LASIGE reliable software systems

EVOLUTIONARY PROGRAM SYNTHESIS FROM REFINED AND DEPENDENT TYPES

Paulo Santos

U LISBOA Ciências ULisboa FCT Fundação para a Ciência e a Tecnologia

MOTIVATION

Secure

Cheaper

Faster

Reliable

Inductive Synthesis

Also known as program synthesis from examples, where pairs of inputs/outputs are provided as the user intention.

MYTH

The algorithm uses *refinement trees* and pairs of inputs/outputs to generate the intended code.

Program synthesis problem in MYTH

```
let stutter : list → list |>
{ [] => []
| [0] => [0; 0]
| [1; 0] => [1; 1; 0; 0]
} = ?
```

Deductive Synthesis

Requires the introduction of a formal specification to declare the user intention.

Different ways to create the specification:

- Domain Specific Language (DSL)
- Contract-based synthesis
- Type System

SyGus

Uses logical constraints and syntactic templates to restrict the space of implementations.

Minimum synthesis in SyGus

```
(synth-fun min ((x Int) (y Int)) Int

  ;; Non-terminals that would be used in the grammar
  ((I Int) (B Bool))

  ;; Define the grammar for allowed implementations of min
  ((I Int (x y 0 1 (+ I I) (- I I) (ite B I I)))
  (B Bool ((and B B) (or B B) (not B)
  (= I I) (<= I I) (>= I I)))))

(declare-var x Int)
(declare-var y Int)

(constraint (<= (min x y) x))
(constraint (<= (min x y) y))
(constraint (or (= x (min x y)) (= y (min x y))))
(check-synth)
```

The categories of benchmarks in which state-of-the-art solvers excel are those with a single function invocation, a single function to synthesize, a complete specification, no use of `let`, and a restricted grammar.

Search Based Program Synthesis, in CACM

SYNQUID

Primarily uses refined polymorphic types to restrict the search space of valid programs.

Incomplete programs are type checked during synthesis, ensuring only correct programs to be generated.

However, it requires the user to specify every single component used in the synthesis.

Absolute values of a list in SYNQUID*

```
(absolutes :: List Int -> List Nat
absolutes = \xs .
  map (?? :: x: Int -> {Int | _v == x || _v == -x}) xs
```

*http://comcom.csail.mit.edu/

ÆON

ÆON is a general purpose programming language that uses refined and dependent types to synthesize complete or partial programs.

- Restricted refined types are used to generate valid expressions.

- Non-restricted refined types are used to synthesize correct individuals.

ÆON

Example 1

LASIGE reliable software systems

Encrypt and decrypt in ÆON

```
type Key {
   {key : Integer | key >= 0 && key <= 1024};
}

decrypt(i : Integer, k : Key) → {j : Integer | j > 0} {
      i – getKey(k);
}

encrypt(i : Integer, k : Key) → {j : Integer | i == decrypt(j, k.key)} {
      i + getKey(k);
}
```

ULISBOA Ciências ULisboa FCT Fundação para a Ciência e a Tecnologia

ÆON

Example 2

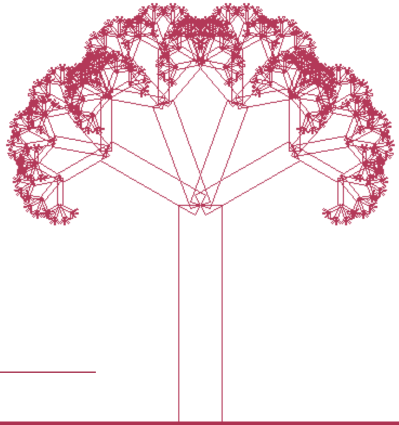
Complete synthesis of turtle problem in ÆON

```
import images;

type Turtle {
    img : Image;
    direction : Double;
    {x : Double | x >= 0 && x <= img.width};
    {y : Double | y >= 0 && y <= img.height};
}

img : Image = loadImage("tree.png", 50, 150);
empty : Image = buildImage(50, 150);
turtle : Turtle = buildTurtle(empty, 0, 0, 0);

drawTree(size : Integer) → out : Image where {out.width == img.width and
                                              out.height == img.height and
                                              @minimize (difference(turtle.image, out))} {
    ■;
}
```



ÆON

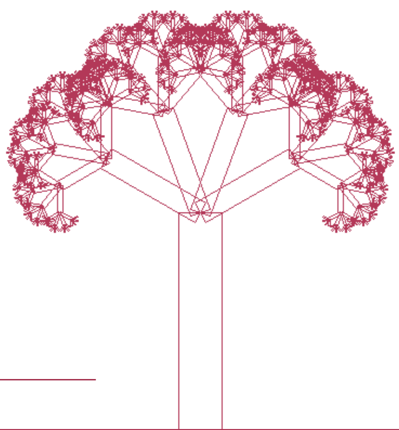
Example 2

Naïve solution in ÆON

```
drawTree(size : Integer) → out : Image where … {
    makeRectangle(size);
    x : Double = getX(turtle);
    y : Double = getY(turtle);
    dir : Double = getDirection(turtle);

    if (size / 5 < 2) {
        getImage(turtle);
    } else {
        turnTurtle(60);
        drawTree(size / 2);
        setTurtleX(x);
        setTurtleY(y);
        setTurtleDirection(dir);
        turnTurtle(20);
        drawTree(size / 2);
        setTurtleX(x);
        setTurtleY(y);
        setTurtleDirection(dir);

        turnTurtle(-20);
        drawTree(size / 2);
        setTurtleX(x);
        setTurtleY(y);
        setTurtleDirection(dir);
        turnTurtle(-60);
        drawTree(size / 2);
        setTurtleX(x);
        setTurtleY(y);
        setTurtleDirection(dir);
        getImage(turtle)
    }
}
```



ÆON

Example 3

Sort in ÆON

```
import arrays;
sort<T>(array : Array<T>) → out:Array<T> where {array.size == out.size and
                                               elems(array) == elems(out) and
                                               forall(range(out.size - 1), \i:Integer → elemAt(out, i) <= elemAt(out, i + 1))} {
    if (size(array) <= 1) {
        array;
    } else {
        x : T = head(array);
        xs : Array<T> = tail(array);
        lesser : Array<T> = sort<T>(filter(\p : T → p < x, xs));
        greater : Array<T> = sort<T>(filter(\p : T → p >= x, xs));
        concat(append(lesser, x), greater);
    }
}
```

