InSynth: Complete Completion using Types and Weights

Tihomir Gvero, Ivan Kuraj,
Viktor Kuncak, Ruzica Piskac

Introduction

- Libraries and APIs - the biggest assets for today’s software developers:
  - large number of classes and methods
- Difficult to start using such APIs productively, even for simple tasks
- IDE (Eclipse and IntelliJ):
  - list declarations for a given receiver object
- IntelliJ - compose simple method sequences
- Need to improve modern IDEs

Introduction

- Observation:
  - the developer often has the type of a desired object in mind (backward-directed completion)
  - val a:InputStream = new File(name).getInputStream()
- We do not want developer to indicate a starting value (e.g. "new File(name)" as receiver)
- We want to use:
  - All declarations in the current scope
  - Desired type
- Generate full expression:
  - Methods/fields/locals sequence
  - All arguments (e.g. receiver)

InSynth

- Implemented for Scala language
- Input:
  - Partial Scala program – visible declarations
  - Program point – desired type

  } **Type Information**
- Output:
  - Code snippets – expression with desired type
- Runs synthesis algorithm based on resolution to find candidate snippets
- Handles ground and function types

Demo

Type Inhabitation Problem

- Given a set of types **T** and a set of variables **A**, a type environment is a set

$$\Gamma = \{a_1 : \tau_1, a_2 : \tau_2, \ldots, a_n : \tau_n\}$$

where τ_i in **T** and a_i in **A**

Type Inhabitation Problem

Given a type environment Γ, a type τ and some calculus, is there are an expression *e* such that $\Gamma \vdash e : \tau$

Succinct Types - Motivation

m_1():Int

m_2(s: String):Boolean

m_3(a: Boolean):Int

m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

List[Int] ?

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

List[Int] ?

With **"τ ?"** we denote we search for an expression with type **τ**.

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

m4(Int?, Int?, Int?)

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

m4(m3(Boolean?), Int?, Int?)

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

m_4(m_3(m_2(String?)), Int?, Int?)

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

Wrong path!

m4(m3(m2(String?)), Int?, Int?)

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

We backtrack!

m_4(Int?, Int?, Int?)

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

m_4(m_1(), Int?, Int?)

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

m_4(m_1(), m_3(m_2(String?), Int?)

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

Again, wrong path!

m4(m1(), m3(m2(String?), Int?)

Succinct Types - Motivation

$m_1()$:Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
$m_4(x_1$: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

$m_4(m_1(), m_1()$, Int?)

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

m_4(m_1(), m_1(), m_3(m_2(String?)))

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

Yet again, wrong path!

m4(m1(), m1(), m3(m2(String?)))

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

m_4(m_1(), m_1(), m_1())

Succinct Types - Motivation

m1():Int
m2(s: String):Boolean
m3(a: Boolean):Int
m4(x1: Int, x2: Int, x3: Int):List[Int]

Desired type = List[Int]

m4(m1(), m1(), m1()) ← We succeed but…

Succinct Types - Motivation

m_1():Int

m_2(s: String):Boolean

m_3(a: Boolean):Int

m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

m_4(m_1(), m_1(), m_1())

m_3(m_2(String?))
Explored 3 times!

Succinct Types - Motivation

m_1():Int
m_2(s: String):Boolean
m_3(a: Boolean):Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]

Desired type = List[Int]

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

List[Int] ?

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m4{Int?}

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m4{m3{Boolean?}}

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} $\rightarrow$ Boolean
m3(a: Boolean):Int	{Boolean} $\rightarrow$ Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} $\rightarrow$ List[Int]
Desired type = List[Int]	List[Int]

m4{m3{m2{String?}}}

Succinct Types - Motivation

m_1():Int	Int
m_2(s: String):Boolean	{String} → Boolean
m_3(a: Boolean):Int	{Boolean} → Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]	{Int} → List[Int]

Desired type = List[Int] List[Int]

Wrong path!

