Semi-Automated Direction-Driven Functional Conversion

Kate Verbitskaia, Igor Engel, Daniil Berezun

JetBrains Research, Programming Languages and Tools Lab

miniKanren workshop @ ICFP 2023

08.09.2023

Relational Programming

One relation to solve many problems

Nondeterminism

Completeness of search

Relational Conversion: Easy

Given a function

```
let rec add x y =
  match x with
  | O  →  y
  | S x1  → S (add x1 y)
```

generate miniKanren relation

```
let rec addᵒ x y z = conde [
  (x ≡ O  ∧  y ≡ z);
  (fresh (x1 z1)
    (x ≡ S x1  ∧
     addᵒ x1 y z1  ∧
     z ≡ S z1)) ]
```

Principal Directions of miniKanren Relations

Every argument of a relation can be either `in` or `out`

For addition relation `add`^{o} `x y z` there are 8 directions:

- *Forward* direction: `add`^{o} `in in out` — addition
- *Backward* direction: `add`^{o} `out out in` — decomposition
- *Predicate*: `add`^{o} `in in in`
- *Generator*: `add`^{o} `out out out`
- `add`^{o} `in out in` — subtraction
- `add`^{o} `out in in` — subtraction
- `add`^{o} `out in out`
- `add`^{o} `in out out`

Each Direction is a Function

Each Direction is a Function (kind of)

Straightforward functions:

- *Forward* direction: `add`^{o} `in in out` — addition
- `add`^{o} `in out in` — subtraction
- `add`^{o} `out in in` — subtraction
- *Predicate*: `add`^{o} `in in in`

Relations:

- *Backward* direction: `add`^{o} `out out in` — decomposition
- *Generator*: `add`^{o} `out out out`
- `add`^{o} `out in out`
- `add`^{o} `in out out`

These relations are functions which return multiple answers (list monad)

miniKanren Comes with an Overhead

Unifications

Occurs-check

Scheduling complexity

Functional Conversion

Given a relation and a principal direction, construct a functional program which generates the same answers as miniKanren would

Preserve completeness of the search

Both inputs and outputs are expected to be ground

Example: Addition in Forward Direction

```
let rec add^o x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x1 z1)
    (x ≡ S x1 ∧
     add^o x1 y z1 ∧
     z ≡ S z1)) ]
```

```
addIIO :: Nat → Nat → Nat
addIIO x y =
  case x of
    O → y
    S x1 → S (addIIO x1 y)
```

Addition in the Backward Direction: Nondeterminism

```
let rec addᵒ x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁)) ]
```

```
addOOI :: Nat → Stream (Nat, Nat)
addOOI z =
  return (O, z) `mplus`
  case z of
    O → Empty
    S z₁ → do
      (x₁, y) ← addOOI z₁
      return (S x₁, y)
```

Free Variables in Answers: Generators

```
let rec add^o x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁ ∧ z ≡ S z₁ ∧ add^o x₁ y z₁) ) ]
```

```
addIOO :: Nat → Stream (Nat, Nat)
addIOO x = case x of
           O → do
             z ← genNat
             return (z, z)
           S x₁ → do
             (y, z₁) ← addIOO x₁
             return (y, S z₁)

genNat :: Stream Nat
genNat = Mature O (S <$> genNat)
```

Predicates

```
let rec add^o x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     add^o x₁ y z₁ ∧
     z ≡ S z₁)) ]
```

```
addIII :: Nat → Nat → Nat → Stream ()
addIII x y z =
  case x of
    O | y == z → return ()
      | otherwise → Empty
    S x₁ →
      case z of
        O → Empty
        S z₁ → addIII x₁ y z₁
```

Conversion Scheme

- Normalization
- Mode analysis
- Functional conversion

Normalization: Flat Term

Flat terms: a var or a constructor which takes *distinct* vars as arguments:

$$\mathcal{FT}_V = V \cup \{\mathcal{C}_i\,(x_1, \ldots, x_{k_i}) \mid x_j \in V, x_j - \mathit{distinct}\}$$

