A Relational Solver for Constraint-based Type Inference

Eridan Domoratskiy, Dmitry Boulytchev

St. Petersburg State University, Russia

Tue 10 Dec 2024

$\lambda^a\mathcal{M}^a$ Programming Language

$\lambda^a\mathcal{M}^a$ — educational programming language:

- Functional: every program is expression, has first-class functions
- Imperative: expressions evaluate in strictly defined order, function calls can produce side effects, data structures mutation allowed
- Native-compiled: programs must be compiled before running, target platform is Intel x86
- Modular: programs can be composed from several modules, that are compiled separately
- Untyped: there aren't statical and runtime type checks

$\lambda^a\mathcal{M}^a$: Simple Program

This program calculates the Ackermann function of $0 \leq m \leq 3$ and $0 \leq n \leq 8$ by definition:

Here we may notice:

- variable definition, using and assignment
- function definition and calls (including recursive)
- conditional and loop expressions
- infix operators (including ";" standing for sequential evaluation operator)

```
var x, m, n ;

fun ack (m, n) {
  if m == 0 then n + 1
  elif m > 0 && n == 0 then ack (m - 1, 1)
  else ack (m - 1, ack (m, n - 1))
  fi
}

x := read () ;
for m := 0, m <= 3, m := m + 1 do
  for n := 0, n <= 8, n := n + 1 do
    write (ack (m, n))
  od
od
```

$\lambda^a\mathcal{M}^a$: Complex Data Structures

The following program generates list of 1000 integers and sorts it using bubble sort:

Here we may notice:

- arrays and linked lists
- pattern matching
- subvalues accessing by index

Linked list is the special case of more general construction, that in $\lambda^a\mathcal{M}^a$ named S-expression

```
fun compare (x, y) { x - y }
fun bubbleSort (l) {
  fun inner (l) {
    case l of
      x : z@(y : tl) ->
       if compare (x, y) > 0
       then [true, y : inner (x : tl) [1]]
       else case inner (z) of [f, z] -> [f, x : z] esac
       fi
    | _ -> [false, l]
    esac
  }
  fun rec (l) {
    case inner (l) of
      [true , l] -> rec (l)
    | [false, l] -> l
    esac
  }
  rec (l)
}
fun generate (n) { if n then n : generate (n - 1) else {} fi }
bubbleSort (generate (1000))
```

$\lambda^a\mathcal{M}^a$: S-expressions and First-Class Functions

The next example is simple λ-calculus interpreter:

We may notice:

- module importing
- first-class functions
- strings
- S-expressions

S-expressions acts like tagged arrays and provides convenient ADT-style data manipulation (like in HASKELL, OCAML, etc)

```
import Data ;

fun emptyContext () { fun (v) {
  failure ("No variable %s in current scope!\n", v.string)
} }
fun extendContext (ctx, v, x) {
  fun (u) { if v === u then x else ctx (u) fi }
}

fun eval (ctx, expr) {
  case expr of
    Val (x) -> x
  | Var (v) -> ctx (v)
  | App (f, x) -> eval (ctx, f) (eval (ctx, x))
  | Lam (v, b) ->
      fun (x) { eval (extendContext (ctx, v, x), b) }
  esac
}
fun eval0 (expr) { eval (emptyContext (), expr) }

var test = App (Lam ("x", Var ("x")), Val ("Hello!")) ;
printf ("%s\n", eval0 (test))
```

$\lambda^a\mathcal{M}^a$: Functional Programming, State Monad

- Since we have in $\lambda^a\mathcal{M}^a$ first-class functions support, sometimes it's convenient to use FP idioms like monads
- For example, to implement backtracking in parser it is convenient to use the state monad
- So, there is an implementation of this monad in the $\lambda^a\mathcal{M}^a$ standard library[1]:
- Also, we may notice utilization of the user-defined infix operators feature of $\lambda^a\mathcal{M}^a$

```
import Fun ;

infix >>= before $ (m, k) {
  fun (state) {
    case m (state) of
      [state, x] -> k (x) (state)
    esac
  }
}

fun pure (x) {
  fun (state) { [state, x] }
}
```

