JambaJuice

a small, functional language with modular type inference

Eric Feng, Emily Sillars
Columbia University
{ef2648,ems2331}@columbia.edu

ABSTRACT

We introduce a small, functional programming language equipped with a prototype modular type inference library, *PLCgen*, designed to alleviate the burden on developers from having to implement their own type inference passes. This library provides an API that allows developers to specify essential information about their language, enabling the automatic generation of type inference routines.

Traditionally, developers have had to invest significant effort in crafting custom-type inference mechanisms for their programming languages. This process often involves grappling with the implementation of complex algorithms, which can be time-consuming and error-prone. Our proposed library aims to address this challenge by offering a standardized and reusable solution for type inference.

By leveraging the API provided by our library, developers can seamlessly integrate type inference capabilities into their language implementations. The library harnesses this information to generate Hindley-Milner typing constraints in Prolog and then utilizes Prolog's constraint solver to resolve them. We chose Prolog for this use case for its succinctness in expressing relevant constraints, which is helpful for both constraint generation, checking for correctness, and extending library capabilities. This approach not only offloads the burden of implementing type inference passes but also ensures accurate and reliable type inference across different language features and constructs.

We hope that our prototype modular type inference library presents a step towards easing the development process for statically-typed functional programming languages. By providing a standardized solution for type inference, we hope to empower developers to focus on language design and implementation, while reducing the complexity and maintenance overhead associated with custom type inference passes.

1 JAMBAJUICE LANGUAGE

1.1 Overview

Functional programmers should find the syntax and usage of *JambaJuice* familiar relative to other functional languages. To gain some intuition of the language by examining an example program.

```
jambatime fact n = {
    if n == 0 { 1 }
    else { n * fact (n - 1) }
}

jambajuice = {
    let localFib = fix \localFib -> \b -> (if b == 0
        ↪ {1} else {b * localFib (b - 1)}) in {
        if (fact 8 > 10)
            { localFib 5 + 10 }
        else
            { 8 }
    }
}
```

In the program above, *jambajuice* and *fact* are top-level definitions respectively. *jambajuice* is the key point of entry to *JambaJuice* programs and as such acts as the "main" function in other programming languages. Observe that *fact* is a recursive definition calculating factorials. Recursive definitions are prepended with the *jambatime* keyword with it serving as our version of the familiar *let rec*. *Let recs* are desugared to utilize the fix point operator, which may also be invoked explicitly, as noted by *localFib* in *jambajuice*. Top-level non-recursive definitions such as *jambajuice* are simply top-level *let* expressions that are desugared into nested lambdas. Top-level recursive definitions such as *fact* are *let recs* which are similarly desugared into nested lambdas, but with a Fix point operator wrapped around them.

1.2 Core Language

The core language of *JambaJuice* is based on an extension of the untyped lambda calculus including the fix point operator, basic base types (booleans, integers), let expressions, and basic arithmetic operations much like the *Poly* programming language from Stephen Diehl's *Write You A Haskell* [1]. Unlike *Poly*, *JambaJuice* infers types by generating and solving constraints in Prolog through our library rather than providing a language-specific implementation of Hindley Milner inference. We chose *Poly* as a reference to base the language features of *JambaJuice* because it is similar in scope with respect to the language features we wished to support type inference for via our library. By using *Poly* as a reference, we were able to reuse much of its core logic in the Lexer, Parser, and Evaluation modules for *JambaJuice's* interpreter, allowing us to focus most of our attention on our type inference library—with changes as necessary owing to the differences in user-level syntax between the two languages and patching small bugs in the reference implementation. With this in mind, *JambaJuice* programs eventually desugar to the same core syntax as *Poly*:

```
type Var = String

data Expr
  = Var Var
  | App Expr Expr
  | Lam Var Expr
```

E6998: Types Languages Compilers, ,
2023.

```
  | Let Var Expr Expr
  | Lit Lit
  | If Expr Expr Expr
  | Fix Expr
  | Op Binop Expr Expr
  deriving (Show, Eq, Ord)

data Lit
  = LInt Integer
  | LBool Bool
  deriving (Show, Eq, Ord)

data Binop = Add | Sub | Mul | Eql | Neq | Lt | Le |
↪  Gt | Ge
  deriving (Eq, Ord, Show)

type Decl = (String, Expr)

data Program = Program [Decl] Expr deriving (Show, Eq)
```

1.3 Implementation

JambaJuice programs are lexed and parsed to the core syntax shown using *Parsec*. Afterward, the AST is passed to our library for type inference and type checking before being interpreted.