m_4{m_3{m_2{String?}}}

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m4{Int?}

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m4{m1}

Succinct Types - Motivation

$m_1():Int$	Int
$m_2(s: String):Boolean$	{String} → Boolean
$m_3(a: Boolean):Int$	{Boolean} → Int
$m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]$	{Int} → List[Int]
Desired type = List[Int]	List[Int]

$m_4\{m_1\}$

Enough info to construct expression

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m4(m1(), m1(), m1())

Succinct Types - Motivation

m1():Int	Int
m2(s: String):Boolean	{String} → Boolean
m3(a: Boolean):Int	{Boolean} → Int
m4(x1: Int, x2: Int, x3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m4(m1(), m1(), m1()) ← We succeed and...

Succinct Types - Motivation

$m_1()$:Int	Int
m_2(s: String):Boolean	{String} → Boolean
m_3(a: Boolean):Int	{Boolean} → Int
m_4(x_1: Int, x_2: Int, x_3: Int):List[Int]	{Int} → List[Int]
Desired type = List[Int]	List[Int]

m_4(m_1(), m_1(), m_1())

m_3(m_2(String?))
Explored **only once!**

Succinct Types - Definition

- Let B be a set of basic types (Int, Boolean, String, List(Int), and etc)
- Set of Scala/lambda types τ(B) is defined by grammar:

$$\tau ::= \tau \rightarrow \tau \mid v \qquad v \in B$$

- Set of all **succinct** types t_s(B) is defined by grammar:

$$T_g ::= \{T_g, \ldots, T_g\} \rightarrow v \qquad v \in B$$

Translation

- With σ we denote the function that converts Scala/ lambda into succinct types:

$$\sigma(v) = v$$

$$\sigma(t1 \rightarrow \ldots \rightarrow tn \rightarrow v) = \{\sigma(t1),\ldots,\sigma(tn)\} \rightarrow v$$

where $v \in B$ and t1,…,tn are Scala/lambda types

SCALA DECLARATIONS	SUCCINCT TYPE
val l: List[Int]	List(Int)
def iTs(a: Int, b:Int): String	{Int} → String
def q(g : Int, f: Int=>Boolean): String	{Int, {Int} → Boolean} → String
class A extends B	{A} → B

Translation

- Scala/lambda type environment:

$$\Gamma_o = \{a_1 : \tau_1, a_2 : \tau_2, \dots , a_n : \tau_n\}$$

- $a: \tau$ - Scala/lambda type declaration
- Translated into succinct type:

$$\sigma(a: \tau) = \sigma(\tau)$$

- Then, succinct type environment is:

$$\Gamma = \sigma(\Gamma_o) = \{\sigma(\tau_1), \sigma(\tau_2), \dots , \sigma(\tau_n)\}$$