[1] The code is slightly changed for a reader convenience

Defects in $\lambda^a\mathcal{M}^a$ Programs

```
fun size (xs) {
  case xs of
    {} -> 0
  | _ : xs -> 1 + size (xs)
  esac
}

-- ...

size ("123")
```

(a) Simple defect

```
fun nextStep (x) { x + 1 }

-- ...

var myComputation = pure (1) >>= nextStep ;

-- ...

var initialState = 0 ;
myComputation (initialState) [1]
```

(b) Difficult defect

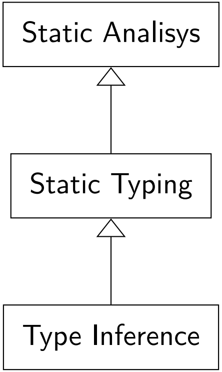
a) A string is passed instead of a list in "`size`":
  - 👍 Match failures are detected as soon as possible with detailed error messages
  - 👍 The cause of a crash is located immediately next to the crash location

b) A programmer forgot to wrap a result of "`nextStep`" in "`pure`":
  - 👎 Calling non-callable values crashes with the "Segmentation fault" error
  - 👎 The cause of a crash may be "across the code" from the crash location

Static Analysis and Static Typing

- **Static analysis** is a method to prevent some classes of program defects before running the program (ahead-of-time)
- A classical way to deal with defects like (b) is **static typing**
- In various modern programming languages, optional explicit type annotations were invented to allow static analysis through static typing (PYTHON, TYPESCRIPT, etc.)
  - 👍 Provides additional documentation
  - 👍 Simplifies static analyzers
  - 👎 Requires to change an existing code to use static analyzer on
  - 👎 Complicates language syntax
  - 👎 Distracts programmer

👍 Instead, we study static typing through full **type inference**, that allows to analyze programs without changing them



Constraint-based Type Inference

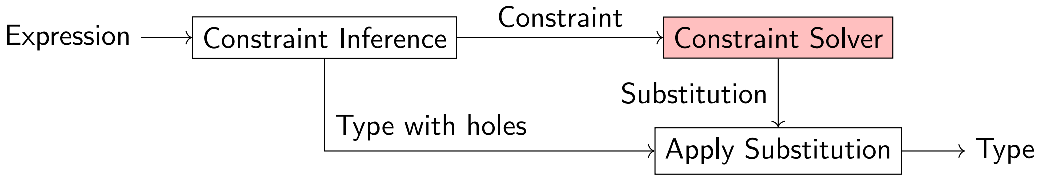
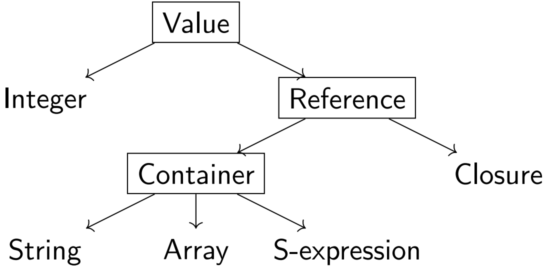


Figure: Constraint-based type inference

- Modern programming languages that support full type inference uses **constraint-based type inference**
- Currently, there are many frameworks that provides unified way to implement constraint-based type inference algorithm for given type system and constraint solver
- Thus, we need to develop the type system and the constraint solver

$\lambda^a\mathcal{M}^a$: Data Shapes Classification



(a) Shapes of data in $\lambda^a\mathcal{M}^a$

The shapes of data:

- Integer — integral value in bounds $[-2^{30}, 2^{30} - 1]$
- String — array of ASCII characters
- Array — array of arbitrary values
- S-expression — array with associated literal tag
- Closure — first-order function

Type System

$$\mathsf{T} ::= \mathbb{Z} \mid \mathbb{S} \mid [\mathsf{T}] \mid \mathcal{X}\ (\mathsf{T}, ..., \mathsf{T}) \sqcup ... \sqcup \mathcal{X}\ (\mathsf{T}, ..., \mathsf{T})$$
$$\mid \mathcal{X} \mid \forall \mathcal{X}, ..., \mathcal{X}.\ C \Rightarrow (\mathsf{T}, ..., \mathsf{T}) \to \mathsf{T} \mid \mu \mathcal{X}.\ \mathsf{T}$$