Examples:

$$C\,(x_1, x_2) \equiv C\,(C\,(y_1, y_2)\,, y_3) \iff x_1 \equiv C\,(y_1, y_2) \wedge x_2 \equiv y_3$$

$$C\,(C\,(x_1, x_2)\,, x_3) \equiv C\,(C\,(y_1, y_2)\,, y_3) \iff x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge x_3 \equiv y_3$$

$$x \equiv C\,(y, y) \iff x \equiv C\,(y_1, y_2) \wedge y_1 \equiv y_2$$

Normalization: Goal

$$
\begin{array}{lcll}
\mathcal{K}_V^N & = & \bigvee (c_1, \ldots, c_n), c_i \in \mathrm{Conj}_V & \text{normal form} \\
\mathrm{Conj}_V & = & \bigwedge (g_1, \ldots, g_n), g_i \in \mathrm{Base}_V & \text{normal conjunction} \\
\mathrm{Base}_V & = & V \equiv \mathcal{FT}_V & \text{flat unification} \\
 & | & R_i (x_1, \ldots, x_{k_i}), x_j \in V, x_j - \mathit{distinct} & \text{flat call}
\end{array}
$$

Mode of a Variable

Mode of a variable: mapping between its instantiations

Ground term contains no variables
Free variable: fresh variable, no info about its instantiation
Once we know that a variable is *ground*, it stays *ground* in subsequent conjuncts

Mode *in*: *ground* $\rightarrow$ *ground*
Mode *out*: *free* $\rightarrow$ *ground*

Mercury uses more complicated modes

Modded Goal

Assign mode to every variable, make sure they are consistent

Modded Unification

- Assignments: $x^{\text{out}} \equiv \mathcal{T}^{\text{in}}$ and $x^{\text{in}} \equiv y^{\text{out}}$
- Guards: $x^{\text{in}} \equiv \mathcal{T}^{\text{in}}$
- Match: $x^{\text{in}} \equiv \mathcal{T}$ ($\mathcal{T}$ contains both *in* and *out* variables)
- Generators: $x^{\text{out}} \equiv \mathcal{T}$

Mode Inference: Initialization

- For all input variables: *ground* $\rightarrow$?
- For all other variables: *free* $\rightarrow$?

$$
\begin{array}{l}
\textbf{let rec } \mathrm{add}^{o}\ \mathrm{x}^{g \rightarrow g}\ \mathrm{y}^{g \rightarrow g}\ \mathrm{z}^{f \rightarrow g} = \textbf{conde} \\
\quad (\mathrm{x}^{g \rightarrow g} \equiv \mathtt{0} \wedge \mathrm{y}^{g \rightarrow g} \equiv \mathrm{z}^{f \rightarrow g}); \\
\quad (\mathrm{x}^{g \rightarrow g} \equiv \mathtt{S}\ \mathrm{x}_1^{f \rightarrow ?} \wedge \\
\quad\ \mathrm{add}^{o}\ \mathrm{x}_1^{f \rightarrow ?}\ \mathrm{y}^{g \rightarrow g}\ \mathrm{z}_1^{f \rightarrow ?} \wedge \\
\quad\ \mathrm{z}^{f \rightarrow g} \equiv \mathtt{S}\ \mathrm{z}_1^{f \rightarrow ?})
\end{array}
$$

Mode Inference: Disjunction

Run inference on each disjunct independently

$$\mathrm{x}^{g \to g} \equiv \mathrm{O} \ \wedge \ \mathrm{y}^{g \to g} \equiv \mathrm{z}^{f \to g}$$

$$\begin{array}{l} \mathrm{x}^{g \to g} \equiv \mathrm{S}\ \mathrm{x}_1^{f \to ?} \ \wedge \\ \mathrm{add}^{o}\ \mathrm{x}_1^{f \to ?}\ \mathrm{y}^{g \to g}\ \mathrm{z}_1^{f \to ?} \ \wedge \\ \mathrm{z}^{f \to g} \equiv \mathrm{S}\ \mathrm{z}_1^{f \to ?} \end{array}$$

Mode Inference: Unification

Propagate the groundness information according to the 4 types of modded unifications

---

$$\mathrm{x}^{g \to g} \equiv \mathrm{S}\ \mathrm{x}_1^{f \to ?} \Rightarrow \mathrm{x}^{g \to g} \equiv \mathrm{S}\ \mathrm{x}_1^{f \to g}$$

$$\mathrm{z}^{f \to g} \equiv \mathrm{S}\ \mathrm{z}_1^{f \to ?} \Rightarrow \mathrm{z}^{f \to g} \equiv \mathrm{S}\ \mathrm{z}_1^{f \to g}$$

