Type Inference 101

Sergey Vinokurov

serg.foo@gmail.com, @5ergv

2016-11-26

Introduction

- Types
- Types and programming
- Why type inference
- The Damas-Hindley-Milner algorithm - implementation, why it's still cool and not scary at all

Types

Initially introduced by Bertrand Russel in Principia Mathematica to constrain sets and prohibit paradoxical sets.

Types

Initially introduced by Bertrand Russel in Principia Mathematica to constrain sets and prohibit paradoxical sets.

Famous Russel's paradox: set of all sets that don't contain themselves, $Y = \{X | X \not\subseteq X\}$

Types

Initially introduced by Bertrand Russel in Principia Mathematica to constrain sets and prohibit paradoxical sets.

Famous Russel's paradox: set of all sets that don't contain themselves, $Y = \{X | X \nsubseteq X\}$

Does it contain itself or not? Which one is true: $Y \subseteq Y$ or $Y \nsubseteq Y$?

Types

Initially introduced by Bertrand Russel in Principia Mathematica to constrain sets and prohibit paradoxical sets.

Famous Russel's paradox: set of all sets that don't contain themselves, $Y = \{X | X \nsubseteq X\}$

Does it contain itself or not? Which one is true: $Y \subseteq Y$ or $Y \nsubseteq Y$?

Neither

Types and programming

But we're interested in software development for the time being.

- Can types help with program development? How?
- Can we get similar betefits using other tools?

Types and programming

But we're interested in software development for the time being.

- Can types help with program development? How?
- Can we get similar betefits using other tools?

Yes, types can help with development of programs. They bring machine-checked guarantees that ensure consistency of the program and eliminate certain classes of bugs.

'Well-typed programs cannot "go wrong"' - quote of Robin Milner, one of the developers of the algorithm this talk aims to introduce you to.

Improving software quality

Thus, types improve quality of the software that we produce. However, there's other well-known method of improving qualify of the software - testing.

Let's see how types compare against tests.

Improving software quality

Tests

Tests - executable code that lives together with the original program and checks its behavior.

Tests assert $\exists x : f(x)$

Improving software quality

Tests

Tests - executable code that lives together with the original program and checks its behavior.

Tests assert $\exists x : f(x)$

Pro

- Can check arbitrary conditions
- Can write tests without changes the original code (provided code is in testable form)

Improving software quality

Tests

Tests - executable code that lives together with the original program and checks its behavior.

Tests assert $\exists x : f(x)$

Pro

- Can check arbitrary conditions
- Can write tests without changes the original code (provided code is in testable form)

Cons

- Must be written by hand/generated by a machine (but someone needs to write the generator!)
- Maintenance
- Needs coverage story to ensure that significant portion of the program is tested

Improving software quality

Types

Type - naively, a set of values

Type checking - *syntactic* method of ensuring consistency of the program by classifying program constituents by types of values they produce

Types assert $\forall x : f(x)$

Improving software quality

Types, continue

Types assert $\forall x : f(x)$

Pro

- Prove things regardless of value
- Make illegal states unrepresentable $=>$ profit
- Has unlimited coverage - any value of particular type will do if typechecker accepts your function
- No need to run the program

Improving software quality

Types, continue

Types assert $\forall x : f(x)$

Pro

- Prove things regardless of value
- Make illegal states unrepresentable => profit
- Has unlimited coverage - any value of particular type will do if typechecker accepts your function
- No need to run the program

Cons

- Testing sophisticated assertions, like "function always produces even integers", requires specialized types and, likely, changes to the soucre code
- Proving assertions not encoded in current types may require significant changes to the source code

Improving software quality

Types, continued

While testing requires writing auxiliary code that tests the original program, introducing types into program almost certainly involves modifications of the original program to make it typecheck.

Some of the costs of introducing types are alleviated by type inference:

- Must annotate program with types
- Must maintain annotations when program changes

Improving software quality

Type inference

Some of the costs of introducing types are alleviated by type inference:

- Must annotate program with types
- Must maintain annotations when program changes

Type inference - automated process of assigning types to the program, guided by the program structure.

One of the famous algorithms for type inference is the Damas-Hindley-Milner algorithm.

The Damas-Hindley-Milner algorithm

A bit of history

History: discovered independently by mathematician Roger Hindley in 1969 and computer scientist Robin Milner in 1978. Robin Milner used the algorithm in the programming language of his own development, named ML, short for Meta Language.

Third surname in the algorithm name is due to Louis Damas, who contributed a close formal analysis and proof of the method in his PhD thesis.

The Damas-Hindley-Milner algorithm

Interesting properties

The algorithm works with sufficiently expressive type system and guarantees inference for any program.