Legend:

- $\mathcal{X}$ — type variables and literal tags
- T — types
- C — constraints

Figure: Syntax of types

The forms of types:

- Integer, String
- Array, S-expression
- Type variable, Function
- Recursive type

- S-expression types are composition of product and sum types
- Function types are generic (by "$\mathcal{X}, ..., \mathcal{X}$") and possible specializations are constrained (by "C")
- We need to allow explicit recursive types to support recursive data structures, since there aren't any type declarations in source code

Type System: Examples

Obvious:

- `0` : $\mathbb{Z}$
- `123` : $\mathbb{Z}$
- `"123"` : $\mathbb{S}$
- `[]` : [T] for any `T`
- `[1, 2, 3]` : $[\mathbb{Z}]$
- `One (Two, Three)` : $One(Two, Three)$
- `One (Two, Three)` : $One(Two \sqcup A, Three \sqcup B \sqcup C) \sqcup D$, since S-expression types may include more that one constructor

Type System: Examples

Obvious:

- `0` : $\mathbb{Z}$
- `123` : $\mathbb{Z}$
- `"123"` : $\mathbb{S}$
- `[]` : $[\mathsf{T}]$ for any T
- `[1, 2, 3]` : $[\mathbb{Z}]$
- `One (Two, Three)` : $One(Two, Three)$
- `One (Two, Three)` : $One(Two \sqcup A, Three \sqcup B \sqcup C) \sqcup D$,
  since S-expression types may include more that one constructor

But:

- `0` : T for any T, since "`0`" is used as null value in many cases
- `{}` : T for any T, since "`{}`" is a syntactic sugar for "`0`"
- `{1, 2, 3}` : $cons(\mathbb{Z}, cons(\mathbb{Z}, cons(\mathbb{Z}, \mathsf{T})))$ for any T, as in previous case
- `{1, 2, 3}` : $\mu\alpha.\ cons(\mathbb{Z}, \alpha)$, since this is a general type of all lists of integers

Constraint System

$$C ::= \top \mid C \wedge C \mid Ind(\mathsf{T}, \mathsf{T}) \mid Call(\mathsf{T}, \mathsf{T}, ..., \mathsf{T}, \mathsf{T}) \mid Sexp_{\mathcal{X}}(\mathsf{T}, \mathsf{T}, ..., \mathsf{T}) \mid ...$$

Figure: Syntax of constraints

The forms of constraints:

- $\top, C_1 \wedge C_2$ — conjunction of constraints
- $Ind(\mathsf{T}, \mathsf{S})$ — values of type T contains elements of type S; i. e. given `x` : T and `idx` : $\mathbb{Z}$, `x [idx]` : S
- $Call(\mathsf{T}, \mathsf{S}_1, ..., \mathsf{S}_n, \mathsf{S})$ — values of type T callable with arguments of types S_i and result have type S; i. e. given `f` : T and $\mathtt{x}_i : \mathsf{S}_i$, $\mathtt{f}\ (\mathtt{x}_1,\ ...,\ \mathtt{x}_n) : \mathsf{S}$
- $Sexp_{\mathcal{X}}(\mathsf{T}, \mathsf{S}_1, ..., \mathsf{S}_n)$ — type T is S-expression type and one of it's constructors have form $\mathcal{X}\ (\mathsf{S}_1, ..., \mathsf{S}_n)$; i. e. given $\mathtt{x}_i : \mathsf{S}_i$, $\mathtt{X}\ (\mathtt{x}_1,\ ...,\ \mathtt{x}_n) : \mathsf{T}$
- ... — other constraints that we don't mention

Constraint System: Examples

- `fun (x) { x }` $: \forall \alpha.\ \top \Rightarrow (\alpha) \to \alpha$
- `fun (xs, i, x) { xs [i] := x ; xs }` $: \forall \alpha, \beta.\ Ind(\alpha, \beta) \Rightarrow (\alpha, \mathbb{Z}, \beta) \to \alpha$
- `fun (f, x) { f (x) }` $: \forall \alpha, \beta, \gamma.\ Call(\alpha, \beta, \gamma) \Rightarrow (\alpha, \beta) \to \gamma$
- `fun (x) { Some (x) }` $: \forall \alpha, \beta.\ Sexp_{Some}(\alpha, \beta) \Rightarrow (\beta) \to \alpha$