Mode Inference: Conjunction

Pick a conjunct according to the priority, propagate groundness

- Guards
- Assignments
- Matches
- Calls with at least one ground argument
- Generators

Mode Inference: Conjunction

$$
\begin{array}{l}
\mathrm{x}^{g \to g} \equiv \mathrm{S}\ \mathrm{x}_1^{f \to ?} \ \wedge \\
\mathrm{add}^{o}\ \mathrm{x}_1^{f \to ?}\ \mathrm{y}^{g \to g}\ \mathrm{z}_1^{f \to ?} \ \wedge \\
\mathrm{z}^{g \to g} \equiv \mathrm{S}\ \mathrm{z}_1^{f \to ?}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{x}^{g \to g} \equiv \mathrm{S}\ \mathrm{x}_1^{f \to g} \ \wedge \\
\mathrm{add}^{o}\ \mathrm{x}_1^{f \to g}\ \mathrm{y}^{g \to g}\ \mathrm{z}_1^{f \to ?} \ \wedge \\
\mathrm{z}^{g \to g} \equiv \mathrm{S}\ \mathrm{z}_1^{f \to ?}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{x}^{g \to g} \equiv \mathrm{S}\ \mathrm{x}_1^{f \to g} \ \wedge \\
\mathrm{add}^{o}\ \mathrm{x}_1^{f \to g}\ \mathrm{y}^{g \to g}\ \mathrm{z}_1^{f \to g} \ \wedge \\
\mathrm{z}^{g \to g} \equiv \mathrm{S}\ \mathrm{z}_1^{f \to g}
\end{array}
$$

Order in Conjunctions

```
let rec mult^o x y z = conde [
  ...
  (fresh (x1 r')
    (x ≡ S x1) ∧
    (add^o y r' z) ∧
    (mult^o x1 y r')
  )]
```

Order in Conjunctions: Slow Version

```
multIIO₁ :: Nat → Nat → Stream Nat
...
multIIO₁ (S x₁) y = do
  (r₁, r) ← addX y
  multIII x₁ y r₁
  return r

multIII :: Nat → Nat → Nat → Stream ()
...
multIII (S x₁) y z = do
  z₁ ← multIIO₁ x₁ y
  addIII y z₁ z
multIII _ _ _ = Empty
```

Premature grounding of z_1 leads to generate-and-test behavior

Order in Conjunctions: Faster Version

```
multIIO :: Nat → Nat → Stream Nat
...
multIIO (S x1) y = do
  r' ← multIIO x1 y
  addXY y r'
```

Functional Conversion: Intermediate Language

$$
\begin{array}{lcll}
\mathcal{F}_V & = & \mathsf{Return}\,[\mathcal{T}_V] & \text{return a tuple of terms} \\
& | & \mathsf{Match}_V\,(\mathcal{T}_V, \mathcal{F}_V) & \text{match a variable against a pattern} \\
& | & \mathsf{Bind}\,[([V], \mathcal{F}_V)] & \text{monadic bind on streams} \\
& | & \mathsf{Sum}\,[\mathcal{F}_V] & \text{concatenation of streams} \\
& | & \mathsf{Guard}\,(V, V) & \text{equality check} \\
& | & \mathsf{Gen}_G & \text{generator} \\
& | & R_i([V], [G]) & \text{function call}
\end{array}
$$

Functional Conversion into Intermediate Language

- Disjunction $\rightarrow$ Sum $[\mathcal{F}_V]$
- Conjunction $\rightarrow$ Bind $[([V], \mathcal{F}_V)]$
- Relation call $\rightarrow R_i([V], [G])$
- Unification $\rightarrow$ Return $[\mathcal{T}_V]$ or Match$_V$ $(\mathcal{T}_V, \mathcal{F}_V)$ or Guard (V, V) or Gen$_G$