ÆON

Example 3

LASIGE reliable software systems

Partial synthesis of sort in ÆON

```
import arrays;
sort<T>(array : Array<T>) → out:Array<T> where {array.size == out.size and
                                elems(array) == elems(out) and
                                forall(range(out.size - 1), \i:Integer → elemAt(out, i) <= elemAt(out, i + 1))} {
  if (■) {
    ■;
  } else {
    x : T = ■;
    xs : Array<T> = ■;
    lesser : Array<T> = sort<T>(filter(■, ■));
    greater : Array<T> = sort<T>(■);
    concat(append(■, ■), ■);
  }
}
```

Partial synthesis of sort in ÆON

```
import arrays;
sort<T>(array : Array<T>) → out:Array<T> where {array.size == out.size and
                                   elems(array) == elems(out) and
                                   forall(range(out.size - 1), \i:Integer → elemAt(out, i) <= elemAt(out, i + 1))} {
    if (■) {
        ■;
    } else {
        ■;
    }
}
```

Complete synthesis of sort in ÆON

```
import arrays;
sort<T>(array : Array<T>) → out:Array<T> where {array.size == out.size and
                                               elems(array) == elems(out) and
                                               forall(range(out.size - 1), \i:Integer → elemAt(out, i) <= elemAt(out, i + 1))} {
      ■;
}
```

FROM ÆON TO ITS CORE

LASIGE reliable software systems

Update of vehicle ownership in a database in ÆON

```
updateVehicle(owner : String, id : Integer) → Integer {
    sql : String = “UPDATE vehicle SET owner=? WHERE id=?”;
    sql = setParameter(sql, owner);
    sql = setParameter(sql, id);
    print(sql);
    executeUpdate(sql);
}
```

Update of vehicle ownership in a database in ÆONCORE

```
updateVehicle : (owner : String) → (id : Integer) → (c : Integer) = \owner : String → \id :
Integer → (\sql : String → (\sql : String → (\sql : String → (_ : Void →
executeUpdate(sql)) (print(sql))) (setParameter(sql, id)) (setParameter(sql, owner)))
("Update vehicle SET owner=? WHERE id=?");
```

FROM ÆON TO ITS CORE

Example: Type declaration conversion

LASIGE reliable software systems

Type declarations in ÆON

```
type Car {
    year : Nat;
    brand : String;
    owner : String;
}
```

Uninterpreted functions of the Car Type in ÆONCORE

```
_Car_year(x : Car) → year : Nat = uninterpreted;
_Car_brand(x : Car) → brand : String = uninterpreted;
_Car_owner(x : Car) → owner : String = uninterpreted;
```

Kinds	$k \ ::= * \ \mid k \ \rightarrow \ k$
Types	$T \ ::= \textbf{Integer} \mid \textbf{Boolean} \mid t \mid x{:}T \rightarrow \ T$ $\mid x{:}T \textbf{ where } e \mid \forall t{:}k.T \mid TT$
Expressions	$e \ ::= \textbf{true} \mid \textbf{false} \mid n \mid x$ $\mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \lambda x{:}T.e \mid ee$ $\mid \Lambda t{:}k.e \mid e[T]$
Contexts	$\Gamma \ ::= \varepsilon \mid \Gamma, x{:}T \mid \Gamma, t{:}k \mid \Gamma, e$

NON-DETERMINISTIC SYNTHESIS
FOR POLYMORPHIC REFINEMENT TYPES

LASIGE reliable software systems

$$\frac{}{\Gamma \vdash \textbf{Boolean} \rightsquigarrow_d \textbf{true}, \textbf{false}} \qquad \frac{}{\Gamma \vdash \textbf{Integer} \rightsquigarrow_d n} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash T \rightsquigarrow_{d+1} x}$$

(SE-Bool, SE-Int, SE-Var)

$$\frac{\Gamma, x : T \vdash U \rightsquigarrow_d e}{\Gamma \vdash (x : T \to U) \rightsquigarrow_{d+1} (\lambda x : T . e)} \qquad \frac{\Gamma \vdash T \rightsquigarrow_d e_2 \qquad \Gamma \vDash e_1[e_2/x]}{\Gamma \vdash (x : T \textbf{ where } e_1) \rightsquigarrow_{d+1} e_2}$$

(SE-Abs, SE-Where)

$$\frac{\Gamma \vdash \textbf{Boolean} \rightsquigarrow_d e_1 \quad \Gamma, e_1 \vdash T \rightsquigarrow_d e_2 \quad \Gamma, \neg e_1 \vdash T \rightsquigarrow_d e_3}{\Gamma \vdash T \rightsquigarrow_{d+1} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}$$

(SE-If)

$$\frac{\Gamma, t : k \vdash T \rightsquigarrow_d e}{\Gamma \vdash (\forall t : k . T) \rightsquigarrow_{d+1} (\Lambda t : k . e)} \qquad \frac{\begin{array}{c} \rightsquigarrow_d k \quad \Gamma \vdash k \rightsquigarrow_d U \ (t \text{ fresh}) \\ \Gamma, t : k \vdash_{[U/t]} T \rightsquigarrow_d V \quad \Gamma \vdash (\forall t : k . V) \rightsquigarrow_d e \end{array}}{\Gamma \vdash T \rightsquigarrow_{d+1} e[U]}$$

(SE-TAbs, SE-TApp)

$$\frac{\begin{array}{c} \rightsquigarrow_d k \quad \Gamma \vdash k \rightsquigarrow_d U \quad \Gamma \vdash U \rightsquigarrow_d e_2 \quad (x \text{ fresh}) \\ \Gamma, x : U \vdash_{[e_2/x]} T \rightsquigarrow_d V \qquad \Gamma \vdash (x : U \to V) \rightsquigarrow_d e_1 \end{array}}{\Gamma \vdash T \rightsquigarrow_{d+1} e_1 e_2} \qquad \frac{\Gamma \vdash T \overset{:>}{\rightsquigarrow}_d U \qquad \Gamma \vdash U \rightsquigarrow_d e}{\Gamma \vdash T \rightsquigarrow_{d+1} e}$$

(SE-App) (SE-Sub)

LASIGE reliable software systems

Each rule has a weight related to the probability of being chosen.

