

Implementing the λ_{GT} Language: A Functional Language with Graphs as First-Class Data

ICGT2023

July 20, 2023

Waseda University, Tokyo, Japan

Jin SANO Kazunori UEDA

Overview

We designed a new purely functional language λ_{GT} ,
which handles graphs as first-class data
with pattern matching¹.



We built a reference implementation of the language
in only 500 lines of OCaml code.

Source <https://github.com/sano-jin/lambda-gt-alpha/tree/icgt2023>

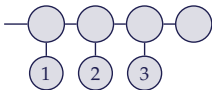
Try it at <https://sano-jin.github.io/lambda-gt-playground/>

¹Sano et al. “Type Checking Data Structures More Complex than Trees”. *Journal of Information Processing*. 2023.

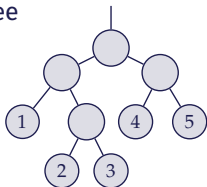
Incorporating GT into general-purpose PL

Algebraic Data Types

List



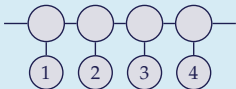
Tree



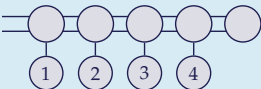
First-class values in λ_{GT}

^{def} = Data that can be input or output of functions

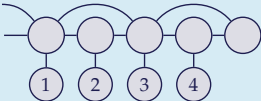
Queue/Difference List



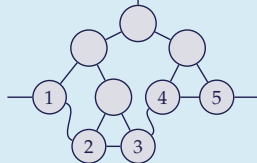
Doubly-Linked List



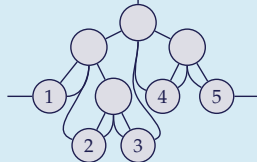
Skip List



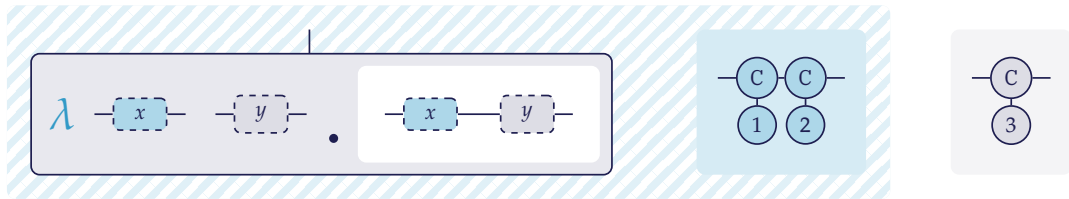
Leaf-Linked Tree



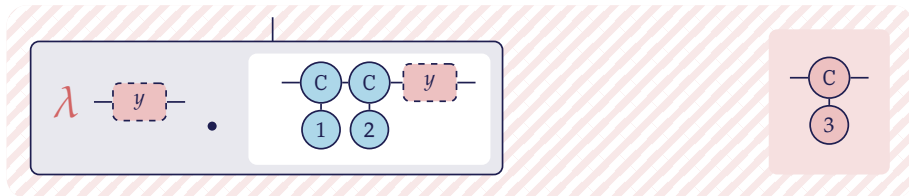
Threaded Tree



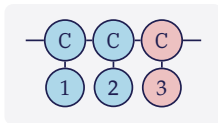
Demo Difference Lists Concatenation in λ_{GT}



