过，除了赋值语句，还有其他情况可以让变量的值域窄化，包括使用 typeof 运算符、真值判断、等值判断、instanceof 运算符，以及使用类型断言等等。其中最后两种方法，涉及到对象，我们目前还没有支持对象特性，所以先不讨论了。我们就讨论一下 typeof 运算符、真值判断和等值判断这三种情况。

首先讨论一下 **typeof 运算符**。其实在前面的例子 foo10 中，我们就使用了 typeof 运算符。typeof 是一个类型运算符，它能返回代表变量类型的字符串。不过它的结果只有少量几个值，包括 number、string、boolean、object、undefined、symbol 和 bigint。

我们再举一个例子 foo11，看看 typeof 是如何影响变量的值域的。

[复制代码](#)

```
1 function foo11(x : number|string){
2   let y: string;
```

```
3     if (typeof x === 'string'){ //x的值域变为string
4         y = x;                  //OK。
5     }
6 }
```

你可以看到，在示例程序 foo11 中，x 原来的类型是 `number|string`。但在 if 条件中，我们用 `typeof` 进行了类型的检测，只有当 x 的类型是 `string` 的时候，才会进入 if 块。所以，在 if 块中，我们用 x 给 `string` 类型的变量 y 赋值是没有错的。

在使用 `typeof` 的表达式中，你可以用四个运算符：`===`、`!==`、`==` 和 `!=`。其中 `===` 和 `==` 的效果是一样的，只不过前者的性能更高。同样，`!==` 和 `!=` 也是等价的。

接着，我们看看真值判断。什么是真值判断呢？我们还是举一个例子，这样理解起来更直观一些。

```
1 function foo12(x : string|null){
2     let y: string;
3     if (x){ //x的值域变为string & !""
4         y = x; //OK。
5     }
6 }
```


[复制代码](#)

在这个例子中，x 的类型是 `string|null`。但在 if 语句中，通过判断 x 是否为真，把 `x=null` 这个选项去掉了，这样就可以把 x 赋给 `string` 类型的 y 了。

这里，我还要给你补充点背景知识。在 TypeScript/JavaScript 中，我们其实可以把其他类型的值放入需要 `boolean` 值的地方，比如 `string`、`number`、`object` 等，它们会被自动转化成 `boolean` 值。不过，其中有一些值会被转化成 `false`，它们是：`0`、`NaN`、`""`（空字符串）、`0n` (`bigint` 类型中的 `0`)、`null`，以及 `undefined`。

除此之外的值，转化为 `boolean` 值以后都是 `true`。所以，在上面 foo12 示例程序的 if 条件中，x 是 `true`，那它就不可能是 `null` 值了，也不可能是空字符串。这样，最后形成的值域就是 `string & ! ""`。

最后，我们再看看**等值判断**。其实我们在上一节就见过等值判断的例子，我们把那个例子程序再拿过来看一下。

 复制代码


```
1 function foo9(age : number|null){
2     if (age == 18 || age == 81){ //age的值域现在是 18|81
3         console.log("18 or 81");
4     }
5     else{ //age的值域是 !18 & !81 & (number | null)
6         console.log("age is empty!");
7     }
8 }
```

在这个例子中有一个 if 语句，其中的条件表达式会生成一个值域，“18|81”。而对于 else 块，则需要先把“18|81”取补集，然后再跟 age 原来的值域求交集。

好了，现在我们就分析完了数据流分析中的第四个要素，也就是转换函数。**接下来我们看看最后一个要素，就是汇聚函数。**

什么时候需要用到汇聚函数呢？对于 if 语句来说，如果程序在 if 块和 else 块中都修改了某个变量的值域，那在 if 语句后面，变量的值域就需要做汇聚。我们还是通过一个例子来说明一下。

在下面的例子 foo13 中有一个 if 语句。在这个 if 语句中，if 块和 else 块分别都有一个对 y 赋值的语句。在 if 块中赋值语句是 y=x。在这里，x 的值域是 number，所以 y 的值域也是 number。在 else 块中，赋值语句是 y = 18，那 y 的值域就是 18。

 复制代码

```
1 function foo13(x : number|null){
2     let y:number|string;
3     let z:number;
4     if (x != null){ //x的值域是number
5         y = x; //y的值域是number
6     }
7     else{
8         y = 18; //y的值域是18
9     } //if语句之后，y的值域是number
10    z = y; //OK
11    return z;
12 }
```


那么，在退出 if 语句的时候，y 的值域应该是什么呢？你稍微分析一下就能看出来，这里应该取两个分支的并集，也就是 number。所以把 y 赋给 z 是可以的。