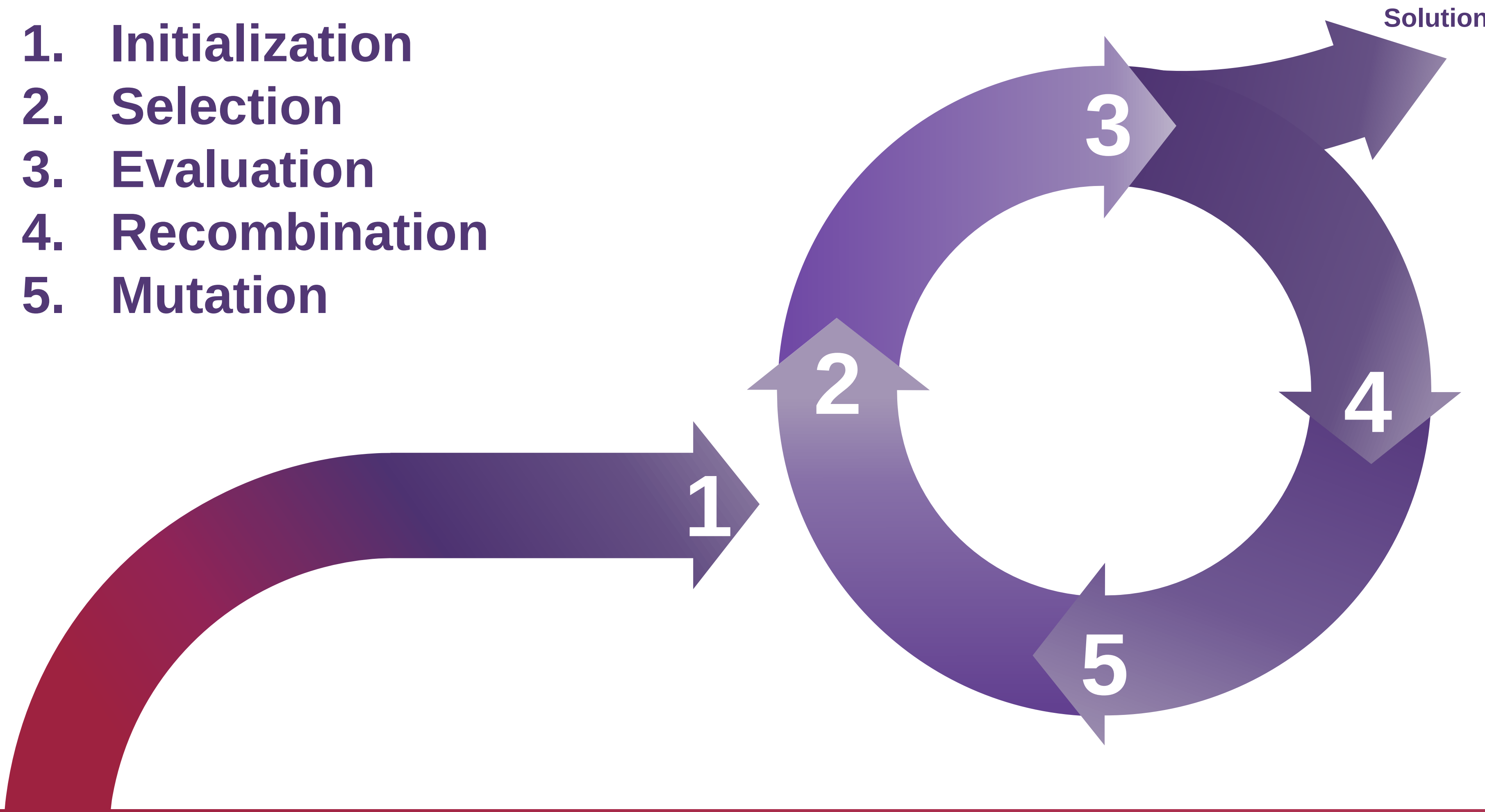
Synthesis Rule	Weight
SE-Bool	1
SE-Int	1
SE-Var	1
SE-App	1
SE-Where	1
SE-If	1
SE-Abs	1
SE-TAbs	1
SE-TApp	1
SE-Sub	1

EVOLUTIONARY SYNTHESIS

Genetic Programming

LASIGE reliable software systems

1. **Initialization**
2. **Selection**
3. **Evaluation**
4. **Recombination**
5. **Mutation**

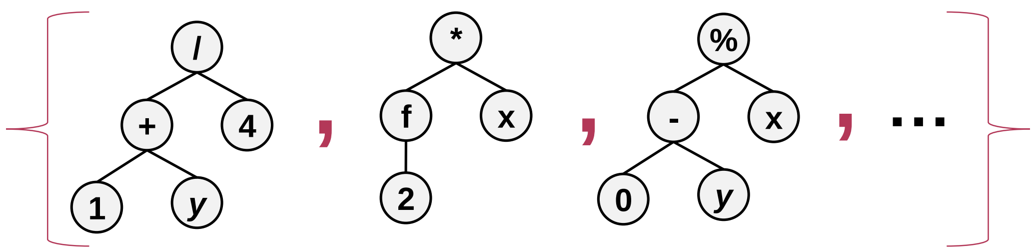


INDIVIDUAL

Each individual is composed by a list of **expressions** used to fill each hole of the original program and the fitness result for each objective.

Individual
expressions
fitness

[34.1, 22.1, 0.0]



INITIALIZATION

LASIGE reliable software systems

Create population of individuals by generating random expressions for each hole in the function from the non-deterministic synthesizer.

i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	$\dots$	i_n

EVALUATION

Fitness extraction

The function f, which is defined using these rules, is recursively used to convert logical predicates into continuous fitness function.