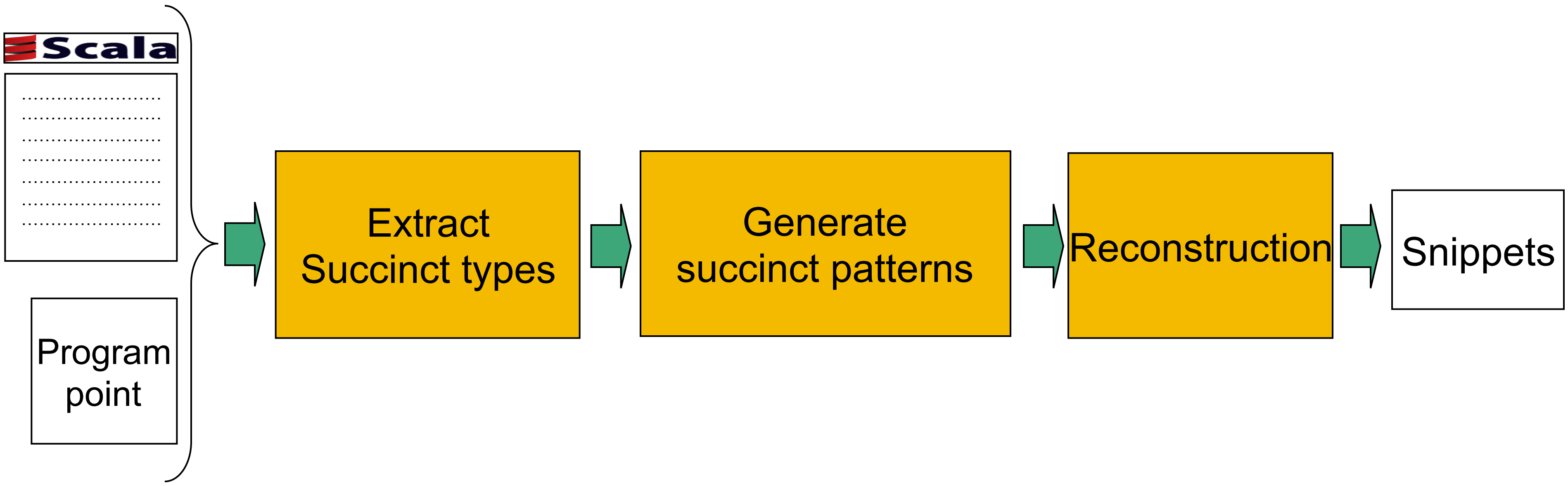
Succinct Pattern

- Γ - succinct environment
- $t_1, \ldots, t_n, t$ are succinct types, and t in B
- Then succinct pattern is:

$$\Gamma @ \{t_1, \ldots, t_n\} : t$$

- Records information on *how* t is inhabited in Γ:
  - There is a type $\{t_1, \ldots, t_n\} \rightarrow t$ in Γ, where $t_1, \ldots, t_n$ are inhabited in Γ

Algorithm



Scala
Program point
Extract Succinct types
Generate succinct patterns
Reconstruction
Snippets

Pattern Generation

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Pattern Generation

Translation

Γ_o		Γ
m1:Int		
m2:Int		Int
m3(c: Char):Boolean		{Char} → Boolean
m4(a: Int, f: Char => Boolean):String	⇨	{Int, {Char} → Boolean} → String
m5(d: Double):Long		{Double} → Long
m6:Double		Double

Desired type = String

Desired type = String

Pattern Generation

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

Desired type = String

Pattern Generation

Patterns

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

Desired type = String

Pattern Generation

Backward search

Patterns

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

String?

Desired type = String

Pattern Generation

Backward search

Patterns Γ

Γ@{ ? }:String ⇐ Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

String? Desired type = String

Pattern Generation

Use type to guess pattern

Patterns Γ

Int
{Char} → Boolean
Γ@{ ? }:String ⇐ {Int, {Char} → Boolean} → String
{Double} → Long
Double

String? Desired type = String

Pattern Generation

Patterns Γ

		Int
		{Char} → Boolean
Γ@{Int?, ({Char} → Boolean)?}:String	⇐	**{Int, {Char} → Boolean} → String**
		{Double} → Long
		Double

String? Desired type = String

Pattern Generation

Patterns	Γ
Γ@{ ? }:Int	Int
	{Char} → Boolean
Γ@{Int?, ({Char} → Boolean)?}:String ⇐	{Int, {Char} → Boolean} → String
	{Double} → Long
	Double
String?	Desired type = String

Pattern Generation

Patterns	Γ
Γ@{ }:Int	Int
	{Char} → Boolean
Γ@{Int?, ({Char} → Boolean)?}:String	{Int, {Char} → Boolean} → String
	{Double} → Long
	Double
String?	Desired type = String

Pattern Generation

Function type

Patterns	Γ
Γ@{ }:Int	
Γ@{Int, ({Char} → Boolean)?}:String ⇦	Int {Char} → Boolean {Int, {Char} → Boolean} → String {Double} → Long Double
String?	Desired type = String

Pattern Generation

Extend environment and search for Boolena expression

Patterns

Γ

Γ@{ }:Int

Γ@{Int, {Char} → (Boolean)?}:String

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

String?