Functional Conversion into Haskell

- TemplateHaskell to generate code
- Stream monad
- do-notation

Functional Conversion into OCaml

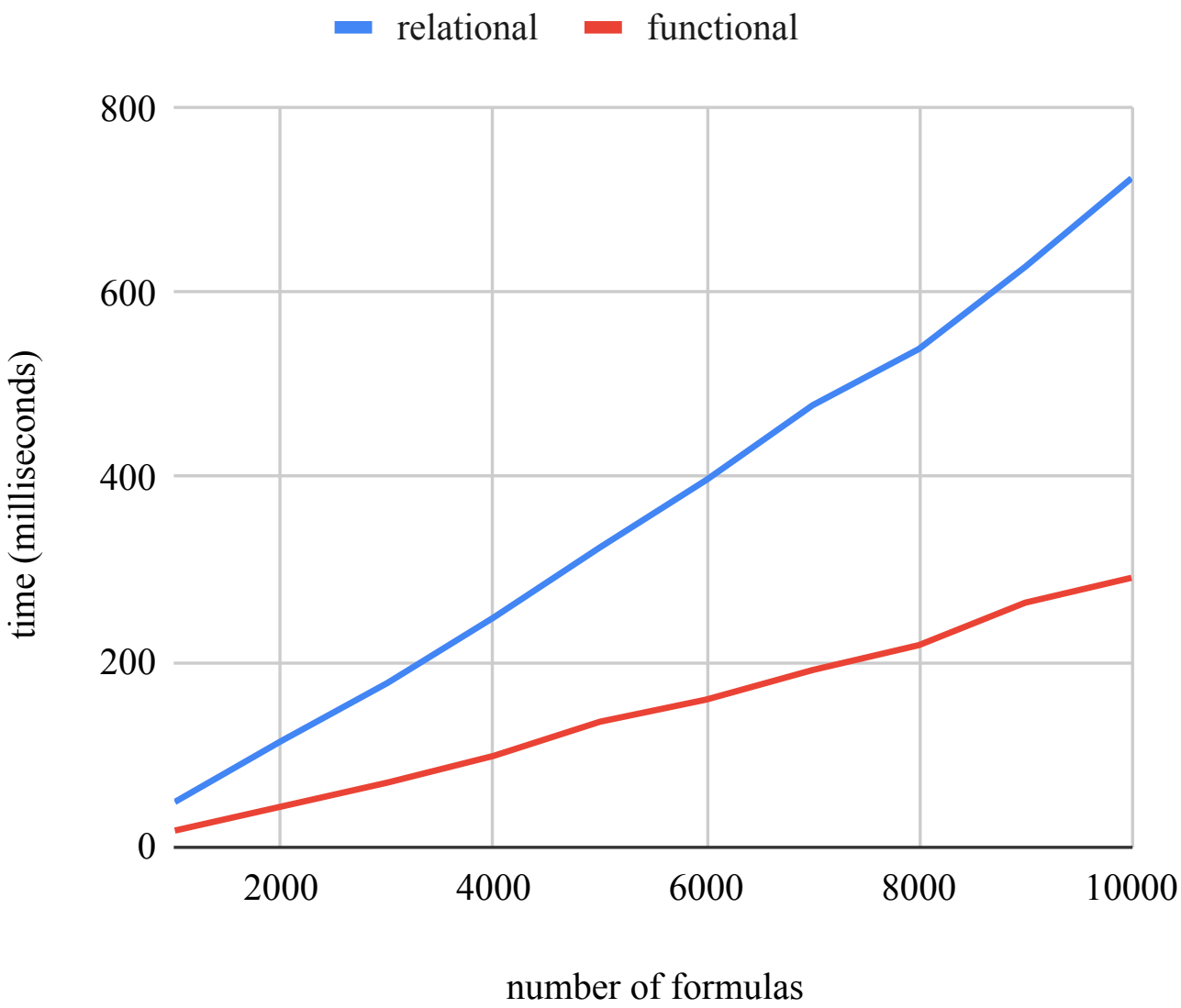
- Hand-crafted (not so) pretty-printer
- Stream monad
- let*
- Taking extra care to employ laziness

