Type checking with Datalog

Solène Husseini, Liliane-Joy Dandy

EPFL - Formal Verification course

December 22, 2022

Introduction

- Type checker are usually tightly integrated to compilers
- Use Datalog for type-checking for simple type systems

Language presentation

- Scala-like syntax
- Simple functional type system
- Top-level functions for recursion
- Lambdas supported of course

```
type Nat {
  case z()
  case s(Nat)
}

def plus(x: Nat, y: Nat): Nat = {
    x match {
        case z() => y
        case s(x1) => s(plus(x1, y))
    }
}
```

Language presentation

- Custom language
- Custom parser, compiler pass to give unique names to variables
- No evaluator / typechecker yet

- Typechecks non-polymorphic functions
- Can typecheck polymorphic functions "by hand"

Datalog

- Simple declarative language similar to prolog, with constraints
- Order independent
- Relations are finite by construction
- Guaranteed to terminate and answer every query (may take time though)
- Current solver: racket solver

```
typed(X,T) :- app(X,F,A), typed(A,Ta),
              typed(F,Tf), fun(Tf,Ta,T).
var(x).
var(y).
app(a,x,y).
typed(a,T)?
```

Represent AST as datalog facts

- Convert AST to datalog

```
var(x).
app(app0, f, x).
lambda(lbd0, arg, argtype, body).
match(match0, scrut, pattern, body).
```

Inference rules: lambdas and applications

$$\frac{C, x : T_1 \vdash b : T_2}{C \vdash \lambda x : T_1 . b : T_1 \rightarrow T_2}$$

```
typed(X,T) :- lambda(X,V,Tv,B), typed(B,Tb),
              fun(T,Tv,Tb).
```

$$\frac{C \vdash e_1 : T_1 \rightarrow T \qquad C \vdash e_2 : T_1}{C \vdash e_1 e_2 : T}$$

```
typed(X,T) :- app(X,E1,E2), typed(E1,Te),
              fun(Te,T1,T), typed(E2,T1).
```

Inference rules : variables

$$\frac{C(x) = T}{C \vdash x : T}$$

```
typed(X,T) :- var(X), context_typed(X,C), lookup(C,X,T).
```

Need to define 3 new relations :

- context_typed
- bind
- lookup

Context typing relation

```
context_typed(B,C1) :- lambda(E,V,T,B), lookup(C1,V,T1),
                       context_typed(E,C).
context_typed(E1,C) :- app(E,E1,E2), context_typed(E,C).
context_typed(E2,C) :- app(E,E1,E2), context_typed(E,C).
```

Generate context typing rules for constructors :

```
context_typed(E1,C) :- s(E,E1), context_typed(E,C).
```

Inference rules : pattern matching

$$\frac{C \vdash E : T_1, C \vdash P \rightarrow B : T_1 \rightarrow T}{C \vdash E \texttt{ match case } P \rightarrow B : T}$$

$$\frac{C \vdash E \texttt{ match case } P_1 \rightarrow B_1 : T, ..., C \vdash E \texttt{ match case } P_n \rightarrow B_n : T}{C \vdash E \texttt{ match [case } P_1 \rightarrow B_1, ..., \texttt{ case } P_n \rightarrow B_n\texttt{]} : T}$$

```
typed(X,T) :- match(X,S,P,B), typed(S,T1),
              typed(P,T1), typed(B,T).
typed(X,T) :- pattern_match_list(X,E1,E2),
              typed(E1,T), typed(E2,T).
```

Inference rules : pattern matching

```
def pred(x: Nat): Nat = {
    x match {
        case z() => z()
        case s(y) => y
    }
}
```

Need for a specific context for the pattern-matched variables:

- Add a new pattern_matching_ctx relation to add the variables on the left hand side to the context

Inference rules : pattern matching

```
def pred(x: Nat): Nat = {
    x match {
        case z() => z()
        case s(y) => y
    }
}
```

Constraint the type of y in the left-hand side:

- add a new type_constr relation for constructors

  ```
  type_constr(X,nat) :- s(E,X).
  ```

Inference rules : pattern matching

```
def pred(x: Nat): Nat = {
    x match {
        case z() => z()
        case s(y) => y
    }
}
```

```
var(y).
bind(new_ctx,old_ctx,y,T) :- type_constr(y,T).
succ(e1,y).
match(e2,x,e1,y).
pattern_matching_ctx(new_ctx,old_ctx,y).
```

Recursivity

- Recursive functions will not work naively
- Emit type assertion for a fake function name
- Refer to those fake functions when emitting datalog for function calls

```
def foo(x: Nat, y: Bool): Nat = {
    bar(x) // will emit type for bar_ascribed
}
```

```
fun(fn2,Bool,Nat).
fun(fn1,Nat,fn2).
typed(foo_ascribed,fn1).
```

Additional relations

- type_eq : checks for equality between two types :

  ```
  fun(f1,nat,nat).
  fun(f2,nat,nat).
  ```