你可以把这个例子稍微修改一下，把 else 块中的赋值语句改为 y = "eighteen"。那当退出 if 语句以后，y 的值域是 number | "eighteen"。这样你再把 y 赋给 z，编译器就会报错。

[复制代码](#)

```
1 function foo14(x : number|null){
2     let y:number|string;
3     let z:number;
4     if (x != null){ //x的值域是number
5         y = x;      //y的值域是number
6     }
7     else{
8         y = "eighteen"; //y的值域是18
9     }              //if语句之后，y的值域是number|"eighteen"
10    z = y;          //编译器报错！
11    return z;
12 }
```

好了，到目前为止，我已经把数据流分析的 5 个要素都识别清楚了。思路清楚以后，你就可以去实现了。至于我的参考实现，你可以看一下 [TypeChecker](#)。

在这里，我要再分享一点心得。你会发现，即使我们已经多次使用数据流分析技术了，每次我还要把 5 个要素都过一遍。这是因为，我们做研发的时候，有个思维框架很重要，它可以引导你的思路，避免一些思维盲区。

比如，我在类型计算中使用数据流分析的时候，一开始注意力被其他技术点吸引了，忘记了用整个分析框架检查一遍，结果就忘记了实现汇聚函数，这就会导致一些功能缺失。后来用框架一检查，马上就补上了这个功能。

好了，到这里，我们已经基本介绍清楚了如何使用数据流分析技术来做类型计算。不过，类型计算还可能受到其他技术的影响。接下来我就介绍一下常量折叠（Constant Folding）和常量传播（Constant Propagation）。常量折叠和常量传播的结果，会进一步影响到类型计算的结果。

常量折叠和常量传播

我们还是先看一个例子来理解一下这两个概念。这个例子中有 x_1 , x_2 和 x_3 三个变量。我们首先给 x_2 赋予常量 10。接着，我们把 x_2+8 赋给 x_3 。从这里你能计算出，其实 x_3 的值也是一个常量，它的值是 18。

[复制代码](#)

```
1 function foo15(x1:number|null):number{
2     let x2 = 10;           //x2是常量10
3     let x3 = x2 + 8;       //x3是常量18
4     if (x1 == x3 ){        //x1的值域是18
5         return x1;         //OK!
6     }
7     return x2;
8 }
```

你看，执行到这里，我们其实在编译期就把 x_2+8 的值计算出来。这样，在生成汇编代码的时候，我们就不需要进行相应的计算了，直接给 x_3 赋值为 18 就行了。这个技术就叫做**常量折叠**。它能让一些常量的计算在编译期完成，这样就能提高程序在运行期的性能。

同时，在 $x_3 = x_2+8$ 这行程序中，还有一个现象，叫做**常量传播**。什么意思呢？在这行中， x_2 的值已经是一个常量 10 了，它的常量值被传播到了 x_2+8 这个表达式中，从而计算出了一个新的常量 x_3 。

再接下来是一个 if 语句。这个时候， x_3 的值传播到了 if 条件中。这就影响到了 x_1 的值域。现在 x_1 的值域就变成 18 了。所以，当我们在 if 块中执行 return x_1 的时候，代码是正确的，满足返回值必须是 number 的要求。

那常量传播具体怎么实现呢？

在 PlayScript 的实现中，我们给每个表达式都添加了一个 constValue 属性。通过遍历树的方式，就可以求出每个表达式的常量值，并记录到 constValue 属性。在生成目标代码的时候，就可以直接使用这个常量值，不需要在运行期做计算。


好了，现在我们已经了解了常量折叠和常量传播技术，也分析了它对类型计算的影响。

不过，到目前为止，对于类型计算的结果，我们都是用在类型检查的场景里。其实，类型计算的结果也能用于类型推导，能够提高类型推导的准确程度。而常量折叠和传播，也会在其中起到作用。

类型推导

在之前 PlayScript 版本中，我们也实现了基本的类型推导功能。但那个时候，类型推导都是基于变量声明时的类型，而不是基于数据流分析来获得变量动态的值域，再根据这个值域做类型推导。基于变量声明进行推导的结果肯定是不够精准的。

同样，我们举个例子看一下。在这个例子中，变量 `a` 的类型是 `number|string`，我们再给 `a` 赋值为 `"hello"`，现在 `a` 的值域是 `"hello"`。再然后呢，我们声明了一个变量 `b`，并把变量 `a` 作为它的初始化值。

 复制代码

```
1 function foo16(a:number|string){
2     a = "hello";    //a的值域是"hello"
3     let b = a;      //推导出b的类型是string
4     console.log(typeof b);
5 }
```