Desired type = String

Pattern Generation

Extend environment

Patterns	Γ
Γ@{ }:Int	
Γ@{Int, {Char} → (Boolean)?}:String	Int {Char} → Boolean {Int, {Char} → Boolean} → String {Double} → Long Double
String?	Desired type = String

Pattern Generation

Extend environment

Patterns

$\Gamma @\{\ \}$:Int

$(\Gamma$ U {Char})
Γ@{Int, {Char} → (Boolean)?}:String

String?

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

Desired type = String

Pattern Generation

Search for Boolean expression

Patterns	Γ
Γ@{ }:Int	
	Int
(Γ U {Char})@{ ? }:Boolean	{Char} → Boolean
Γ@{Int, {Char} → (Boolean)?}:String	{Int, {Char} → Boolean} → String
	{Double} → Long
	Double
String?	Desired type = String

Pattern Generation

Patterns	Γ
Γ@{ }:Int	Int
(Γ U {Char})@{Char?}:Boolean	{Char} → Boolean
Γ@{Int, {Char} → (Boolean)?}:String	{Int, {Char} → Boolean} → String
	{Double} → Long
	Double
String?	Desired type = String

Pattern Generation

Patterns		Γ
Γ@{ }:Int (Γ U {Char})@{ ? }: Char (Γ U {Char})@{Char?}:Boolean Γ@{Int, {Char} → (Boolean)?}:String	⇐	Int {Char} → Boolean {Int, {Char} → Boolean} → String {Double} → Long Double
String?		Desired type = String

Pattern Generation

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char?}:Boolean
Γ@{Int, {Char} → (Boolean)?}:String

String?

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

Desired type = String

Pattern Generation

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → (Boolean)?}:String

String?

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

Desired type = String

Pattern Generation

Patterns		Γ
Γ@{ }:Int		
(Γ U {Char})@{ }: Char		Int
(Γ U {Char})@{Char}:Boolean		{Char} → Boolean
Γ@{Int, {Char} → Boolean}:String	⇦	{Int, {Char} → Boolean} → String
		{Double} → Long
		Double
String?		Desired type = String

Pattern Generation

Patterns	Γ
Γ@{ }:Int	
(Γ U {Char})@{ }: Char	Int
(Γ U {Char})@{Char}:Boolean	{Char} → Boolean
Γ@{Int, {Char} → Boolean}:String	{Int, {Char} → Boolean} → String
	{Double} → Long
	Double

Success! We can synthesize a snippet with type String.

Desired type = String

Pattern Generation

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ

Int
{Char} → Boolean
{Int, {Char} → Boolean} → String
{Double} → Long
Double

Backward search benefit!

Desired type = String

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

[ ]:String

HOLE

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

[ ]:String

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

[ ]:String

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4([ ]:Int , [ ]:Char => Boolean)

Reconstruction

Patterns

Partial expressions

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

m4([ ]:Int , [ ]:Char => Boolean)

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4([ ]:Int , [ ]:Char => Boolean)

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , [ ]:Char => Boolean)

m4(m2 , [ ]:Char => Boolean)

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , [ ]:Char => Boolean)

m4(m2 , [ ]:Char => Boolean)

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => [ ]:Boolean)

m4(m2 , (x: Char) => [ ]:Boolean)

Reconstruction

Patterns

Γ@{ }:Int

(Γ U {Char})@{ }: Char

(Γ U {Char})@{Char}:Boolean

Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int

m2:Int

m3(c: Char):Boolean

m4(a: Int, f: Char => Boolean):String

m5(d: Double):Long

m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => [ ]:Boolean)

m4(m2 , (x: Char) => [ ]:Boolean)

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => [ ]:Boolean)

m4(m2 , (x: Char) => [ ]:Boolean)

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} → Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => m3([ ]:Char))

m4(m2 , (x: Char) => m3([ ]:Char))