Constraint Solver

- We define binary relation "$C_1 \Vdash C_2$" that means "C_1 implies C_2", in terms of natural deduction
- The job of constraint solver is to suggest for the given "C" a type substitution "σ" so that "$\top \Vdash C\sigma$" satisfied, or state that it doesn't exist

! Any constraint "C" has form "$C_1 \wedge C_2 \wedge ... \wedge C_n$", where C_i is an **atomic constraint** (constraint without conjunctions)

Constraint Solver

- We define binary relation "$C_1 \Vdash C_2$" that means "C_1 implies C_2", in terms of natural deduction
- The job of constraint solver is to suggest for the given "C" a type substitution "σ" so that "$\top \Vdash C\sigma$" satisfied, or state that it doesn't exist
- ! Any constraint "C" has form "$C_1 \wedge C_2 \wedge \ldots \wedge C_n$", where C_i is an **atomic constraint** (constraint without conjunctions)
- If we consider atomic constraints as predicates over types, any constraint is just CNF in first-order logic

Constraint Solver

- We define binary relation "$C_1 \Vdash C_2$" that means "C_1 implies C_2", in terms of natural deduction
- The job of constraint solver is to suggest for the given "C" a type substitution "σ" so that "$\top \Vdash C\sigma$" satisfied, or state that it doesn't exist
- ! Any constraint "C" has form "$C_1 \wedge C_2 \wedge \ldots \wedge C_n$", where C_i is an **atomic constraint** (constraint without conjunctions)
- If we consider atomic constraints as predicates over types, any constraint is just CNF in first-order logic
- So, we need a tool that could solve that CNF or state that there aren't solutions

Constraint Solver

- We define binary relation "$C_1 \Vdash C_2$" that means "C_1 implies C_2", in terms of natural deduction
- The job of constraint solver is to suggest for the given "C" a type substitution "σ" so that "$\top \Vdash C\sigma$" satisfied, or state that it doesn't exist
- ! Any constraint "C" has form "$C_1 \wedge C_2 \wedge \ldots \wedge C_n$", where C_i is an **atomic constraint** (constraint without conjunctions)
- If we consider atomic constraints as predicates over types, any constraint is just CNF in first-order logic
- So, we need a tool that could solve that CNF or state that there aren't solutions
- Here we meet **relational programming**

Relational Programming

Relational programming (especially, miniKanren):

- a special case of logic programming without side effects
- minimalistic embedded programming language (implementing as DSL library for existing programming languages)
- presented in The Reasoned Schemer [1]
- implemented for many popular programming languages: Scheme, OCaml, Kotlin, etc.
- have proven search completeness [2]

miniKanren: Terms

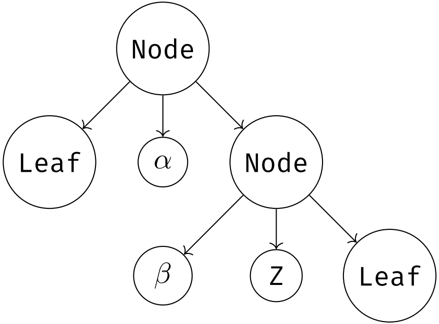
Term in logic programming is an expression of the following syntax:

$$t ::= \mathcal{X} \mid \mathcal{C}(t, ..., t),$$

where $\mathcal{X}$ — variable, $\mathcal{C}$ — tag

Examples of term:

- α
- `None`
- `Some`(α)
- `Z` — Peano number "0"
- `S(S(S(Z)))` — Peano number "3"
- `Node`$($`Leaf`$, \alpha,$ `Node`$(\beta,$ `Z`, `Leaf`$))$



miniKanren: Goals

Goal in miniKanren is a special case of logical expression:

$$g ::= \top \mid \bot \mid t = t \mid \mathcal{P}(t, ..., t) \mid g \wedge g \mid g \vee g \mid \exists \mathcal{X}.\ g,$$

where $\mathcal{P}$ — predicate, so we haven't implication, negation and forall quantifier

Examples of goal:

- $\alpha = \beta$
- $\alpha = \mathsf{A} \vee \alpha = \mathsf{B}$
- $P(\alpha) \wedge Q(\alpha, \beta)$
- $\alpha = \texttt{None} \vee \exists \beta.\ \alpha = \texttt{Some}(\beta)$
- $\exists \beta.\ \alpha = \mathsf{S}(\beta)$ — predicate "$\alpha \geq 1$" on Peano numbers

miniKanren: Definitions

Definition in miniKanren is like definition in logic programming but with a goal on the right-hand side:

$$\mathcal{P}(\mathcal{X}, ..., \mathcal{X}) \equiv g,$$

and the goal mustn't has free variables except listed on the left-hand side

Examples of definition:

- $P \equiv \top$
- $Q(\alpha, \beta) \equiv \alpha = \beta \vee \exists \gamma.\ \beta = \mathsf{S}(\gamma) \wedge Q(\alpha, \gamma)$
- $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$ — relation "$\alpha + \beta = \gamma$" on Peano numbers

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:

$$add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

1. $\mathsf{S}(\mathsf{Z}) = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \xi \vee \exists \alpha', \gamma'.\ \mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

1. $\mathsf{S}(\mathsf{Z}) = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \xi \vee \exists \alpha', \gamma'.\ \mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
2. $\mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

1. $\mathsf{S}(\mathsf{Z}) = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \xi \vee \exists \alpha', \gamma'.\ \mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
2. $\mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
3. $add^o(\mathsf{Z}, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma'); \xi \mapsto \mathsf{S}(\gamma')$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

1. $\mathsf{S}(\mathsf{Z}) = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \xi \vee \exists \alpha', \gamma'.\ \mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
2. $\mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
3. $add^o(\mathsf{Z}, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma'); \xi \mapsto \mathsf{S}(\gamma')$
4. $\mathsf{Z} = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \gamma' \vee \exists \alpha', \gamma''.\ \mathsf{Z} = \mathsf{S}(\alpha') \wedge \gamma' = \mathsf{S}(\gamma'') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma''); \xi \mapsto \mathsf{S}(\gamma')$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

1. $\mathsf{S}(\mathsf{Z}) = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \xi \vee \exists \alpha', \gamma'.\ \mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
2. $\mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
3. $add^o(\mathsf{Z}, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma'); \xi \mapsto \mathsf{S}(\gamma')$
4. $\mathsf{Z} = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \gamma' \vee \exists \alpha', \gamma''.\ \mathsf{Z} = \mathsf{S}(\alpha') \wedge \gamma' = \mathsf{S}(\gamma'') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma''); \xi \mapsto \mathsf{S}(\gamma')$
5. $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \gamma'; \xi \mapsto \mathsf{S}(\gamma')$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \xi)$

1. $\mathsf{S}(\mathsf{Z}) = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \xi \vee \exists \alpha', \gamma'.\ \mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
2. $\mathsf{S}(\mathsf{Z}) = \mathsf{S}(\alpha') \wedge \xi = \mathsf{S}(\gamma') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma')$
3. $add^o(\mathsf{Z}, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma'); \xi \mapsto \mathsf{S}(\gamma')$
4. $\mathsf{Z} = \mathsf{Z} \wedge \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \gamma' \vee \exists \alpha', \gamma''.\ \mathsf{Z} = \mathsf{S}(\alpha') \wedge \gamma' = \mathsf{S}(\gamma'') \wedge add^o(\alpha', \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \gamma''); \xi \mapsto \mathsf{S}(\gamma')$
5. $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \gamma'; \xi \mapsto \mathsf{S}(\gamma')$
6. $\top; \xi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))))$

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \land \beta = \gamma \lor \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \land \gamma = \mathsf{S}(\gamma') \land add^o(\alpha', \beta, \gamma')$

Query: $add^o(\xi, \psi, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))))$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\xi, \psi, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))))$

- $\xi \mapsto \mathsf{Z}, \psi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))))$

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \land \beta = \gamma \lor \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \land \gamma = \mathsf{S}(\gamma') \land add^o(\alpha', \beta, \gamma')$

Query: $add^o(\xi, \psi, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))))$