- The algorithm is complete - it can infer type of any syntactically valid programs
- The algorithm infers most general type, also called the *principal type*
- Time complexity is *exponential*, i.e. $O(2^n)$, in the size of the processed term, but algorithm is nonetheless widely used in programming language implementations. Exponential processing time is triggered by pathological programs that are never written by hand.

Preliminaries - expression language

Intro

The language we're inferring types for is pretty minimalistic, yet expressive. It has following forms

- constants
- if-expressions
- binary primitives: addition, multiplication, equality comparison
- variables
- function application
- lambda abstraction (unnamed functions)
- let-expressions
- recursive let-expressions

Preliminaries - expression language

Sample programs

- 0, 1, *true*
- $1 + 2 \cdot 3$
- $\lambda x \,.\, x + 2$
- $\lambda f\ x \,.\, f(x) == 0$
- $\lambda x \,.\, \text{let } y = x \cdot x \text{ in } y \cdot y$
- $\text{letrec } f = \lambda n \,.\, \text{if } n == 0 \text{ then } 1 \text{ else } n \cdot f(n + (-1)) \text{ in } f(5)$

Preliminaries - type system

Atomic types

The algorithm deals with expressive type system that we should introduce first. As basic building blocks we have atomic types.

Atomic types - primitive types available in every programming language, e.g. booleans, integers, strings, etc.

Preliminaries - type system

Composite types

Next we have means of combining atomic types to get new types.

Composite types - types defined inductively over other types. For example, functions are composite type - they have argument and result. We denote functions as

$$\text{is\_even} : \mathit{int} \rightarrow \mathit{bool}$$

Preliminaries - type system

Parametric polymorphism

Our types can have universally quantified variables in them. E.g. function that takes two arguments and returns first will have type

$$\text{compose} : \forall \alpha\ \beta\ \gamma\ .\ (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

We are free to use any type in place of α, β and γ, provided these variables are substituted to the same value everywhere in polymorphic type, e.g. using substitutions

$$\alpha \mapsto \text{int},\ \beta \mapsto \text{bool},\ \gamma \mapsto (\text{int} \to \text{int})$$

we can obtain

$$\text{compose} : (\text{bool} \to (\text{int} \to \text{int})) \to (\text{int} \to \text{bool}) \to \text{int} \to (\text{int} \to \text{int})$$

Preliminaries - type system
Parametric polymorphism

Parametricity allows to generalize functions and make them work over any types. Not all function can be generalized. For example, given function

$$\mathsf{f} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$

$$\mathsf{f} = \lambda x\ y\ .\ y + y$$

we may generalize first argument to arbitrary type because it's not used.

However, we must leave second argument as-is because it's used as a number.

Preliminaries - type system

Parametric polymorphism

In order to account for parametric polymorphism our types can have variables in them.

Preliminaries - type system

Type schemes

So, types can be generalized up to some point. The most general form of a type is called **principal type**. The Damas-Hindley-Milner algorithm is guaranteed to find pricipal type, if one exists.

The Damas-Hindley-Milner algorithm

Standard notation

The standard notation to describe typing rules is *natural deduction* due to Gerhard Genzen.

Example:

$$\frac{\text{All men are mortal} \qquad \text{Socrates is man}}{\text{Socrates is mortal}}$$

$$\frac{A \rightarrow B \qquad A}{B} \text{ [Modus Ponens]}$$

The Damas-Hindley-Milner algorithm

Standard notation

The standard notation to describe typing rules is *natural deduction* due to Gerhard Genzen.

Example:

$$\frac{\text{All men are mortal} \qquad \text{Socrates is man}}{\text{Socrates is mortal}}$$

$$\frac{A \to B \qquad A}{B}\,[\text{Modus Ponens}]$$

General form:

$$\frac{\text{Assumption}_1 \qquad \text{Assumption}_2 \qquad \dots}{\text{Conclusion}}\,[\text{Rule name}]$$

The Damas-Hindley-Milner algorithm

Constants - booleans

The algorithm is defined inductively over syntax tree of the program. Each construct gets its own rule.

Constants have no assumptions

$$\frac{}{\Gamma \vdash \mathsf{true} \ : \ \mathsf{bool}} \ [\text{Cst-true}] \qquad \frac{}{\Gamma \vdash \mathsf{false} \ : \ \mathsf{bool}} \ [\text{Cst-false}]$$

Funny $\Gamma \vdash$ notation means 'in the environment Γ'.