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => m3([ ]:Char))

m4(m2 , (x: Char) => m3([ ]:Char))

Reconstruction

Patterns

Γ@{ }:Int

(Γ U {Char})@{ }: Char

(Γ U {Char})@{Char}:Boolean

Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => m3([ ]:Char))

m4(m2 , (x: Char) => m3([ ]:Char))

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => m3([ ]:Char))

m4(m2 , (x: Char) => m3([ ]:Char))

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Partial expressions

m4(m1 , (x: Char) => m3(x))

m4(m2 , (x: Char) => m3(x))

Reconstruction

Patterns

Γ@{ }:Int
(Γ U {Char})@{ }: Char
(Γ U {Char})@{Char}:Boolean
Γ@{Int, {Char} $\rightarrow$ Boolean}:String

Γ_o

m1:Int
m2:Int
m3(c: Char):Boolean
m4(a: Int, f: Char => Boolean):String
m5(d: Double):Long
m6:Double

Desired type = String

Expressions

m4(m1 , (x: Char) => m3(x))

m4(m2 , (x: Char) => m3(x))

Completeness and Complexity

Type Inhabitation [Statman, 1979]

- the problem is PSPACE-complete

Theorem

The type inhabitation in ground applicative calculus without generating lambda expressions can be solved in polynomial time.

- For the case when we generate lambda expressions as well, we implemented a complete algorithm

Quantitative Type Inhabitation Problem

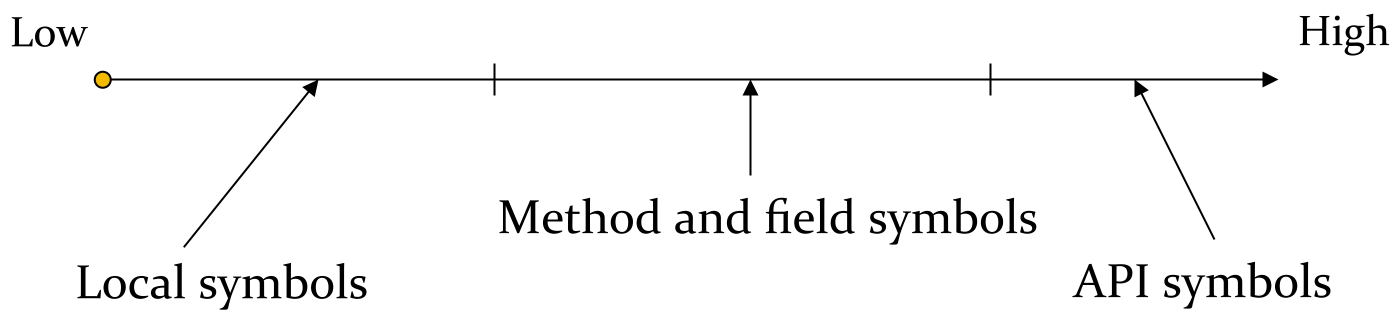
Quantitative Type Inhabitation Problem

Given a type environment Γ, a type τ and some calculus, is there are an expression *e* such that $\Gamma \vdash e : \tau$, and such that e is the "best possible"

- to all type assumptions we assign the weight
- a lower the weight indicates a more relevant term
- weight of a term or a type is computed as the sum of the weights of all symbols

System of Weights

- Symbol weights – used for ranking solution and for directing the search
- Weight of a term is computed based on
  - precomputed term weights - frequency
  - proximity to the program point where the tool is invoked



Evaluation

- 50 benchmarks
- Expected expressions appears
- Among top 10 snippet in 96%
- As a top snippet in 64%
- Average execution time 145ms
- Weights drastically improve the results

Conclusion

- **InSynth** - the interactively deployed synthesis tool in Eclipse for Scala:
  - Ground and function types
  - Weights indicating preferences
- New succinct representation
- Soundness and relative completeness
- We have found to be fast enough and helpful in synthesizing meaningful code fragments

http://lara.epfl.ch/w/insynth

Thank you