Logical predicates

1
2
3
4

Boolean	Continuous
$true, false$	$0.0, 1.0$
$x = y$	$\lvert x - y \rvert_N$
$x \neq y$	$1 - f(x = y)$
$a \vee b$	$f(a) * f(b)$
$a \wedge b$	$f(a) + f(b) - f(a)f(b)$
$a \rightarrow b$	$f(\neg a \vee b)$
$\neg a$	$1 - f(a)$
$x \leq y$	$\lvert (x - y)^+ \rvert_N$
$x < y$	$\lvert (x - y)^+ + \delta \rvert_N$

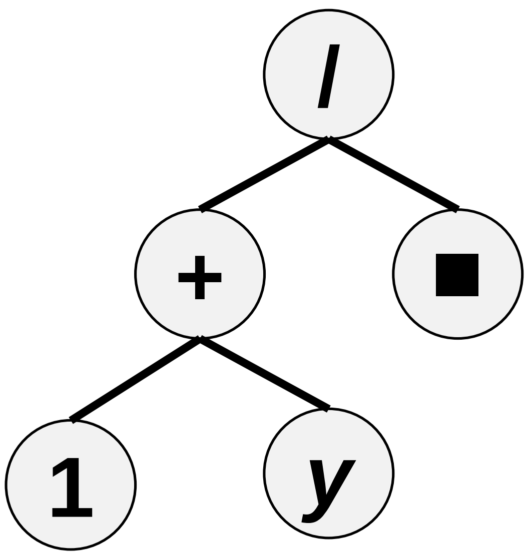
1
2
3
4

Fitness functions

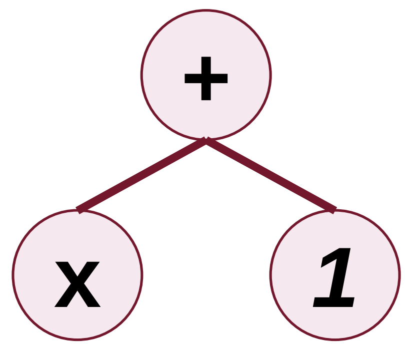
EVALUATION

Random Testing

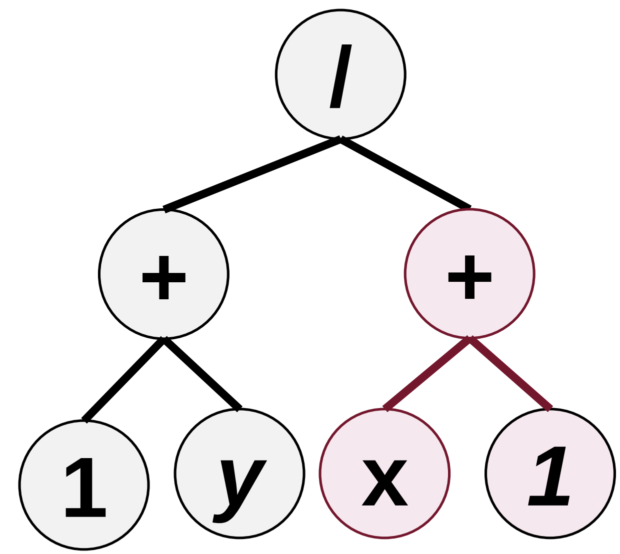
1. Each individuals holes are filled with the synthesized expressions;



Incomplete function



Synthesized expression



Complete Expression

EVALUATION

Random Testing

1. Each individuals holes are filled with the synthesized expressions;
2. Random inputs are generated from refined polymorphic types;

Caesars cipher and decipher in ÆON

```
type Key {
    {key : Integer | key >= 0 && key <= 1024};
}

decrypt({i : Integer | i > 0}, k : Key) → {j : Integer | j > 0} {
    ■;
}
```

t_1	$i = 1$	$k = \{k.key = 1024\}$
t_2	$i = 130$	$k = \{k.key = 2\}$
t_3	$i = 144$	$k = \{k.key = 1\}$
t_4	$i = 10$	$k = \{k.key = 30\}$
t_5	$i = 43$	$k = \{k.key = 55\}$
t_6	$i = 3$	$k = \{k.key = 999\}$

EVALUATION

Random Testing

LASIGE reliable software systems

1. Each individuals holes are filled with the synthesized expressions;
2. Random inputs are generated from refined polymorphic types;
3. Obtain the results of running each test for the filled individual;

t_1	$i = 1$	$k = \{k.key = 1024\}$	$out = 1$
t_2	$i = 130$	$k = \{k.key = 2\}$	$out = 230$
t_3	$i = 144$	$k = \{k.key = 1\}$	$out = 44$
t_4	$i = 10$	$k = \{k.key = 30\}$	$out = 40$
t_5	$i = 43$	$k = \{k.key = 55\}$	$out = 31$
t_6	$i = 3$	$k = \{k.key = 999\}$	$out = 0$

EVALUATION

Random Testing

1. Each individuals holes are filled with the synthesized expressions;
2. Random inputs are generated from refined polymorphic types;
3. Obtain the results of running each test for the filled individual;
4. Evaluate the result using the extracted fitness functions from the predicates.

t_1	$i = 1$	$k = \{k.key = 1024\}$	$out = 1$
t_2	$i = 130$	$k = \{k.key = 2\}$	$out = 230$
t_3	$i = 144$	$k = \{k.key = 1\}$	$out = 44$
t_4	$i = 10$	$k = \{k.key = 30\}$	$out = 40$
t_5	$i = 43$	$k = \{k.key = 55\}$	$out = 31$
t_6	$i = 3$	$k = \{k.key = 999\}$	$out = 0$