The interpreter evaluates expressions in a given environment (*TermEnv*) of variable bindings, represented as a `Map` from variable names to values of type `Value`. The possible values of the interpreted language consist of the following:

```
type TermEnv = Map.Map String Value

data Value
  = VInt Integer
  | VBool Bool
  | VClosure String Expr TermEnv
```

Where a closure consists of a lambda expression and the environment in which it was defined.

To interpret, we take in an expression and an environment, then evaluate the expression to a value by pattern matching on the appropriate type of expression and evaluating relevant subexpressions recursively. The initial environment of the program is an empty map, and upon evaluation, we return the resulting value and an updated environment with the binder bound to the value. The final value that we present to the user is the value belonging to the "main" function (keyword *jambajuice*).

2 HINDLEY-MILNER TO PROLOG

2.1 Overview

We first worked through some examples from Lerner's type inference notes [3] to gain an intuition of the Hindley Milner type inference process. For the basis of our formal translation, we use Edwards' description of the Hindley-Milner typing rules from lecture [2]. The extraction and insertion of type schemes from/into the environment gamma Γ are handled by our modular type inference library, *PLCgen*. Aside from those parts, each rule translates directly to Prolog, provided we assume unique type variables assigned to every node in the source program's AST, and the following clauses:

```
arrow([_,_]).
snd(A,B):-A=[H1,H2],B=H2.
fst(A,B):-A=[H1,H2],B=H1.
```

A function type $T_1 \rightarrow T_2$ is represented by a two-element list validated by the fact "arrow". The output type of a function type can be extracted using the rule "snd". The input type of a function type can be extracted using the rule "fst".

```
hasTypeScheme(F,FORALL,T) :- generalize(F,FORALL,T).
```

Function F has a type scheme represented by a list of quantified variables FORALL, and a type T. We can give F this type scheme provided we can generalize the type T.

```
generalize(FUNC_I,FORALL,T):- hasType(FUNC_I,T),
include(var,T,FORALL).
generalize(FUNC_I,[],T):-
↪  hasType(FUNC_I,T),T\=[],T\=[H|T].
```

We can generalize T into a scheme provided it is a type (some function has type T), and the list of quantified types FORALL contains the free variables in T.

The filter predicate in Prolog requires lists as input, so we need a second rule for generalize that creates a degenerate type scheme from a non-arrow type. Also, an empty list or list of one variable is never a valid type.

```
instantiates(Y,F) :- hasTypeScheme(F,FORALL,T),Y=T.
```

Type Y instantiates type scheme F provided its type unifies with T.

2.2 Application

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}\ \text{app}$$

We construct a general App AST node with unique type variables for each subnode:

```
(App (e1 :: X1) (e2 :: X2) ) :: X3
```

To apply the app rule, the type of e_1 must be a function type, and the input type of e_1 must be the type of e_2. Then we can conclude the type of $e_1\ e_2$ is the return type of e_1.

Prolog translation:

```
arrow(X1),
fst(X1,X2),
snd(X1,X3).
```

2.3 Lambda Abstraction

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x\, .\, e : \tau_1 \rightarrow \tau_2}\ \text{abs}$$

We construct a general Lambda AST node with unique type variables for each subnode:

```
(Lam (Var "x" :: X1) (e :: X2) ) ::X3
```

Using the abs rule, we say that given a binder "x" of type X1 and a body expression e of type X2, we can conclude a lambda abstraction with binder "x" and body e has type $X1 \rightarrow X2$.

Prolog Translation:

```
arrow(X3),
fst(X3,X1),
snd(X3,X2).
```

2.4 Let

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}\ \text{let}$$

We construct a general Let AST node with unique type variables for each subnode:

```
(Let ((Var "x") :: X1) (e1 :: X2) (e2 :: X3)) :: X4
```

Using the let rule, we can say that provided e_1 has type X2, the binder "x" has a type scheme formed from the generalization of X2, and e_2 has type X3, the lambda abstraction node containing binder "x", binder value e_1, and body e_2 has the type of its body.

