Отладчик типов языка программирования Scala в IntelliJ IDEA

Роман Васильев
руководитель: Александр Подхалюзин

Академический университет

13 июня 2017 г.

Введение

Система типов

«Система типов - это гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений».

Система типов в языке программирования Scala:

- Статическая
- Строгая
- С выводом типов

Рассматриваемые процессы

- Проверка сводимости типов.
- Вывод типов.
- Выбор перегрузки.

Спецификация Scala

- Сводимость вводится как отношение, соответствующее транзитивному замыканию над набором правил вывода.
- Локальный вывод типов:
  - $expr.x$
  - $expr$
  - $expr(d_1, ..., d_m)$
- Разрешение перегрузки сначала пытается отсеять кандидатов, не обращаясь к конкретным переданным аргументам. После этого идет выбор наиболее специфичного представителя.

Существующее решение

Scala type debugger [1] [2] использует инфраструктуру логгирования компилятора Scala для сбора информации и библиотеку prefuse для пользовательского интерфейса.

Проблемы:

- Используется специальная, инструментированная версия компилятора.
- Последний коммит в 2012.

[1] https://github.com/hubertp/prefuse-type-debugger

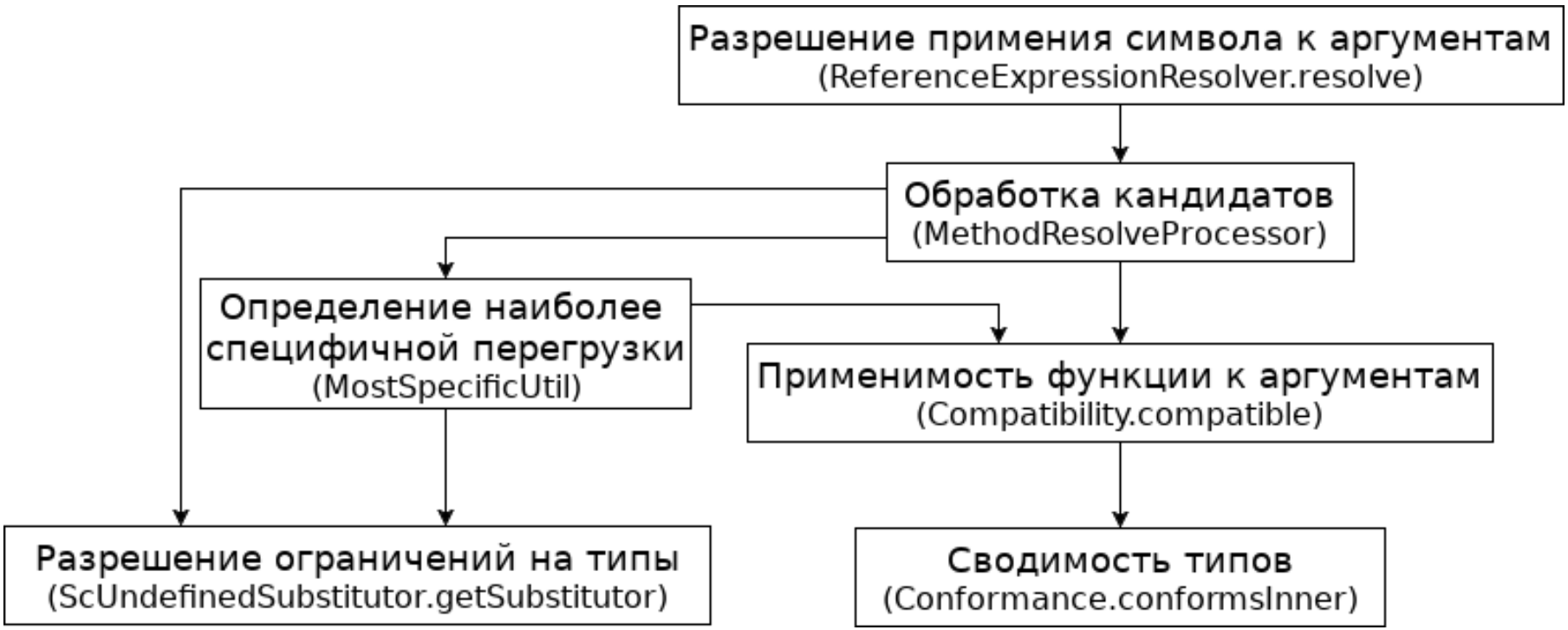
[2] https://infoscience.epfl.ch/record/179877/files/typedebugger-aplc2012.pdf