1 → $\sum_{i=0}^{n} f_1(t_i)$

2 → 6.0

3 → 0.0

4 → 3.10

SELECTION

ε-Lexicase Selection

Population

i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	…	i_n

1 2 3 4 Objectives

SELECTION

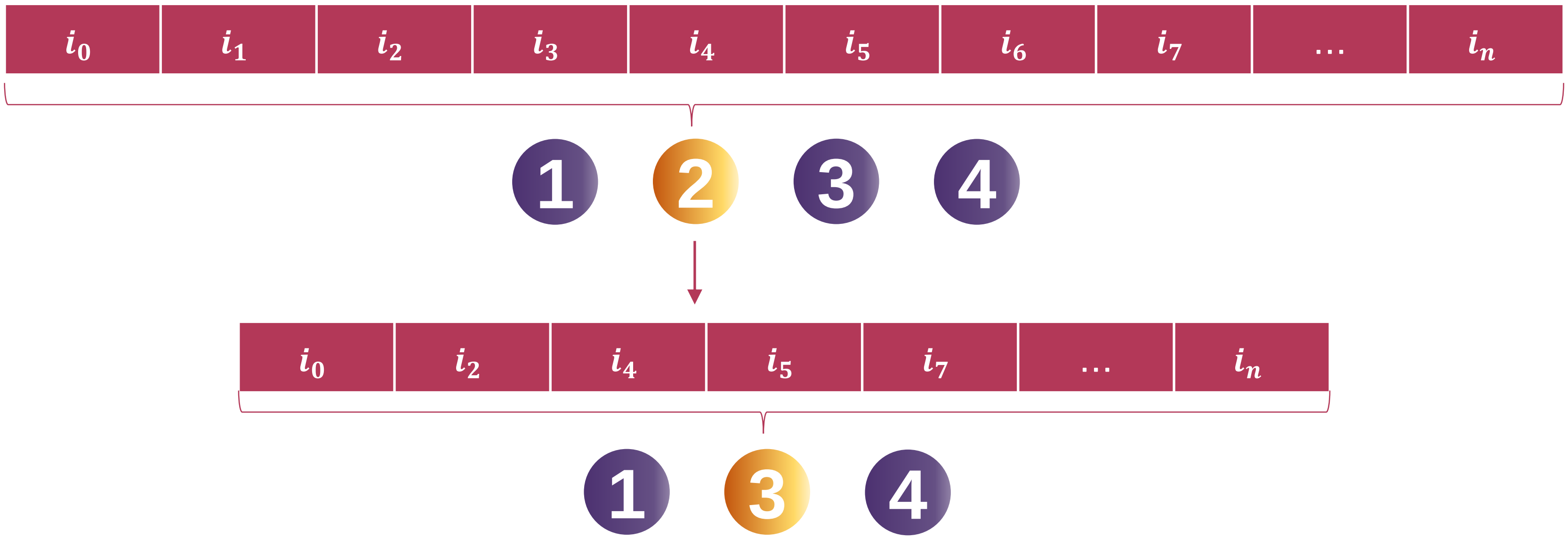
ε-Lexicase Selection

i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	$\dots$	i_n

1 2 3 4

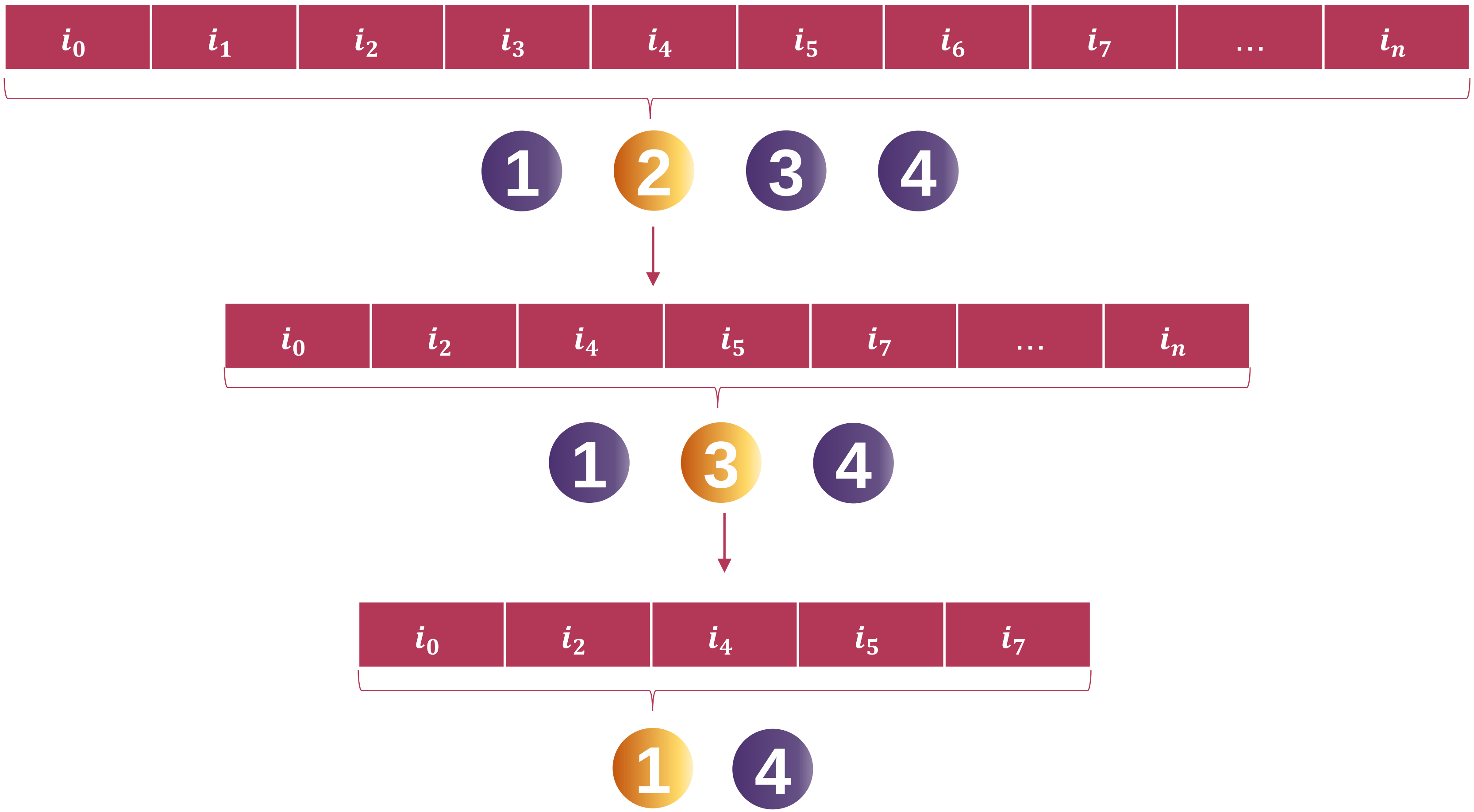
SELECTION

ε-Lexicase Selection



SELECTION