Prolog translation:

```
X1 = X2,
X4 = X3.
```

Note that the generalization step corresponds to a *PLCgen* internal function call that inserts a type scheme into the current environment. See section 3 for more details.

2.5 Var

$$\frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau}\ \text{var}$$

We construct a general Var AST node with a unique type variable:

```
(Var "x") :: X5
```

The var rule says that if binder "x" has type scheme σ in the environment, we can assign a Var node containing binder "x" the type X1, where X5 instantiates the type scheme σ.

When the interpreter encounters a var node in its source program's AST, it calls the *PLCgen* API function *genVar*. This function looks up "x" in the environment, and generates the following Prolog constraints in each case:

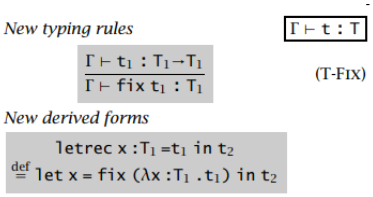
(1) binder "x" refers to a let binder (has a type scheme)
There exists a let expression of the form
`(Let ((Var "x") :: X1) (e1 :: X2) (e2 :: X3)) :: X4`
so our instance of binder "x" must have a type that instantiates the scheme associated with type X1.
Prolog Translation:
`copy_term(X1,X5).`
Note that `copy_term` is a built-in predicate in SWI Prolog that creates a copy of its left argument with fresh variables, and then unifies this copy with the right argument.

(2) binder "x" refers to a lambda binder (has a type)
There exists a lambda expression of the form
`(Lam (Var "x" :: X1) (e :: X2) ) ::X3`
so our instance of binder "x" must have a type that instantiates the degenerate scheme (type) associated with type X1.
Prolog Translation:
`X5 = X1.`

(3) binder "x" refers to a top-level function (has a type scheme)
If binder "x" refers to some top-level function, there exists a compound term in our Prolog knowledge base such that function x has some type. Our instance of binder "x" must instantiate this function's type scheme.
Prolog translation:
`instantiates(X5, x).`

2.6 Extending with Recursion

To extend our Hindley-Milner type system with recursion, we use the fixed point operator typing rules described by Pierce in [4].



T-Fix says that provided an expression t_1 has type $T_1 \rightarrow T_1$, The fixed point of t_1 has type T_1. Note that this rule requires the argument of fix be a lambda abstraction (or potentially a data constructor).

The derived forms illustrate explicitly that the argument to Fix is also a special lambda expression in which the binder is the name of the recursive function being defined.

We construct a general Fix AST node:

```
(fix (Lam (Var "x")::X1 e :: X2)::X3 )::X4
```

Using the T-Fix rule, the expression t_1 corresponds to `(Lam (Var "x")::X1 e :: X2)::X3`.

Prolog translation:

```
arrow(X1),
snd(X3,X4).
```

Using "fst" instead of "snd" here would work just as well.

3 MODULAR TYPE INFERENCE

We define a modular type inference library in Haskell called *PLCgen* (short for Prolog Constraint Generation). The goal of this library is to perform Hindley-Milner type inference for any functional language interpreted or compiled with Haskell.

Given some basic language-specific and program-specific information from the source language interpreter, *PLCgen* generates a corresponding Prolog knowledge base, queries it to find the type of each node in the source program's AST, and returns the results.

Current limitations of our library include a lack of support for Algebraic Data Types and name shadowing.

3.1 ASTs as a Universal Connection

To give PLCgen a common interface to interact with any source language, we rely on the user providing PLCgen with unique node numbers for ever node in their source program's abstract syntax tree. Once PLCgen is given these node numbers, it creates a table from node ID number to unique type variable that it maintains throughout the constraint generation process.

3.2 User API

Each user API function modifies PLCgen's internal state in some way. This internal state is outlined in more detail in the next section, but the most relevant information for users to be aware of when harnessing the API is

(1) Top-level functions are processed one at a time
(2) The library must know about all nodes in the program before starting to process any function
(3) Per-node API calls should be performed as part of an AST traversal
(4) The library's internal state keeps track of the Hindley-Milner typing environment Γ.