Цель и задачи

Цель: Реализовать отладчик процесса проверки типов в Scala Plugin.
Задачи:

- Инструментировать Scala Plugin для сбора промежуточной информации.
- Дать интерпретацию, приближенную к спецификации Scala.
- Минимизировать влияние инструментации во время выполнения на оставшуюся часть плагина.

Архитектура Scala Plugin



Инструментирование и макросы

- def f(..., h: Option[H]) → def f(...); def f$I(..., h: Option[H])
- class F(..., h: Option[H]) → class F(...); object F.$I(..., h: Option[H])

```
@uninstrumental("handler")
case class MostSpecificUtil(
  elem: PsiElement,
  length: Int,
  handler: Option[DCHandler.Resolver] = None
)(implicit typeSystem: TypeSystem)
```

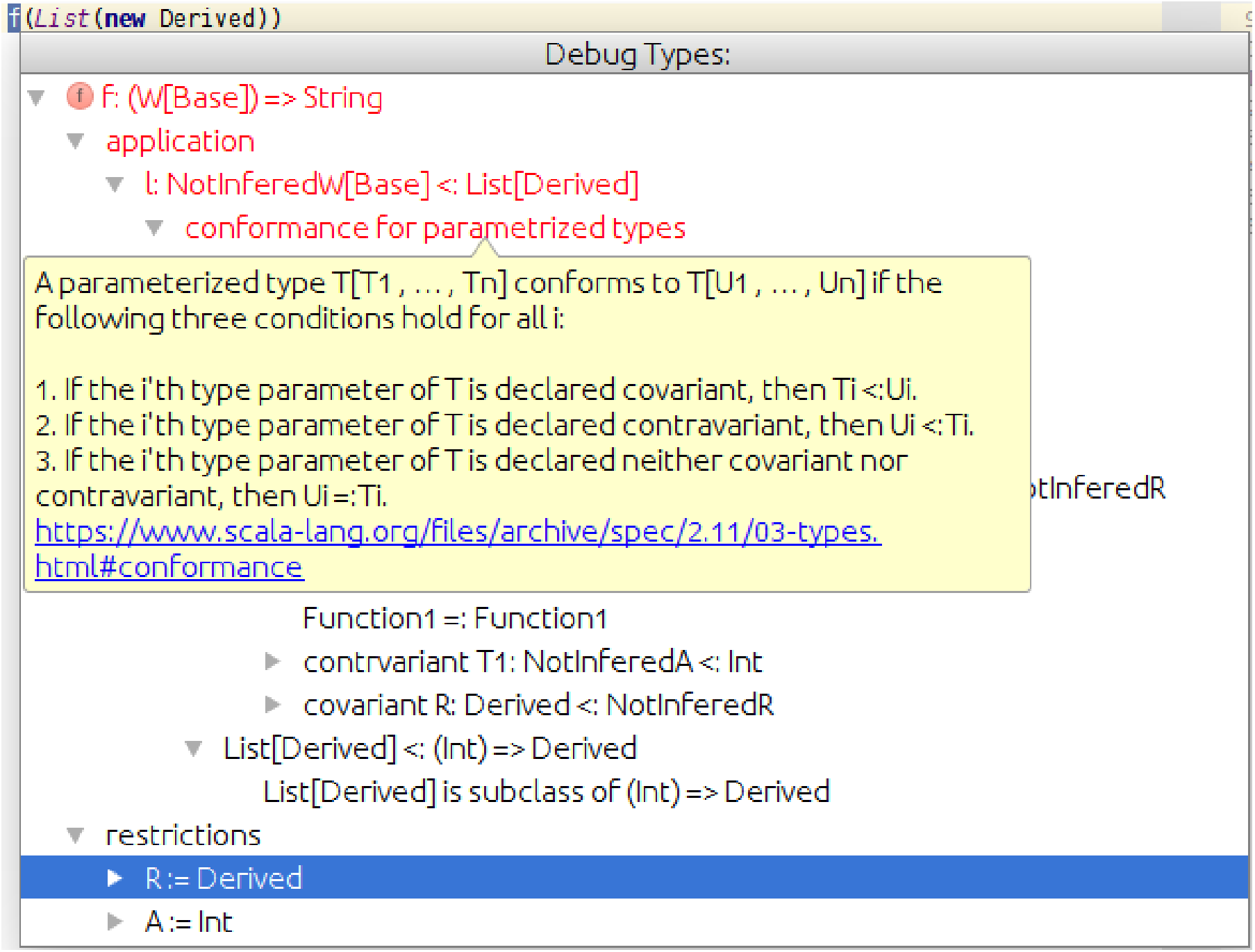
Пример

```
class Base

class Derived extends Base

def f[W[X] <: Seq[X]](l: W[Base]) = "1"

def f[A, R](f: A => R) = "2"

f(List(new Derived))
```

```
f(List(new Derived))
```