  Then we should have f1 = f2

```
type_eq(T,U) :- type(T), T=U.
type_eq(T,U) :- type_eq(U,T).
type_eq(T,U) :- type_eq(T,S), type_eq(S,U).
type_eq(T,U) :- fun(T,T1,T2), fun(U,U1,U2),
                type_eq(T1,U1), type_eq(T2,U2).
```

Tricky edge cases

- Typing a term necessitates that it is declared
- Cannot enumerate all funs, would need to create arbitrary identifiers
- Use a trick to type lambdas in particular

```
f = (\n: Nat -> Bool. n(true()))
n: Nat -> Bool
true: Bool
f: (Nat -> Bool) -> Bool
```

- If no type $(Nat \rightarrow Bool) \rightarrow Bool$ is defined, then $typed(f, T)$? will not find T
- Deduce necessary type as we go
- Use the (unique) name of the lambda expr to name its type

```
fun(X,Tv,Tb) :- lambda(X,V,Tv,B), typed(B,Tb).
```

Generate data types as Datalog relations

```
type Nat {
  case z()
  case s(Nat)
}

type(Nat).

typed(z,Nat).
fun(tfn0,Nat,Nat).
typed(s,tfn0).

typed(E,Nat) :- z(E).
typed(E,Nat) :- s(E,E1), typed(E1,Nat).
type_constr(X,Nat) :- s(E,X).

context_typed(E0,C) :- s(E,E0), context_typed(E,C).
```

Generate top-level terms as Datalog relations

```
def plus1(x: Nat): Nat = {
    s(x)
}
def plus1: Nat -> Nat = {
    \x: Nat. s(x)
}
```

```
app(app78,_s7,_x39).
var(_x39).
bind(ctx__plus118,global_ctx,_x39,_Nat1).
lambda(_plus118,_x39,_Nat1,app78).
context_typed(_plus118,global_ctx).
fun(tfn81,_Nat1,_Nat1).
typed(_plus118_ascribed,tfn81).
```

Generic types

```
type List[A] {
  case nil()
  case cons(A, List[A])
}
```

```
type(T) :- type(A), list(T,A).
typed(E,T) :- nil(E,A), list(T,A).
typed(E,T) :- cons(E,Elm,List), typed(Elm,A),
              typed(List,T), list(T,A).
context_typed(E1,C) :- cons(E,E1,L), context_typed(E,C).
context_typed(E2,C) :- cons(E,El,E2), context_typed(E,C).
type_constr(X,T) :- cons(E,X,XS), type_constr(E,T1),
                    list(T1,T).
type_constr(XS,T) :- cons(E,X,XS), type_constr(E,T).
```

Polymorphic functions

```
def map[T, U](f: T -> U, ls: List[T]): List[U] = {
  ls match {
    case nil() => nil()
    case cons(h, t) => cons(f(h), map[T, U](f, t))
  }
}
```

- Polymorphic function
- Generic data types
- Recursivity
- Type inference

Inference rules for polymorphism

$$\frac{C \vdash t : T}{C \vdash \Lambda a.t : \forall a.T} \qquad \frac{C \vdash t : \forall a.T}{C \vdash t[U] : [U/a]T}$$

```
typed(X,T) :- forall_intro(X,E1,A), typed(E1,T1),
               type_gen(T,A,T1).
typed(X,T) :- forall_elim(X,E,A,U), typed(E,T1),
              type_gen(T1,A,T2), replace(T,U,A,T2).
```

2 new relations :

- type_gen
- replace

The type_gen relation

The type $X = \forall a.T$ is expressed

```
type_gen(x,a,t).
```

The replace relation

```
replace(T,U,A,T1) :- type(A), type(U), A=T1, T=U.
replace(T,U,A,T1) :- simple_type(T1), T=T1,
                     type(U), type(A).
replace(T,U,A,T1) :- fun(T1,T2,T3), replace(R2,U,A,T2),
                     replace(R3,U,A,T3), fun(T,R2,R3).
replace(T,U,A,T1) :- type_gen(T1,B,T2),
                     replace(R2,U,A,T2),
                     type_gen(T,B,R2).
```

Example : identity function

```
var(x).
lambda(id,x,t,x).
fun(t_t,t,t).
context_typed(id,empty).
bind(ctx1,empty,x,t).
forall_intro(forall_id,id,t).
type_gen(id_type,t,t_t).
```

Example : identity function

```
z(zero).

forall_elim(id_nat,forall_id,t,T) :- typed(zero,T).
app(id_zero,id_nat,zero).

typed(id_zero,T)?
> typed(id_zero,nat).
```

Example : identity function

```
tru(_true).

forall_elim(id_bool,forall_id,t,T) :- typed(_true,T).
app(id_true,id_bool,_true).

typed(id_true,T)?
> typed(id_true,bool).
```

Main issue : generating existing types

Reminder : to avoid infinite relations, we must only consider types defined in the program.

The type of the identity function is $\forall t. t \to t$

When we apply identity to a nat, after forall elimination, it becomes $nat \to nat$

But this can only typecheck if the type $nat \to nat$ has been defined somewhere!

Further work

- Automate type checking for generic types and polymorphic functions
- Find a better solver
- Generate necessary types