- $\xi \mapsto \mathsf{Z}, \psi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))))$
- $\xi \mapsto \mathsf{S}(\mathsf{Z}), \psi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))$

miniKanren: Program

- miniKanren program is a list of relation definitions and a query (just a goal)
- Unlike logic programming languages like Prolog, definitions must be distinct by name
- Program evaluation results in all possible substitutions of the query's free variables, that satisfy the query
- Consider the program:
  $add^o(\alpha, \beta, \gamma) \equiv \alpha = \mathsf{Z} \wedge \beta = \gamma \vee \exists \alpha', \gamma'.\ \alpha = \mathsf{S}(\alpha') \wedge \gamma = \mathsf{S}(\gamma') \wedge add^o(\alpha', \beta, \gamma')$

Query: $add^o(\xi, \psi, \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))))$

- $\xi \mapsto \mathsf{Z}, \psi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))))$
- $\xi \mapsto \mathsf{S}(\mathsf{Z}), \psi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))$
- $\xi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{Z})), \psi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{Z}))$
- $\xi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \psi \mapsto \mathsf{S}(\mathsf{Z})$
- $\xi \mapsto \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))), \psi \mapsto \mathsf{Z}$

Verifier-to-Solver Approach

Given problem:

- We can implement relational program "$P(x)$" that verifies problem solution "x" — verifier
- Running relational verifier "$P(\xi)$" on free variable "ξ" gives solution "$\xi \mapsto x$" — solver
- Moreover, it gives the all possible solutions due to search completeness

Verifier-to-Solver Approach

Given problem:

- We can implement relational program "$P(x)$" that verifies problem solution "x" — verifier
- Running relational verifier "$P(\xi)$" on free variable "ξ" gives solution "$\xi \mapsto x$" — solver
- Moreover, it gives the all possible solutions due to search completeness
- Unfortunately, relational program evaluation may hangs in practice due to exponential search complexity
- ...or simply, due to specific problem undecidability

Verifier-to-Solver: Application

- We may implement relational verifier for "$\Vdash$"
- "$\Vdash$" implemented directly in miniKanren works slowly and don't answer at all when proper substitution isn't exists
- Also, vanilla miniKanren implementations doesn't allow to deal with recursive terms, that are needed to deal with recursive types
- Wildcard logic variables [3]:
  - Most of miniKanren implementations supports inequality as a primitive
  - Wildcard variables allows to say "$\forall\psi.\ \xi \neq \mathtt{Cons}(\psi)$" instead of "$\exists\psi.\ \xi \neq \mathtt{Cons}(\psi)$"
- Non-relational optimizations to specialize relational implementation in the constraint solving problem:
  - Term shape check — non-relational primitives that give an ability to introspect current evaluation state and direct evaluator manually
  - Occurs hooks — an ability to hook an occurs check to permit unnatural recursive equations solving while unification

Term Shape Check

```
let rec contains x xs = ocanren {
  fresh x', xs' in xs == x'::xs' &
    { x == x'
    | x =/= x' & contains x xs'
    }
}
```

```
let rec contains x xs = ocanren
  { is_var xs &
    { fresh xs' in xs == x::xs' }
  | is_not_var xs &
    fresh x', xs' in xs == x'::xs' &
      { x == x'
      | x =/= x' & contains x xs'
      }
  }
```

- ? How to implement `contains`^{o} for sets encoded as lists?
- 👎 Trivial implementation will generate a lot of syntactically different, but semantically identical lists
- 👍 With the shape checking we are able to enforce a single solution in the case when the tail of list is free