ε-Lexicase Selection



SELECTION

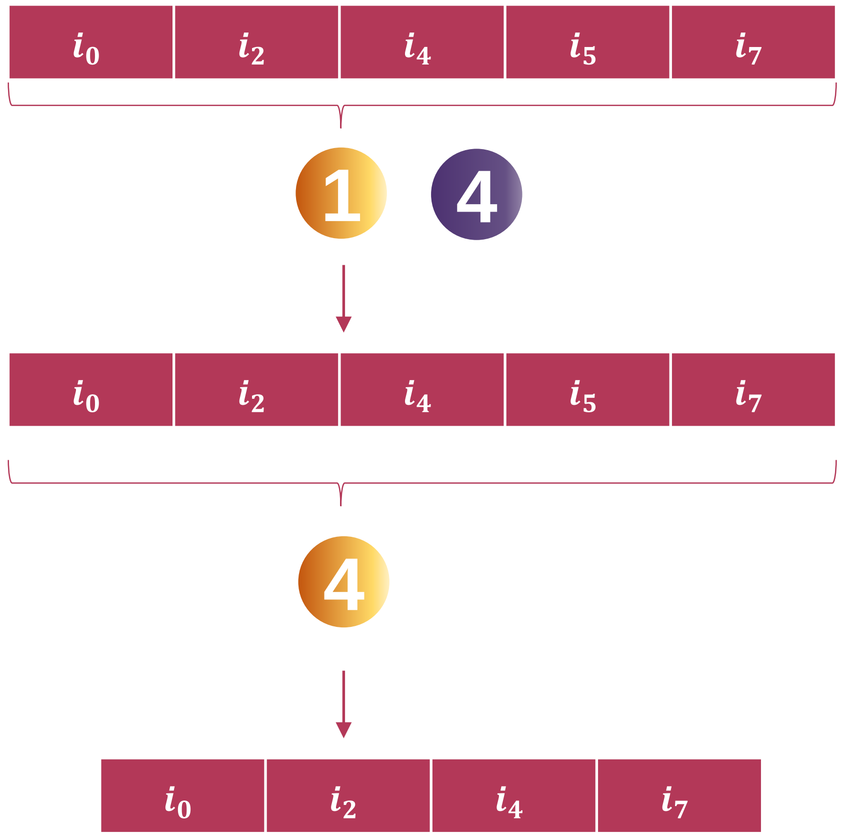
ε-Lexicase Selection

i_0	i_2	i_4	i_5	i_7

1 4

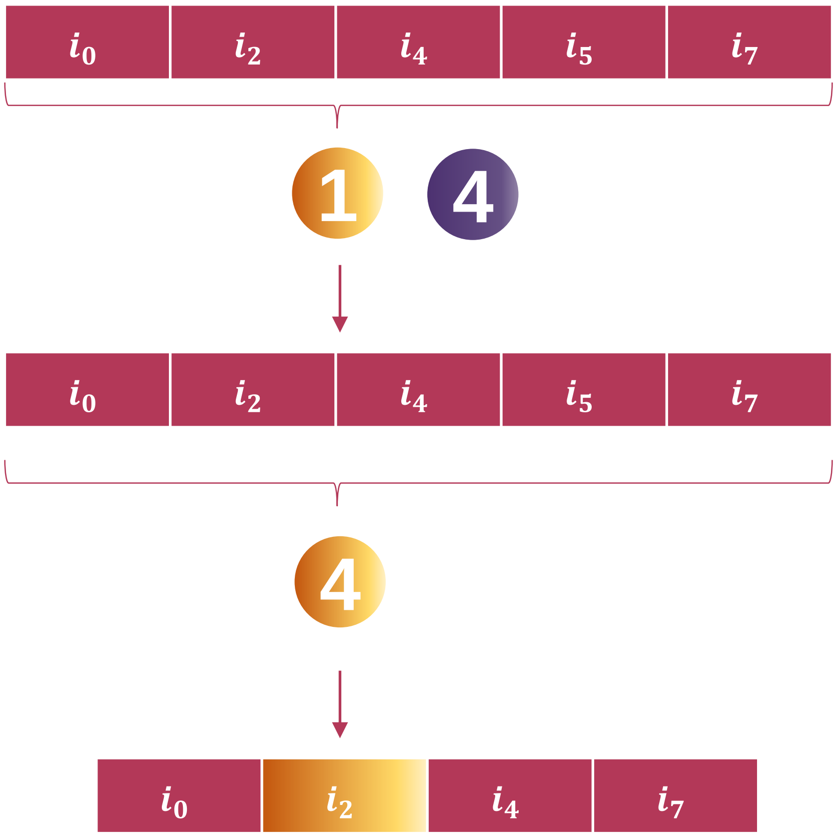
SELECTION

ε-Lexicase Selection



SELECTION

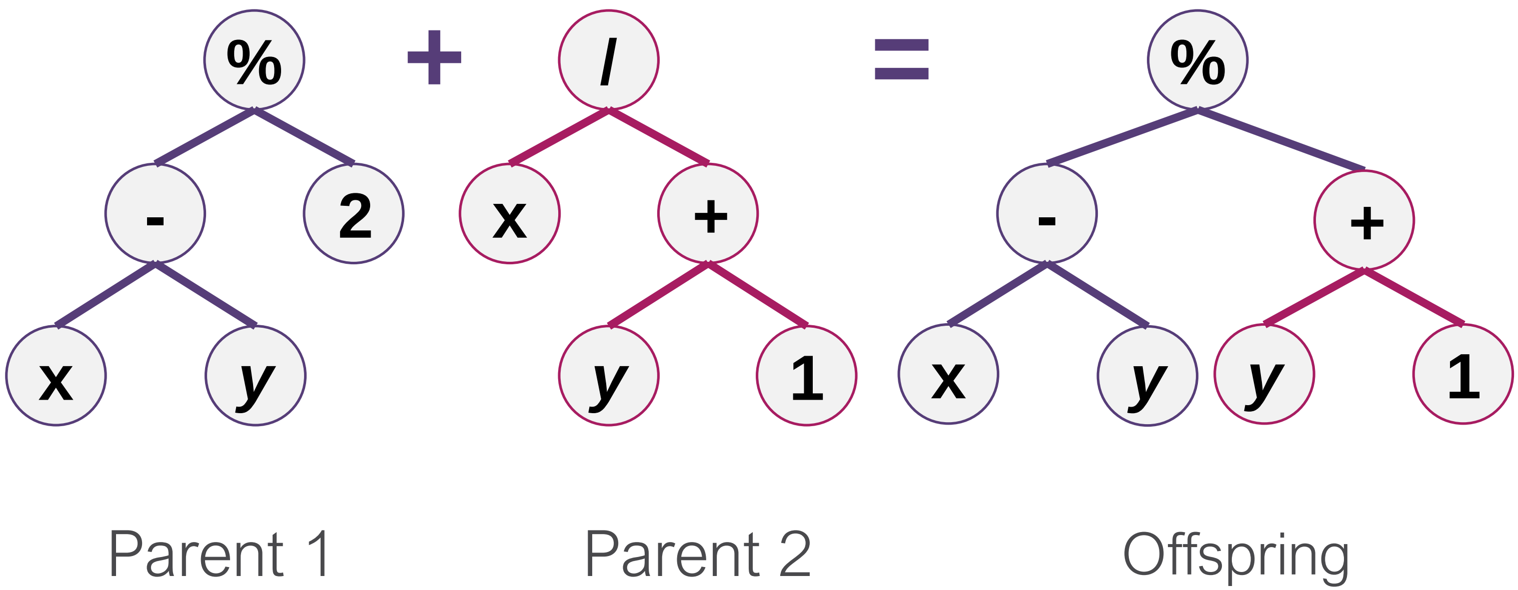
ε-Lexicase Selection



CROSSOVER

ε-Lexicase selection to choose two random individuals.

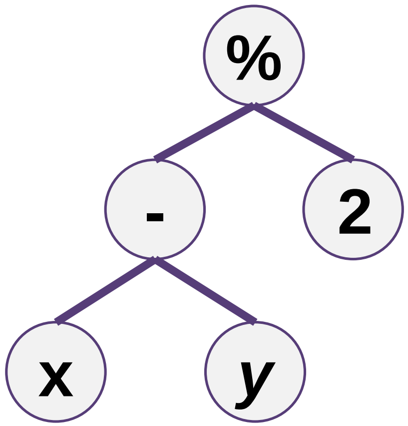
A random node with the same type of the selected one is chosen for crossover.