The Damas-Hindley-Milner algorithm

Constants - integers

Integer constants have no assumptions either

$$\frac{\text{n is integer}}{\Gamma \vdash n \ : \ \text{int}}\,[\text{Cst-integer}]$$

The Damas-Hindley-Milner algorithm

If expressions

If expressions require that condition to have boolean type and branch types must match.

$$\frac{\Gamma \vdash c \ : \ \mathsf{bool} \qquad \Gamma \vdash t \ : \ \alpha \qquad \Gamma \vdash f \ : \ \alpha}{\Gamma \vdash \mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ f \ : \ \alpha}\ \text{[Expr-if-then-else]}$$

The Damas-Hindley-Milner algorithm

Binary primitives

Addition, multiplication and equality work over numbers.

$$\frac{\Gamma \vdash a \ : \ \mathsf{int} \qquad \Gamma \vdash b \ : \ \mathsf{int}}{\Gamma \vdash a + b \ : \ \mathsf{int}} \text{[Expr-add]}$$

$$\frac{\Gamma \vdash a \ : \ \mathsf{int} \qquad \Gamma \vdash b \ : \ \mathsf{int}}{\Gamma \vdash a \cdot b \ : \ \mathsf{int}} \text{[Expr-mul]}$$

$$\frac{\Gamma \vdash a \ : \ \mathsf{int} \qquad \Gamma \vdash b \ : \ \mathsf{int}}{\Gamma \vdash a == b \ : \ \mathsf{int}} \text{[Expr-eq]}$$

The Damas-Hindley-Milner algorithm

Variables

Variables cannot be typed by itself, they get their meaning from the context - the environment Γ.

$$\frac{(v, \tau) \in \Gamma}{\Gamma \vdash v \ : \ \tau} \text{[Expr-var]}$$

The Damas-Hindley-Milner algorithm

Example

We're given environment $\Gamma = [(x, \mathsf{int})]$.

Let's find type of expression $1 + x$.

$$\frac{}{\Gamma \vdash 1 + x \ : \ \mathsf{int}}$$

The Damas-Hindley-Milner algorithm

Example

We're given environment $\Gamma = [(x, \mathsf{int})]$.

Let's find type of expression $1 + x$.

$$\frac{\Gamma \vdash 1 \ : \ \mathsf{int} \qquad \Gamma \vdash x \ : \ \mathsf{int}}{\Gamma \vdash 1 + x \ : \ \mathsf{int}} \text{[Expr-add]}$$

The Damas-Hindley-Milner algorithm

Example, continued

We're given environment $\Gamma = [(x, \mathsf{int})]$.

Let's find type of expression $1 + x$.

$$\dfrac{\dfrac{1 \text{ is integer}}{\Gamma \vdash 1 \ : \ \mathsf{int}}\,[\text{Cst-integer}] \qquad \Gamma \vdash x \ : \ \mathsf{int}}{\Gamma \vdash 1 + x \ : \ \mathsf{int}}\,[\text{Expr-add}]$$

The Damas-Hindley-Milner algorithm

Example, continued

We're given environment $\Gamma = [(x, \mathrm{int})]$.

Let's find type of expression $1 + x$.

$$\dfrac{\dfrac{1 \text{ is integer}}{\Gamma \vdash 1 \ : \ \mathrm{int}}\,[\text{Cst-integer}] \qquad \dfrac{(x, \mathrm{int}) \in \Gamma}{\Gamma \vdash x \ : \ \mathrm{int}}\,[\text{Expr-var}]}{\Gamma \vdash 1 + x \ : \ \mathrm{int}}\,[\text{Expr-add}]$$

The Damas-Hindley-Milner algorithm

Function application

Function application ensures that only function are applied. In addition, it checks that function is applied to the correct argument.

$$\frac{\Gamma \vdash f \ : \ \alpha \rightarrow \beta \qquad \Gamma \vdash x \ : \ \alpha}{\Gamma \vdash f(x) \ : \ \beta} \text{[Expr-app]}$$

The Damas-Hindley-Milner algorithm

Lambda abstraction

Lambda abstraction ensures that it's body type-checks in the extended environment, where argument is bound.

$$\frac{\Gamma,\ x := \alpha \vdash e \ :\ \beta}{\Gamma \vdash \lambda x\ .\ e \ :\ \alpha \to \beta}\ \text{[Expr-lam]}$$

The Damas-Hindley-Milner algorithm

Specialiation

A special rule is available for specializing types schemes.

$$\frac{\Gamma \vdash e \ : \ \alpha \qquad \alpha \sqsubseteq \beta}{\Gamma \vdash e \ : \ \beta} \text{[Expr-spec]}$$

The $\sqsubseteq$ relation denotes when one type is an instance of another, e.g. $(\mathsf{int} \rightarrow \mathsf{int}) \sqsubseteq (\alpha \rightarrow \alpha)$

The Damas-Hindley-Milner algorithm

Generalization

Generalization allows to add quantification over variables not captured by the context.

$$\frac{\Gamma \vdash e \ : \ \alpha \qquad \beta \notin \Gamma}{\Gamma \vdash e \ : \ \forall \beta \ . \ \alpha} \text{[Expr-gen]}$$

References

- Programming and Programming Languages - Krishnamurthi S., Lerner B., Politz J. G.
- Wikipedia articla - Hindley-Milner type system

Thank you!

Thank you!

Questions?