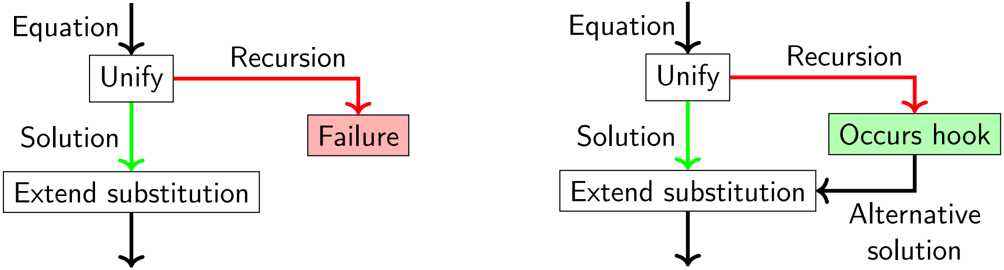
Search Space of Function Types

- ? How to solve a constraint of the form "$Call(\mathsf{T}, \mathsf{S}_1, \ldots, \mathsf{S}_n, \mathsf{S})$" when "T" is a free logic variable?
- The only form that "T" could be is: $\forall \mathcal{X}_1, \ldots, \mathcal{X}_m.\ C \Rightarrow (\mathsf{T}_1, \ldots, \mathsf{T}_n) \to \mathsf{T}'$
- In the implementation: `TArrow (fxs, fc, fts, ft)`
- A straightforward implementation will generate the variety of function types with all possible values of "`fxs`" and "`fc`"
- ! **Assumption:** all needed function types come from relational query, i. e. we don't need to generate them
- Just enforce the simplest possible type in the case when "T" is free: $\forall.\ \top \Rightarrow (\mathsf{S}_1, \ldots, \mathsf{S}_n) \to \mathsf{S}$
- ! This approach may cut off some solutions when "$Call$" is being solved too early

Solving Ordering

- Solving constraints left-to-right works poorly
- Let's support all planned to solve constraints and pick them one-by-one in a *good* order
- As a *good* order, we use picking a minimum by relational heuristic comparator, that inspects shapes (using "`is_var`") of constraint arguments
- For example, constraints of form "$Call(\mathsf{T}, \mathsf{S}_1, ..., \mathsf{S}_n, \mathsf{S})$" with logic variable in place of "T" are being picked last
- In addition to performance gain, we have fixed a problem with early solution that was mentioned before

Occurs Hooks



(a) Without occurs hooks (b) With occurs hooks

- Occurs hook is a callback that called when a recursive equation occurred in unification: $\xi = Term(\xi)$
- A hook returns an alternative right side of equation, so the new equation will be: $\xi = hook(\xi, Term(\xi))$

Evaluation

- We used the $\lambda^a\mathcal{M}^a$ compiler tests
- 👍 The majority of tests were successfully typechecked
- 👎 S-expression types are slowing down performance exponentionally of the number of $Sexp$ constraints
- 👍 Other constraints aren't so slow
- ! Occurs hooks gives an ability to deal with recursive types, but results not as good as possible, e.g.:
  - Good type: $\mu\alpha.\ \texttt{Nil} \sqcup \texttt{Cons}(\mathbb{Z}, \alpha)$
  - Produced type: $\texttt{Nil} \sqcup \texttt{Cons}(\mathbb{Z}, \mu\alpha.\ \texttt{Nil} \sqcup \texttt{Cons}(\mathbb{Z}, \alpha))$

Evaluation

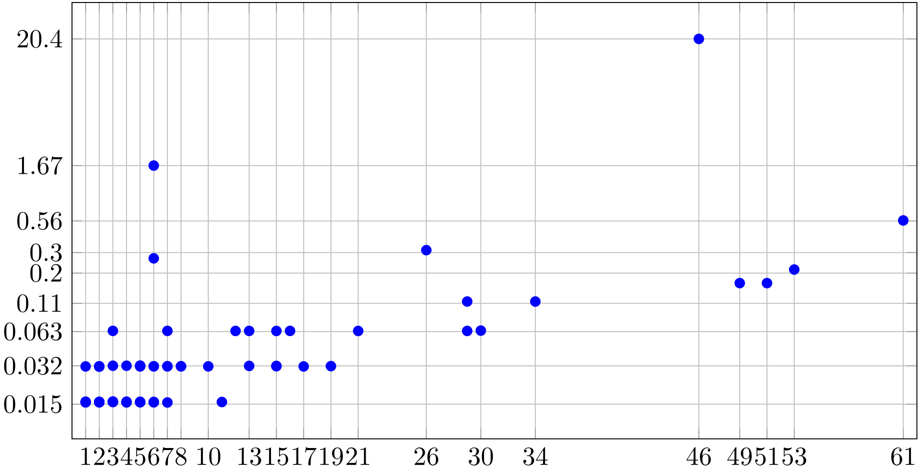
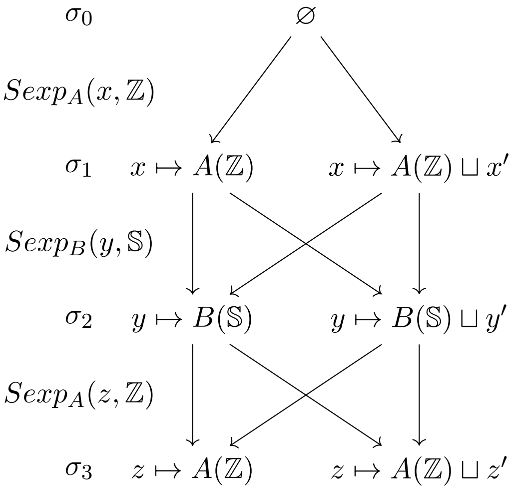