Future work might include partial crossovers!

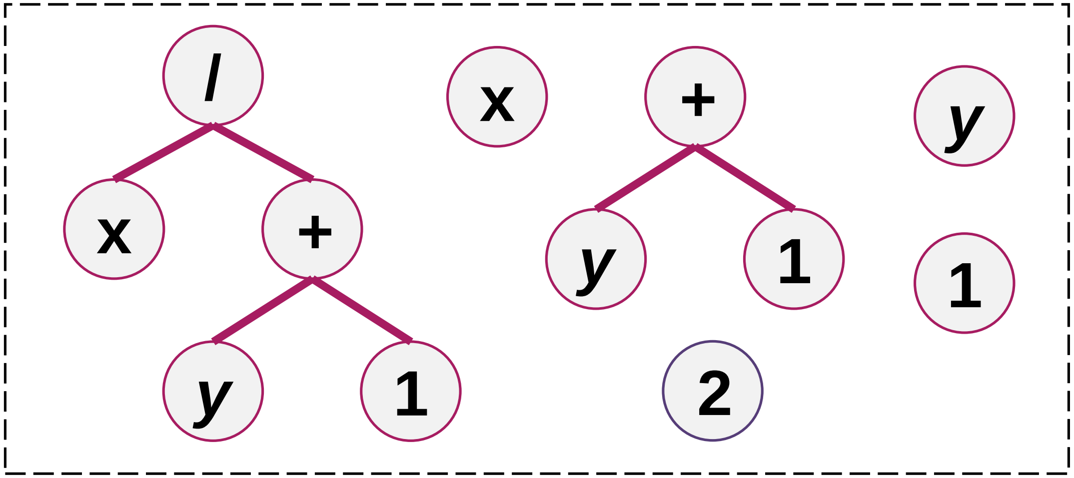
CROSSOVER

- If no node is found, split the second individual and use it as genetic material for the non-deterministic synthesizer.



Parent 1

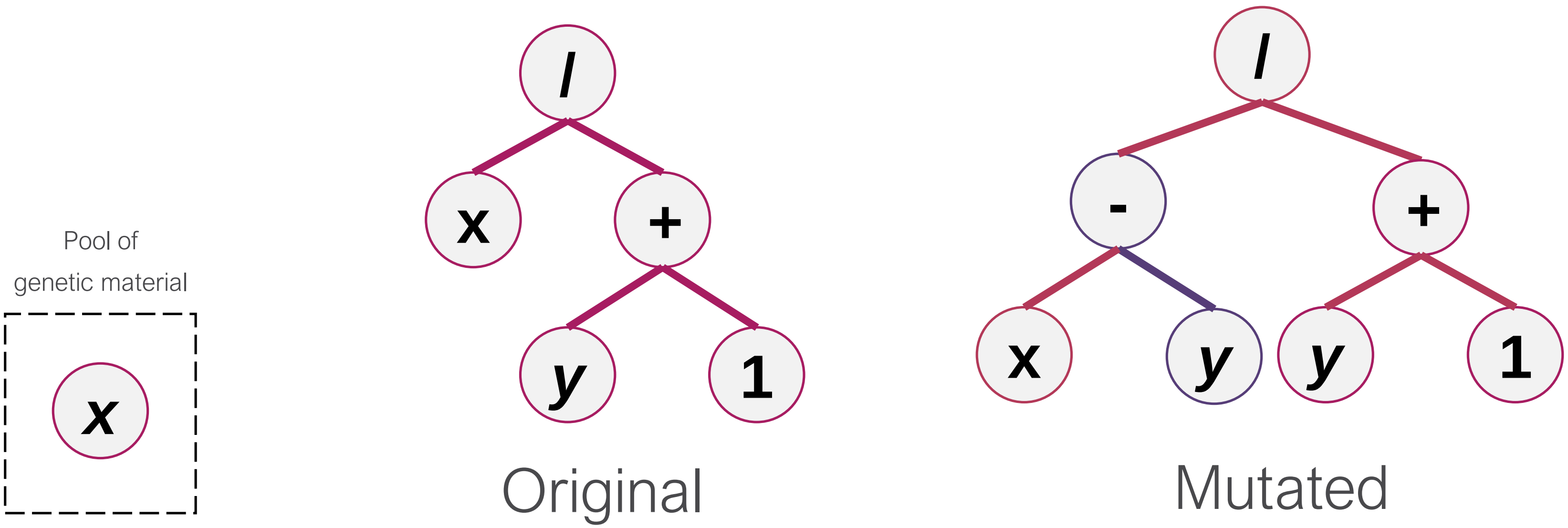
Pool of genetic material



MUTATION

A random node is chosen and its type is used in the synthesizer to generate a mutation.