Examples of per-node API Functions

- `enterApp :: NodeID -> NodeID -> NodeID -> PLC ()`
  Generates Prolog constraints for an App AST node
- `enterLet :: NodeID -> String -> NodeID -> NodeID -> NodeID -> PLC ()`
  Generates Prolog constraints for a Let AST node
- `genFix :: NodeID -> NodeID -> NodeID -> PLC ()`
  Generates Prolog constraints for a Fix AST node
- `genVar :: String -> NodeID -> PLC ()`
  Generates Prolog constraints for a Var AST node

In addition to functions that generate Hindley-Milner constraints for basic AST nodes, we provide two general API calls the user can use to add custom language features and typing rules to PLCgen's type inference system.

(1) `unifyNodeWithType NodeID -> String -> PLC ()`
Retrieves the type variable associated with the given NodeID and unifies it with a built-in type.
(2) `unifyNodeWithNode NodeID -> NodeID -> PLC ()`
Retrieves the type variables associated with the given IDs and unifies them.

Our JambaJuice interpreter takes advantage of these API functions to type check its `if-else` and `Op` nodes.

3.3 Internal State

PLCgen maintains a state consisting of language-specific information (built-in functions), and program-specific information (the node IDs associated with each node in the source program's AST, the Prolog constraints associated with the source program). It also keeps track of temporary function-specific information, as PLCgen generates constraints for top-level functions one at a time.

Function-specific information includes the set of all node IDs corresponding to the current function's subtree, the Prolog constraints associated with this function, and a lookup table from variable names to pertinent variable information(specifically, whether the binder was introduced by a let expression, lambda expression, or refers to a top-level function, and with the former two cases, the node ID of the node in which the binder was first introduced).

When PLCgen concludes constraint generation for a top-level function (sparked by the `exitFunc` user API call), it appends all the function-specific constraints to its set of program-specific constraints.

Once all functions have been processed, PLCgen dumps its program constraints into a Prolog file, prefixed by the helper constraints described in section 2.1, and concluded with the line

```
:- initialization
↪  forall((hasType(X,Y),not(isTopLevelDef(X))),
↪  (write(X),write(' '), writeln(Y))),halt().
```

which automatically prints the results of type inference upon loading the knowledge base.

3.4 Backend

After generating the source program's corresponding Prolog file, PLCgen spawns a swipl Prolog process to load and query the knowledge base. The swipl Prolog output is piped back into PLCgen, which then parses the plain text results into a map from NodeIDs to Types, where

```
type NodeID = Integer
data Type = String | Arrow Type Type
```

If the size of the results table is the same size as the node Id to type variable table from PLCgen's internal state, the program type checks. Finally, the results table is returned to the user along with a notice of success or failure.

4 FUTURE WORK

The JambaJuice programming language and library are prototypes demonstrating the usage of modular type inference with respect to our library. With that in mind, there are many potential areas for future work and improvement. Some examples include:

- Language Extensions: Expanding the core language of JambaJuice by introducing new language constructs such as algebraic data types and case expressions.
- Extending Core Library Functionality: Add user API calls to support typing algebraic data types. Improve upon and add to the general API function calls that allow users to introduce custom language features/-typing rules. For example, `NodeInstantiatesNode` and `NodeInstantiatiesTypeScheme` function calls could be helpful.
- Optimizing Performance: Currently, our library generates prolog constraints that can take a noticeable amount of time to solve depending on the source program. This suggests our generated prolog files are not realistic for larger, more complex programs. Investigating optimization techniques for prolog constraint generation/solving may enhance the efficiency and scalability of our library.
- Better Error Messages: To make PLCgen a feasible tool to incorporate into interpreters and compilers, specific error messages about the cause of type-checking failure are needed.

5 CONCLUSION

Our simple functional language, JambaJuice, demonstrates how the implementation of type inference in functional languages may be simplified using our modular type inference library. By providing a modular approach and leveraging Hindley Milner typing constraints along with Prolog's constraint solver, the library offloads the burden on developers to write custom-type inference passes. By leveraging our library's API, developers may focus more on language design and implementation, while still ensuring accurate and reliable type inference results.

REFERENCES

[1] Stephen Diehl. 2015. Write You A Haskell. https://smunix.github.io/dev.stephendiehl.com/fun/index.html
[2] Stephen A. Edwards. 2023. The Hindley-Milner Type System. http://www.cs.columbia.edu/~sedwards/classes/2023/6998-spring-tlc/hindleymilner.pdf
[3] Benjamin Lerner. 2019. Lecture 11: Type Inference. https://course.ccs.neu.edu/cs4410sp19/lec_type-inference_notes.html
[4] Benjamin C. Pierce. 2007. *Types and Programming Languages.* The MIT Press.

