TypingNet

Project Progress and Formalizations

ANONYMOUS AUTHOR(S)

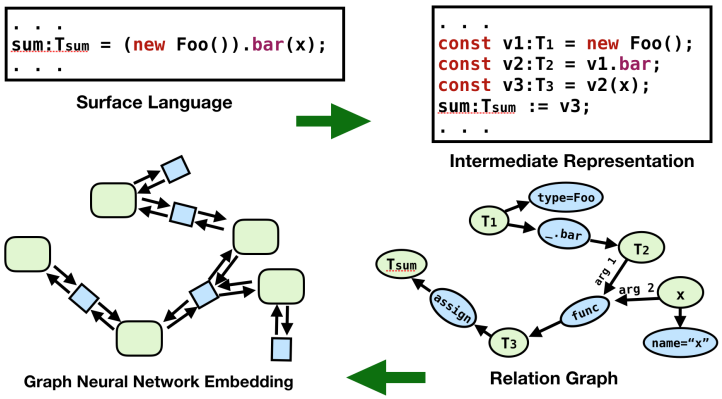
....

Additional Key Words and Phrases: type inference, NLP, graph neural networks

1 TYPE INFERENCE WORKFLOW

Our deep learning-based type inference workflow consists of the following steps:

(1) **Surface Language Parsing**: Parse Javascript/Typescript programs into a Scala DSL.
(2) **Intermediate Representation Translation**: This IR is similar to static single assignment form. We generate variable definitions for all intermediate expressions appeared in the original programs, and this in turn allows us to also generate type variables for all places where a type needs to be inferred.
(3) **Predicate Graph Construction**: From the IR, we extract various relations between type variables (such as subtyping or equality constraints) and convert them into *predicates* over type variable nodes. NLP features such as variable/field names are also encoded as unary predicates. This gives us a predicate graph consisting of two kinds of nodes: type variable nodes and predicate nodes.
(4) **Graph Neural Network Embedding**: Using the relation graph, we apply graph neural networks to obtain vector representations of each type variable node. This is an iterative process: each node is initialized with a vector representation computed from its node features, and in each subsequent iteration, is updated using messages sent along all incoming relation edges.
(5) **Type Decoding**: Finally, we feed the obtained vector representations of each type variable to another neural network to predict its actual type. During training time, a user-annotated label type can be used to compute a loss gradient, which is back-propagated through the graph neural network embedding process to update all network weights.



2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

x, l, t	∈ <identifiers>	
α	∈ <concrete types> ∪ {?}	
e ::=		expressions:
	x	variables
	c	constants
	e(e, ..., e)	function call
	{ l: e, ..., l: e }	object literal
	e.l	field access
	if e **then** e **else** e	ternary expression
S ::=		statements:
	var x: α = e	variable introduction
	e := e	assignment
	[**return**] e	expression statement
	if e **then** S **else** S	conditional
	while e **do** S	while loop
	{ S; ... ; S }	block
f ∈	**function** x(x: α, ... , x: α): α ≡ S	function definition
	class t [**extends** t] { l: α, ... , l: α, f, ... , f }	class definition

Fig. 1. Surface Language Syntax

x, l, t	∈ <identifiers>	
α	∈ <type variables>	
e ::=		expressions:
	x	variables
	c	constants
	x (x , ... , x)	function call
	{ l: x , ..., l: x }	object literal
	x .l	field access
	if x **then** x **else** x	ternary expression
S ::=		statements:
	var x: α = e	variable introduction
	x := x	assignment
	return x	return statement
	if x **then** S **else** S	conditional
	while x **do** S	while loop
	{ S; ... ; S }	block
f ∈	**function** x : α (x: α, ... , x: α): α ≡ S	function definition
	class t : α [**extends** t] { l: α, ... , l: α, f, ... , f }	class definition

Fig. 2. Intermediate Representation Syntax

2 SURFACE LANGUAGE

Grammar is shown in figure 1.

3 INTERMEDIATE REPRESENTATION

Grammar is shown in figure 2.

l, n	∈ <identifiers>	
α	∈ <type variables>	
τ	∈ <concrete types>	
r ::=		relations:
	$\alpha \equiv \alpha$	equality
	$\alpha <: \alpha$	subtype
	$\alpha := \alpha$	assignable
	$\alpha \sim \textbf{bool}$	used as bool
	$\alpha_{\text{name}} = n$	has name
	α **extends** α	inheritance
	$\alpha \equiv \tau$	is type
	$\alpha \equiv (\alpha, ..., \alpha) \rightarrow \alpha$	is function type
	$\alpha \equiv \alpha(\alpha, ..., \alpha)$	application
	$\alpha \equiv \{l : \alpha, ..., l : \alpha\}$	is object type
	$\alpha \equiv \alpha.l$	field access

Fig. 3. Predicates over Type Variables

4 PREDICATE GRAPH

The predicate grammar is shown in figure 3. The rules to generate predicates are shown in figure 4.

A APPENDIX

Text of appendix ...

$$\frac{}{P(\Phi, \textbf{var}\ x : \alpha = e) = (\alpha \equiv \text{type}(\Phi, e))} \text{ (variable intro)}$$

$$\frac{\Phi[x_i] = \alpha_i}{P(\Phi, x_1 := x_2) = (\alpha_1 := \alpha_2)} \text{ (assignment)}$$

$$\frac{\Phi[x_1] = \alpha_1 \quad \Phi[\textbf{return}] = \alpha_2}{P(\Phi, \textbf{return}\ x_1) = (\alpha_1 <: \alpha_2)} \text{ (return)}$$

$$\frac{S = \textbf{if}\ x_1\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \quad \Phi[x_1] = \alpha_1}{P(\Phi, S) = (\alpha_1 \sim \textbf{bool} \wedge P(\Phi, S_1) \wedge P(\Phi, S_2)} \text{ (conditional)}$$

$$\frac{S = \{S_1, ..., S_n\} \quad \Phi' = \text{update}(S, \Phi)}{P(\Phi, S) = (P(\Phi', S_1) \wedge ... \wedge P(\Phi', S_n))} \text{ (block)}$$

$$\frac{S = \textbf{function}\ x_f : \alpha_f(x_1 : \alpha_1, ..., x_n : \alpha_n) : \alpha_r \equiv S_r \quad \Phi' = \Phi[\textbf{return} \Leftarrow \alpha_r]}{P(\Phi, S) = (\alpha_f \equiv (\alpha_1, ..., \alpha_n) \rightarrow \alpha_r \wedge P(\Phi', S_r))} \text{ (function)}$$

$$\frac{S = \textbf{class}\ t : \alpha_c\ \textbf{extends}\ d\{l_{1...n} : \alpha_{l_1...l_n}, f_{1...m} : \alpha_{f_1...f_m}\} \quad \Phi' = \Phi[\textbf{this} \Leftarrow \alpha_c]}{P(\Phi, S) = (\alpha_c \equiv \{l_{1...n} : \alpha_{l_1...l_n}, f_{1...n} : \alpha_{f_1...f_n}\} \wedge P(\Phi', f_1) \wedge ... \wedge P(\Phi', f_m))} \text{ (class)}$$

Fig. 4. Predicate Generation Rules