Evaluation

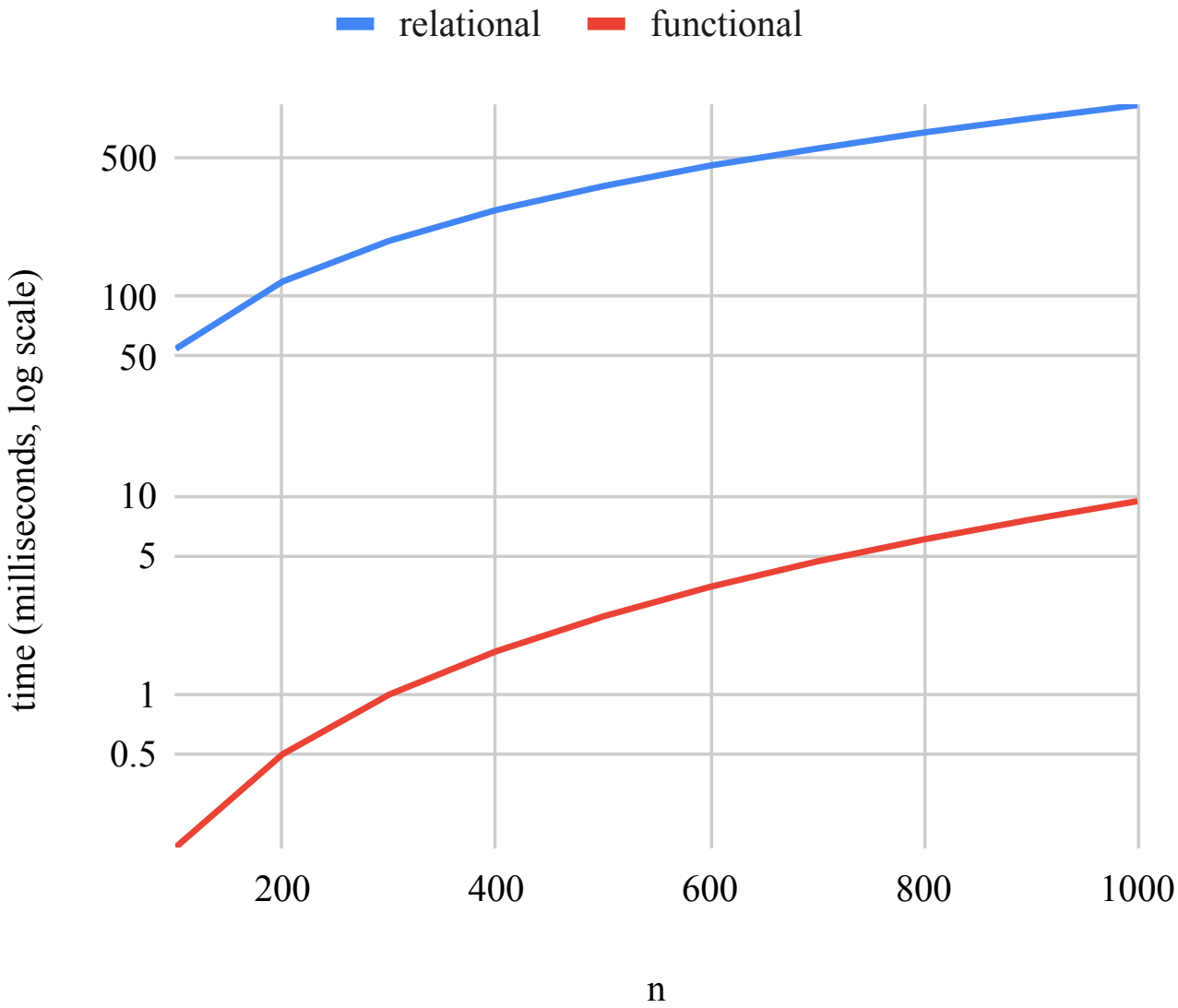
We converted relational interpreters and measured execution time

- Logic formulas generation
  - Inverse computation of an evaluator of logic formulas
  - Generating formulas which evaluate to **true**
- Multiplication relation
  - Forward direction: multiplication
  - Backward direction: division
  - Generation