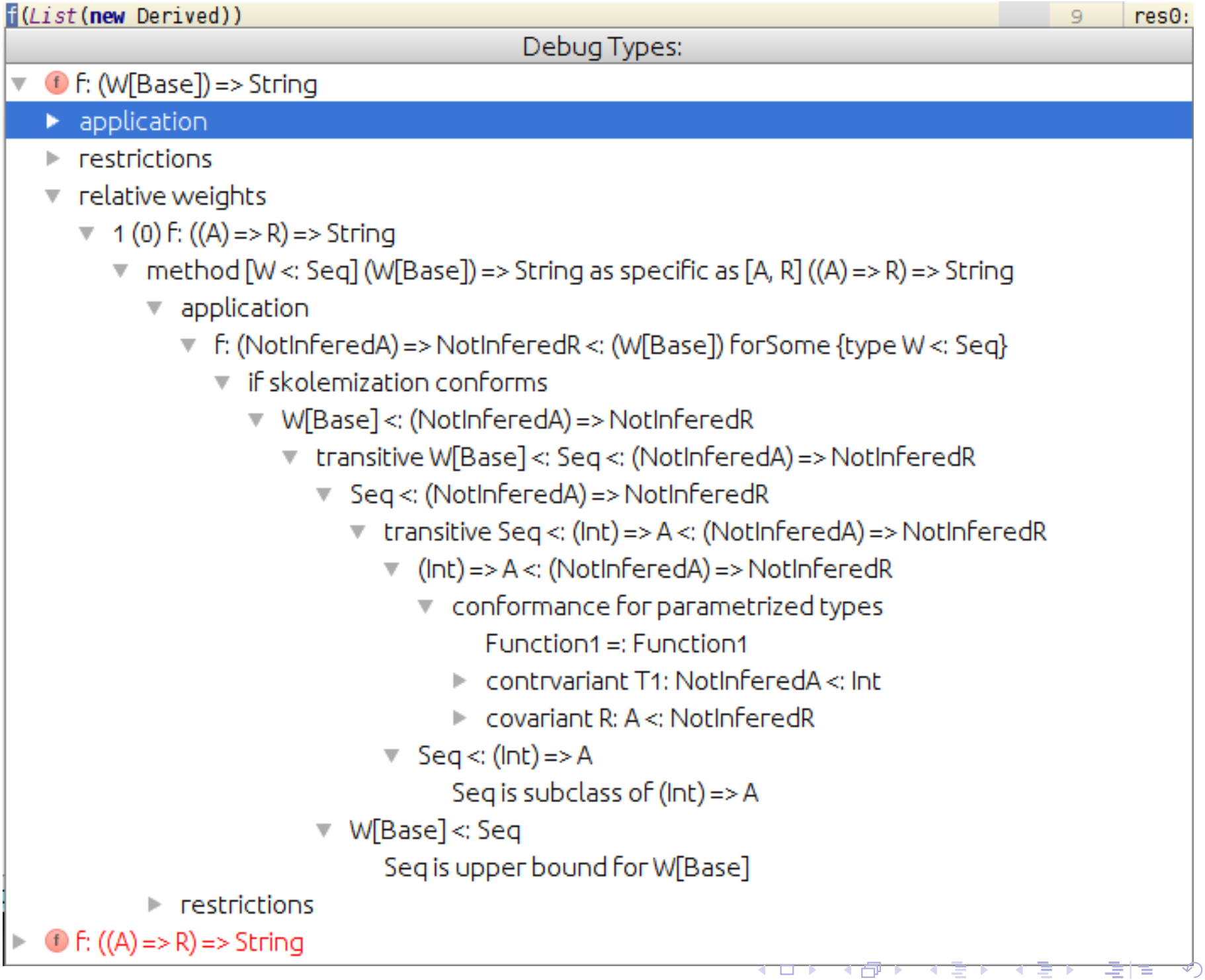
Debug Types:

- f: (W[Base]) => String
  - application
    - l: NotInferedW[Base] <: List[Derived]
      - conformance for parametrized types
        - List =: NotInferedW
        - invariant X: Derived =: Base
- f: ((A) => R) => String
  - application
    - f: (NotInferedA) => NotInferedR <: List[Derived]
      - transitive List[Derived] <: (Int) => Derived <: (NotInferedA) => NotInferedR
        - (Int) => Derived <: (NotInferedA) => NotInferedR
          - conformance for parametrized types
            - Function1 =: Function1
            - contrvariant T1: NotInferedA <: Int
            - covariant R: Derived <: NotInferedR
        - List[Derived] <: (Int) => Derived
          - List[Derived] is subclass of (Int) => Derived
  - restrictions
    - R := Derived
    - A := Int


f(List(new Derived))
Debug Types:
f: (W[Base]) => String
application
l: NotInferedW[Base] <: List[Derived]
conformance for parametrized types
A parameterized type T[T1 , … , Tn] conforms to T[U1 , … , Un] if the following three conditions hold for all i:
1. If the i'th type parameter of T is declared covariant, then Ti <:Ui.
2. If the i'th type parameter of T is declared contravariant, then Ui <:Ti.
3. If the i'th type parameter of T is declared neither covariant nor contravariant, then Ui =:Ti.
https://www.scala-lang.org/files/archive/spec/2.11/03-types.html#conformance
otInferedR
Function1 =: Function1
contrvariant T1: NotInferedA <: Int
covariant R: Derived <: NotInferedR
List[Derived] <: (Int) => Derived
List[Derived] is subclass of (Int) => Derived
restrictions
R := Derived
A := Int

Исправленный пример

```scala
class Base

class Derived extends Base

def f[W[+X] <: Seq[X]](l: W[Base]) = "1"

def f[A, R](f: A => R) = "2"

f(List(new Derived))
```


f(List(new Derived))
9
res0:
Debug Types:
f: (W[Base]) => String
application
restrictions
relative weights
1 (0) f: ((A) => R) => String
method [W <: Seq] (W[Base]) => String as specific as [A, R] ((A) => R) => String
application
f: (NotInferedA) => NotInferedR <: (W[Base]) forSome {type W <: Seq}
if skolemization conforms
W[Base] <: (NotInferedA) => NotInferedR
transitive W[Base] <: Seq <: (NotInferedA) => NotInferedR
Seq <: (NotInferedA) => NotInferedR
transitive Seq <: (Int) => A <: (NotInferedA) => NotInferedR
(Int) => A <: (NotInferedA) => NotInferedR
conformance for parametrized types
Function1 =: Function1
contrvariant T1: NotInferedA <: Int
covariant R: A <: NotInferedR
Seq <: (Int) => A
Seq is subclass of (Int) => A
W[Base] <: Seq
Seq is upper bound for W[Base]
restrictions
f: ((A) => R) => String

Результат

Основываясь на спецификации Scala, в Scala Plugin был добавлен инструмент, позволяющий явно визуализировать процессы проверки типов.

Снижено влияние инструментации, благодаря использованию макросов.

Код можно посмотреть в репозитории
`https://github.com/nizshee/intellij-scala`

Конец

Сложности

- Scala.
- Несоответствие сущностей, используемых в плагине и в документации Scala. Частая необходимость в обратном анализе.
- Наглядный пользовательский интерфейс.
- Работа с макросами. Трудность отладки.