Remaining subtrees are also used as genetic material, allowing the partial synthesis of programs.



EVALUATION

Work in Progress!

1. **Usability**
   - Evaluate language usability with students and researchers.
2. **Performance**
   - Comparison on successful synthesis on synthesis benchmarks.
3. **Versatility**
   - Applying the different techniques implemented on automatic repair system.

CURRENT FUTURE WORK

Work for the Thesis

1. **Optimization of the synthesis procedure**
   - Adaptive inductive biased weights optimization in ÆONCORE
2. **Optimization of the generated code**
   - Removal of garbage code from synthesized programs
3. **Individuals evaluation improvement**
   - Other than random tests, also include automatic edge case tests depending on refinements
4. **ÆON automatic program repair system from non-restricted refined types**
5. **Evaluation**
6. **Write thesis!**

LASIGE reliable software systems

EVOLUTIONARY PROGRAM SYNTHESIS FROM REFINED AND DEPENDENT TYPES

Thank you!

ULISBOA Ciências ULisboa FCT Fundação para a Ciência e a Tecnologia

HELPER

Index

- Æoncore: Expression Type Synthesis
- Æoncore: Type Synthesis
- 9-Puzzle Synthesis Example

ÆONCORE

Type Synthesis

$$\frac{}{\Gamma \vdash * \rightsquigarrow_d \textbf{Integer}} \qquad \frac{}{\Gamma \vdash * \rightsquigarrow_d \textbf{Boolean}} \qquad \frac{t \colon k \in \Gamma}{\Gamma \vdash k \rightsquigarrow_d t}$$

(ST-Int, ST-Bool, ST-Var)

$$\frac{\Gamma \vdash * \rightsquigarrow_d T \quad \Gamma, x \colon T \vdash * \rightsquigarrow_d U}{\Gamma \vdash * \rightsquigarrow_{d+1} (x \colon T \to U)} \qquad \text{(ST-Abs)}$$

$$\frac{\Gamma \vdash k \rightsquigarrow_d T \quad \Gamma, x \colon T \vdash \textbf{Boolean} \rightsquigarrow_d e}{\Gamma \vdash k \rightsquigarrow_{d+1} (x \colon T \ \textbf{where} \ e)} \qquad \text{(ST-Where)}$$

$$\frac{\Gamma, t \colon k \vdash k' \rightsquigarrow_d T}{\Gamma \vdash (k \to k') \rightsquigarrow_{d+1} (\forall t \colon k.T)} \qquad \text{(ST-TAbs)}$$

$$\frac{\rightsquigarrow_d k' \quad \Gamma \vdash (k' \to k) \rightsquigarrow_d T \quad \Gamma \vdash k' \rightsquigarrow_d U}{\Gamma \vdash k \rightsquigarrow_{d+1} T U} \qquad \text{(ST-TApp)}$$

ÆONCORE

Expression Type Synthesis

$$\frac{b = \textbf{true}, \textbf{false}}{\Gamma \vdash b \Rightarrow [b\colon \textbf{Boolean}]} \quad \text{(T-Bool)}$$

$$\frac{}{\Gamma \vdash n \Rightarrow [n\colon \textbf{Integer}]} \qquad \frac{x\colon T \in \Gamma}{\Gamma \vdash x \Rightarrow [x\colon T]} \quad \text{(T-Int, T-Var)}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \textbf{Boolean} \quad \Gamma, e_1 \vdash e_2 \Rightarrow T \quad \Gamma, \neg e_1 \vdash e_3 \Rightarrow U}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Rightarrow T \sqcup U} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash T \Leftarrow * \quad \Gamma, x\colon T \vdash e \Rightarrow U \quad \Gamma, x\colon T \vdash U \Leftarrow *}{\Gamma \vdash (\lambda x\colon T.e) \Rightarrow [(\lambda x\colon T.E)\colon (x\colon T \to U)]} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow V \quad \Gamma \vdash V \Downarrow (x\colon T \to U) \quad \Gamma \vdash e_2 \Leftarrow T}{\Gamma \vdash e_1 e_2 \Rightarrow [e_1 e_2\colon U[e_2/x]]} \quad \text{(T-App)}$$

$$\frac{\Gamma, t\colon k \vdash e \Rightarrow T}{\Gamma \vdash (\Lambda t\colon k.e) \Rightarrow [(\Lambda t\colon k.e)\colon (\forall t\colon k.T)]} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash e \Rightarrow V \quad \Gamma \vdash V \Downarrow (\forall t\colon k.U) \quad \Gamma \vdash T \Leftarrow k}{\Gamma \vdash e[T] \Rightarrow [e[T]\colon U[T/t]]} \quad \text{(T-TApp)}$$

LASIGE reliable software systems

9-puzzle solver synthesis in ÆON

```
import arrays;

type Puzzle {
    pieces : Array<Integer>;
}

native up({puzzle : Puzzle | pos(0, puzzle.pieces) < 8}) → out : Puzzle;
native down({puzzle : Puzzle | pos(0, puzzle.pieces) > 2}) → out : Puzzle;
native left({puzzle : Puzzle | pos(0, puzzle.pieces) % 3 > 1}) → out : Puzzle;
native right({puzzle : Puzzle | pos(0, puzzle.pieces) % 3 > 0}) → out : Puzzle;

solve(puzzle : Puzzle) → out : Puzzle where { len(puzzle.pieces) == len(out.pieces) and
                    forall(range(0, len(out.pieces)), \x : Integer → elem(out.puzzle, x) == x) } { ■; }
```