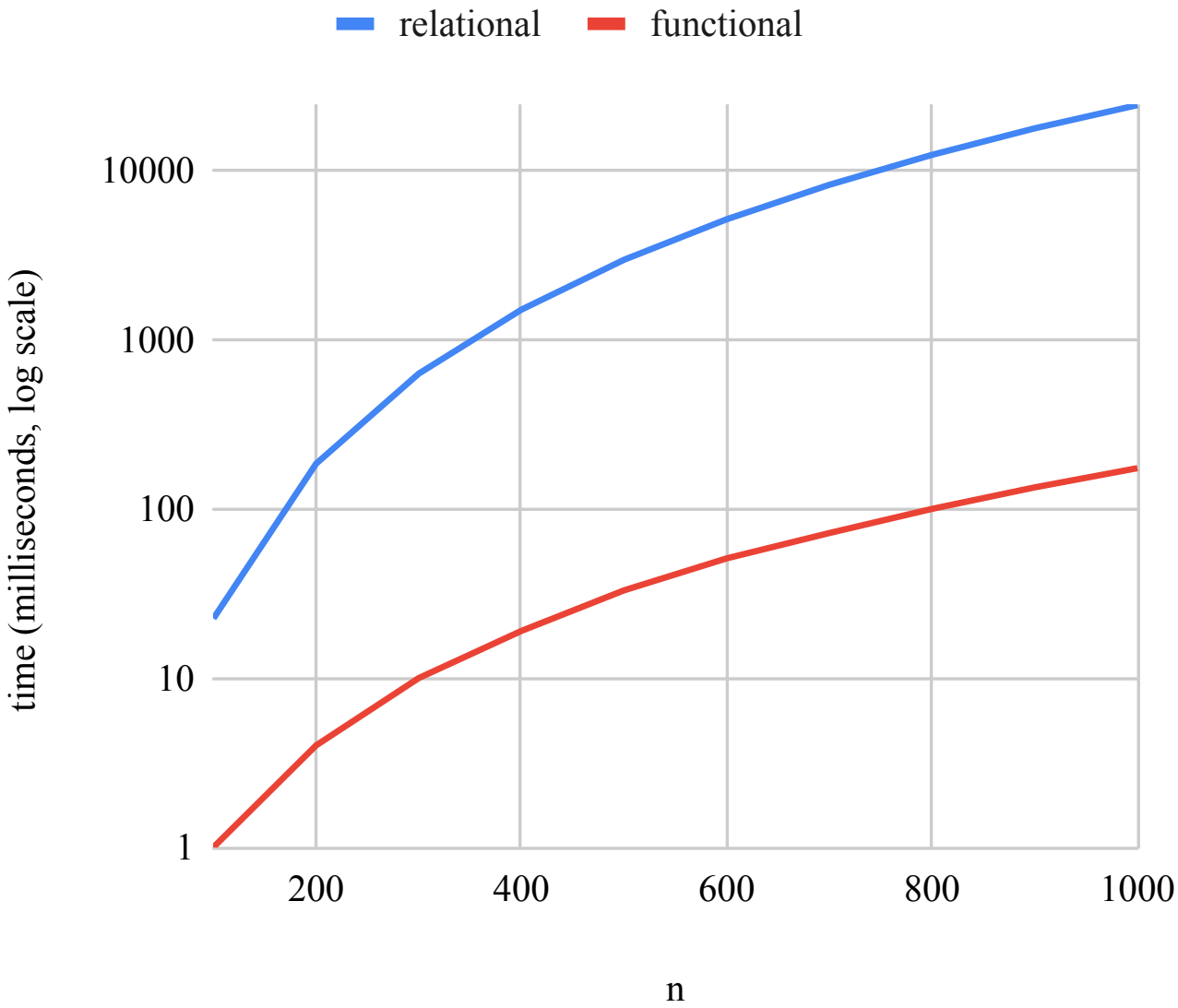
Generation of Logic Formulas:
`evalo [` **true** `;` **false** `;` **true** `] q` **true**