APPENDIX

(Continued on the next page in single column format)

An example JambaJuice Program

```
jambatime g x = {
    g x
}

jambajuice = {
    let id = \x -> x in {
        if (id true) {id 4} else {id 7}
    }
}
```

Generated Prolog Constraints

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          these predicates should be at the top of every typechecking file              %%

use_module(library(apply)).       % to import include
:- discontiguous hasType/2.       % ignore discontiguous warnings
:- discontiguous hasTypeScheme/3. % ignore discontiguous warnings
:- discontiguous isTopLevelDef/1. % ignore discontiguous warningsgit
:- style_check(-singleton).       % ignore singleton var warnings

% generate a type scheme for a top level definition if possible
hasTypeScheme(F, FORALL, T) :- generalize(F, FORALL, T).

% convert the type of a top level definition into a type scheme
generalize(FUNC_I, FORALL, T):- hasType(FUNC_I, T), include(var, T, FORALL).
generalize(FUNC_I, [], T):- hasType(FUNC_I, T), T\=[], T\=[H|T].

% convert the type scheme of a top-level definition into a type
instantiates(Y,F) :- hasTypeScheme(F, FORALL, T), Y=T.
% we use copy_term/2 to instantiate the type scheme of a local definition (let bindings)

% helper constraints
arrow([_,_]).
snd(A,B):-A=[H1,H2],B=H2.
fst(A,B):-A=[H1,H2],B=H1.
%%                                                                                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7):-(snd(X1,X0),arrow(X2),arrow(X1),fst(X1,X2),snd(X1,X3),X2=X2,arrow(X3),fst(X3,X4),
snd(X3,X5),X4=X4,arrow(X6),fst(X6,X7),snd(X6,X5),X6=X2,X7=X4).
hasType(g , X0):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
isTopLevelDef(g).
hasType(node_0 , X0):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_1 , X1):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_2 , X2):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_3 , X3):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_4 , X4):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_5 , X5):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_6 , X6):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
hasType(node_7 , X7):-g_typechecks(X0,X1,X2,X3,X4,X5,X6,X7).
jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22):-(X9=X10,X8=X13,arrow(X10),fst(X10,X11),
snd(X10,X12),X11=X11,X12=X11,X14=bool,X17=X20,X13=X17,X13=X20,arrow(X15),
fst(X15,X16),snd(X15,X14),X16=bool,arrow(X18),fst(X18,X19),snd(X18,X17),X19=int,
arrow(X21),fst(X21,X22),snd(X21,X20),X22=int,copy_term(X9,X9),copy_term(X9,X15),
copy_term(X9,X18),copy_term(X9,X21)).
hasType(jambajuice , X8):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
```

```
isTopLevelDef(jambajuice).
hasType(node_8 , X8):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_9 , X9):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_10 , X10):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_11 , X11):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_12 , X12):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_13 , X13):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_14 , X14):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_15 , X15):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_16 , X16):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_17 , X17):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_18 , X18):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_19 , X19):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_20 , X20):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_21 , X21):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).
hasType(node_22 , X22):-jambajuice_typechecks(X8,X9,X10,X11,X12,X13,X14,X15,X16,X17,X18,X19,X20,X21,X22).

% output results of typechecking!
:- initialization forall((hasType(X,Y),not(isTopLevelDef(X))), (write(X),write(' '), writeln(Y))),halt().
```

Type Inference Results

```
node_0 [_2538,_2544]                   %% the type of function g
node_1 [[_2538,_2544],[_2538,_2544]]   %% the type of the argument to Fix, (\g -> (\x -> g x))
node_2 [_2538,_2544]
node_3 [_2540,_2546]
node_4 _2436
node_5 _2436
node_6 [_2540,_2546]
node_7 _2436
node_8 int
node_9 [_2524,_2524]                 %% the type of id inside of the Let
node_10 [_2524,_2524]
node_11 _2436
node_12 _2436
node_13 int
node_14 bool
node_15 [bool,bool]                  %% the type of id inside (id true)
node_16 bool
node_17 int
node_18 [int,int]                    %% the type of id inside (id 4)
node_19 int
node_20 int
node_21 [int,int]                    %% the type of id inside (id 7)
node_22 int
```