Figure: Elapsed time by the number of constraints

Evaluation: S-expression Types

- Example:
  $Sexp_A(x, \mathbb{Z}) \wedge Sexp_B(y, \mathbb{S}) \wedge Sexp_A(z, \mathbb{Z})$
- We have $m = 3$ different S-expression types x, y, z and $n = 2$ different constructors $A(\mathbb{Z}), B(\mathbb{S})$
- Number of branches is $\mathcal{O}(n^m)$
- As a result, we have about 8 branches only from this constraints
- It explains the high time consumption on the previous slide



Evaluation: S-expression Types

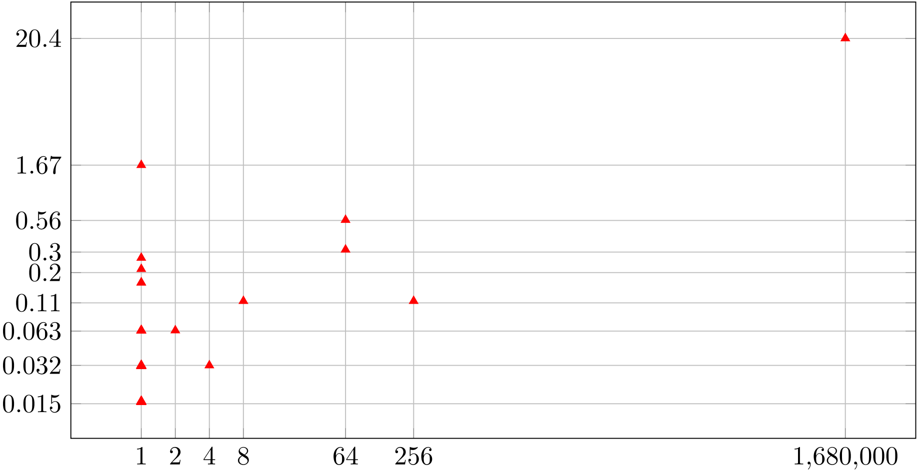


Figure: Elapsed time by the number of $Sexp$ branches

Results

- ✔ Relational programming is applicable to constraint-based type inference
- ✔ To specialize relational solvers we may inspect current state (`is_var`/`is_not_var`)
- ✔ Recursive terms may be partially emulated using non-relational hooks over "occurs check"
- This results are preliminary that need more research

More technical details are available in [4]

Results

- ✔ Relational programming is applicable to constraint-based type inference
- ✔ To specialize relational solvers we may inspect current state (`is_var`/`is_not_var`)
- ✔ Recursive terms may be partially emulated using non-relational hooks over "occurs check"
- ⋯ This results are preliminary that need more research

Questions?

More technical details are available in [4]

References

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov.
The Reasoned Schemer.
The MIT Press. MIT Press, 2005.

Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev.
Certified semantics for relational programming.
In *Asian Symposium on Programming Languages and Systems*, pages 167–185. Springer, 2020.

Dmitry Kosarev, Daniil Berezun, and Peter Lozov.
Wildcard logic variables.
In *miniKanren and Relational Programming Workshop*, 2022.

Eridan Domoratskiy and Dmitry Boulytchev.
A relational solver for constraint-based type inference.
arXiv preprint arXiv:2408.17138, 2024.