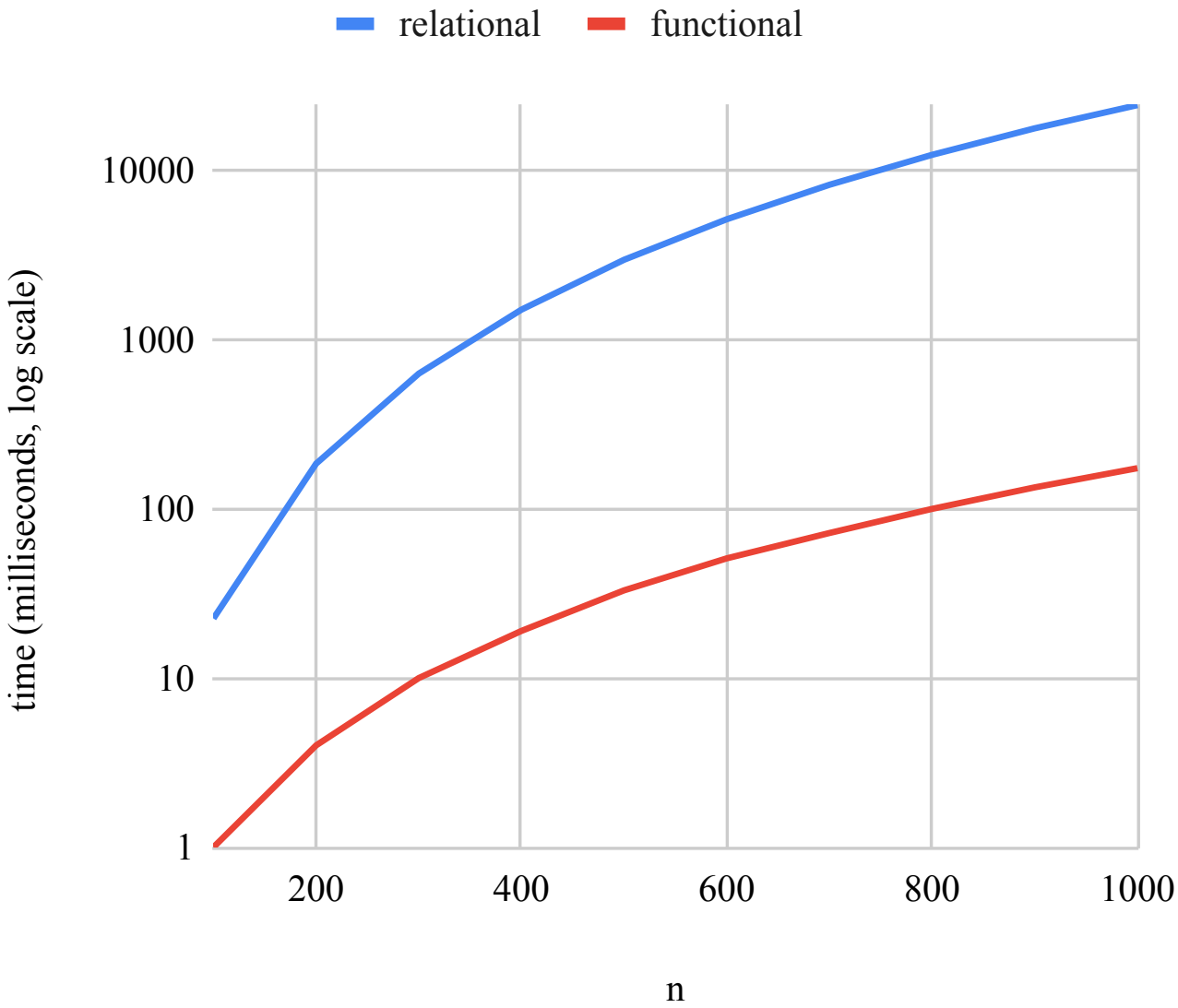
Multiplication: `mulo n 10 q`



Division: `mulo (n/10) q n`



Multiplication Generation: `take n (mulo 10 q r)`



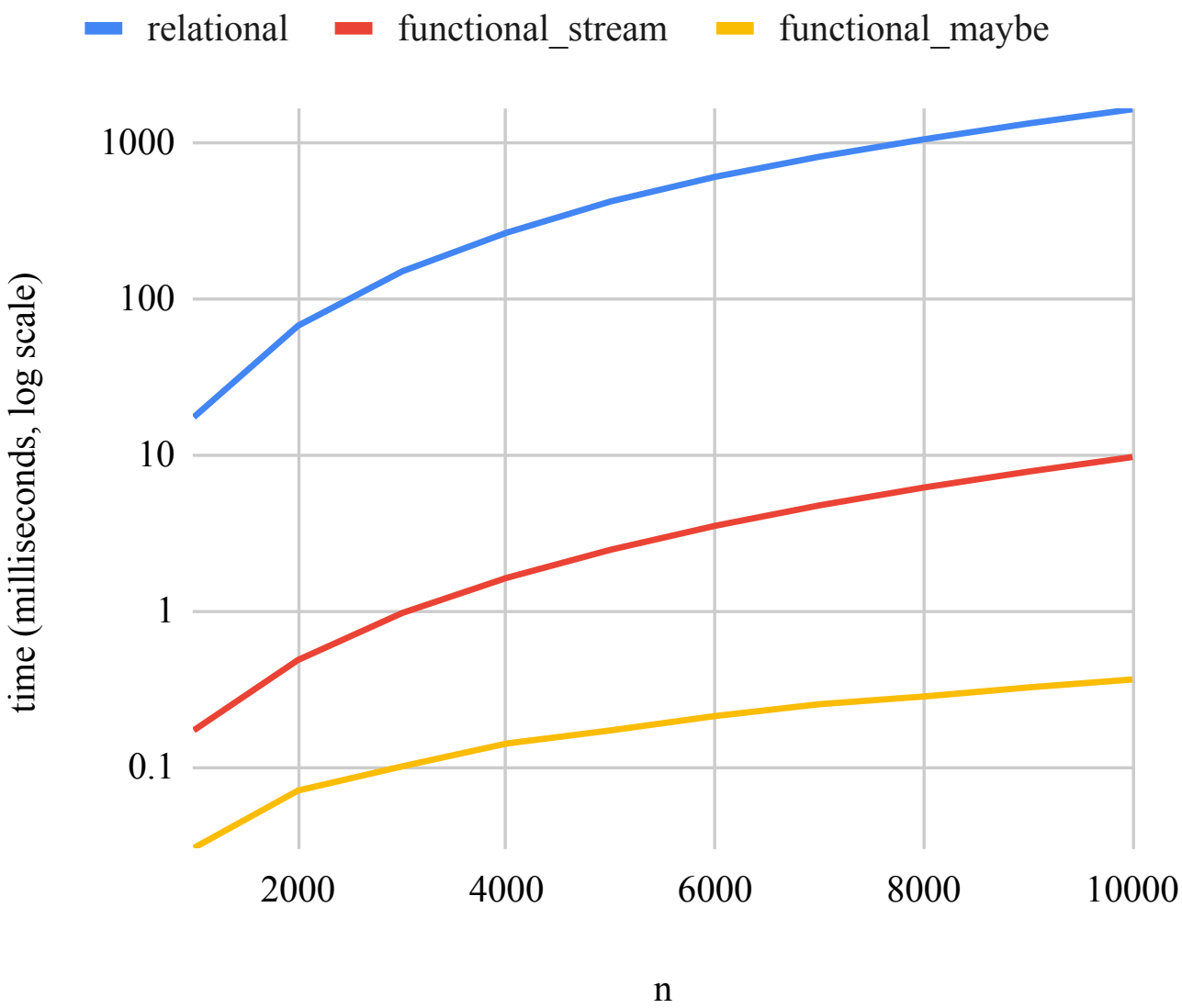
Data Types

We generate weird data type declarations:

```
type term =
  | Conj of (term * term)
  | Cons of (term * term)
  | Disj of (term * term)
  | Falso
  | Neg of term
  | Succ of term
  | Trueo
  | Var of term
  | Zero
```

```
elemᵒ i st v =
  fresh (h t i₁) conde [
    (i ≡ Zero ∧ st ≡ Cons (v, t));
    (i ≡ Succ i₁ ∧ st ≡ Cons (h, t) ∧ elemᵒ i₁ t v)]
```

Need for Determinism Check: `mulo q 10 1000`



Need for Determinism Check

- Replacing Stream with Maybe improves performance about 10 times for relations on natural numbers
- Functional (no monad) version is still faster
- Use determinism check to figure out when replacing Stream is feasible
- How to combine different monads naturally?

Need for Partial deduction

miniKanren can run a verifier backwards to get solver

run `q` (`eval`^{o} `q` **true**)

Augmenting functional conversion with partial deduction must be beneficial

Conclusion

Conclusion

- We presented a functional conversion scheme as a series of examples
- The conversion speeds up implementations considerably
- We implemented the conversion scheme in Haskell
- Found some way to order conjuncts

Future work

- Integration with partial deduction
- Integration into a relational interpreters for solving framework