那么问题来了，现在 `b` 应该是什么类型呢？我给你两个候选答案，让你选一下：

选项 1：`b` 的类型 `a` 原来的类型是一样的，都是 `number|string`。

选项 2：`b` 的类型是 `string`，因为采用常量传播技术，我们已经知道 `a` 的值是 `"hello"` 了。

我估计你应该会选出正确的答案，就是选项 2。其实，上面的 `"let b = a"` 这个语句，就等价于 `"let b = 'hello' "`，所以你应该能够推导出 `b` 的类型是 `string`。

不过，这里要注意，我们不能因为 `a` 当前的值域是 `"hello"`，就推导出变量 `b` 的类型也是值类型 `"hello"`，这就把变量 `b` 限制得太死了。TypeScript 会采用 `"hello"` 的基础类型 `string`。

类型推导还有更复杂一点的场景。比如，在下面的例子中，我们仍然用 `a` 来初始化变量 `b`。不过，现在 `a` 的值域是 `10|null`。

[复制代码](#)

```
1 function foo17(a:number|string|null){
2     if(a == 10 || a == null){ //a的值域是10|null
3         let b = a;           //推导出b的类型是number|null
4         if (b == "hello"){    //编译器报错！
5             console.log("whoops");
6         }
7     }
8 }
```

基于 `a` 的值域，编译器会把 `b` 的类型推导为 `number|null`。所以，这个时候如果我们用 `b=="hello"` 让 `b` 跟字符串做比较，编译器就会报错，指出类型 `number|null` 和 `string` 之间没有重叠，所以不能进行 `==` 运算。

```
→ 24 git:(master) × tsc --strict example_type2.ts
example_type2.ts:135:13 - error TS2367: This condition will always return 'false'
since the types 'number | null' and 'string' have no overlap.

135         if (b == "hello"){
           ~~~~~

Found 1 error.
```

好了，通过刚才的分析，相信你对类型计算的在类型推导中的作用，也有了一些直观的了解。

课程小结

今天的内容就是这些。在今天这节课，我希望你能在以下几个方面有所收获。

首先，我们采用数据流分析的框架，可以动态地计算变量在每行代码处的值域，或者叫做类型。通过变量赋值、`typeof` 运算符、真值判断和等值判断等操作，变量的值域会不停地被窄化。不过，在多个条件分支汇聚的地方，又会通过求并集而把值域变宽。

第二，常量折叠技术能够在编译期提前计算出常量，这样我们就不需要在运行期再计算了，从而提高程序性能。而常数传播技术，能够把常数随着代码传播到其他地方，从而计

算出更多的常量。这些传播出去的常量，还会让类型计算的结果更加准确。

第三，类型计算的结果不仅可以用于类型检查，还可以用于类型推导，让类型推导的结果更加准确。

今天这节课实现的功能，你仍然可以参考 [🔗TypeUtil](#) 和 [🔗TypeChecker](#) 的实现，并且运行 [🔗example_type2.ts](#) 示例程序。

为了更好地支持类型计算的功能，我还给编译器增加了对 `typeof` 语法的支持。增加的新语法规则叫做 `typeOfExp`。

[📄 复制代码](#)

```
1 primary: literal | functionCall | '(' expression ')' | typeOfExp ;  
2 typeOfExp : 'typeof' primary;
```

另外，我还增加了对于 `===` 和 `!==` 的支持。现在你对于支持新的语法规则应该已经驾轻就熟了，所以我在这里就不多展开了。你可以去看看 [🔗示例程序的源代码](#)。

那么，对 TypeScript 的类型系统和其他编译器前端功能的实现，我们到此就告一个段落了。这些功能将会给我们后面实现编译器后端特性提供很好的支撑！

思考题

今天的思考题是关于类型推导的。如果 `b` 的值域是 `0 | 1 | true | false`，那么在 `let a = b` 这样一个变量声明语句中，编译器推导出的 `a` 的类型应该是什么呢？

欢迎你把这节课分享给更多对编译器前端感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

1. 这节课示例代码的 [🔗目录](#)；
2. 这节课你仍然需要关注 [🔗TypeChecker](#) 和 [🔗TypeUtil](#) 的代码；
3. Parser 中解析 [🔗TypeOfExp](#) 的代码，非常简单；

4. 测试程序仍然是放在 [example_types.ts](#) 中，不过例子更多了。你每次可以注释掉其他的例子，只运行其中的一个，测试编译器的行为。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | 增强编译器前端功能第3步：全面的集合运算

下一篇 26 | 增强更丰富的类型第1步：如何支持浮点数？

4 周年庆限定

299元随心畅学卡

五门课程任你选，总价值高达千元

超值拿下 

畅学卡

¥299

4 周年

精选留言 (1)

 写留言



罗乾林

2021-10-11

思考题a的类型是number和boolean

展开