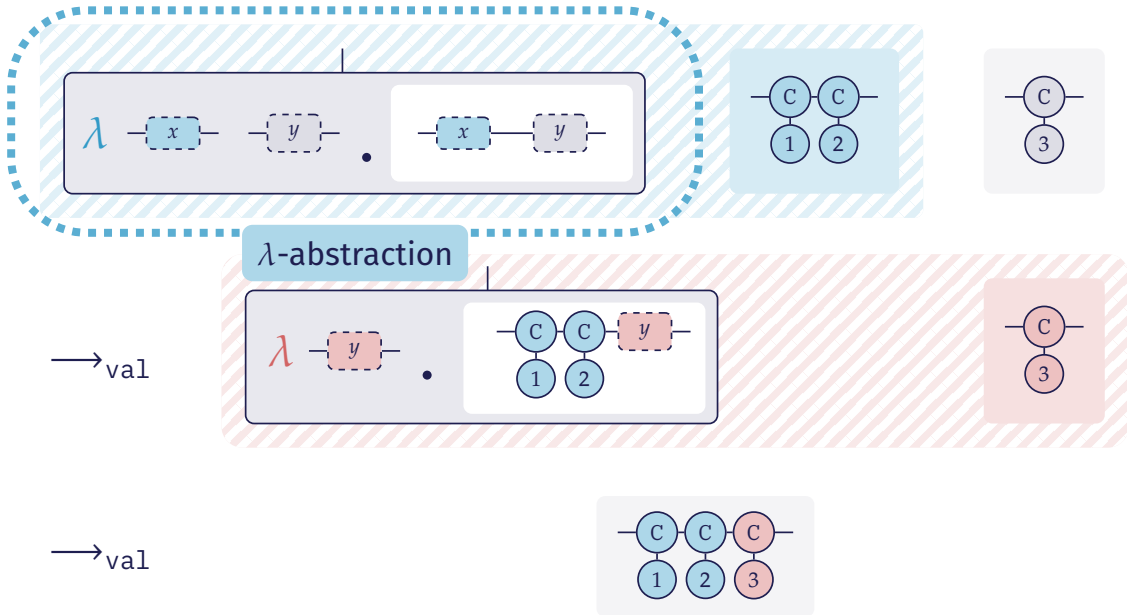
\longrightarrow_{val}



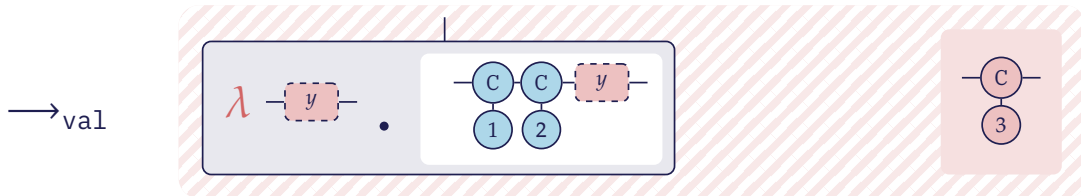
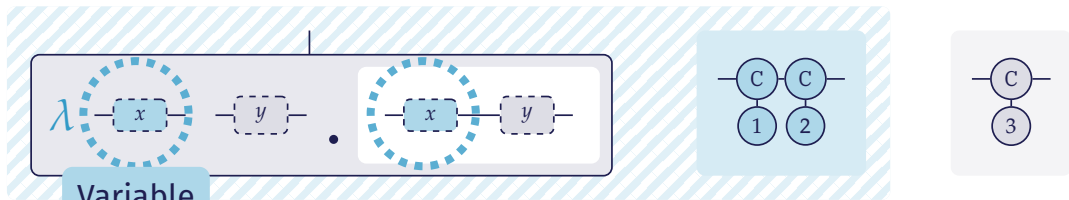
\longrightarrow_{val}



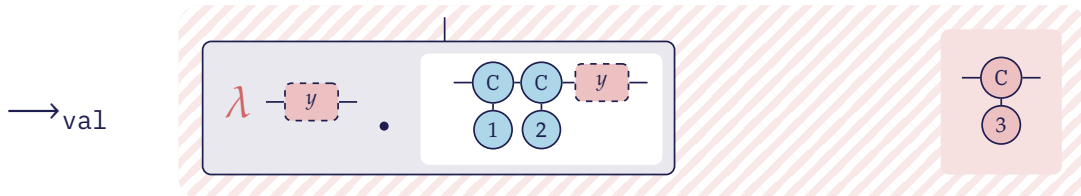
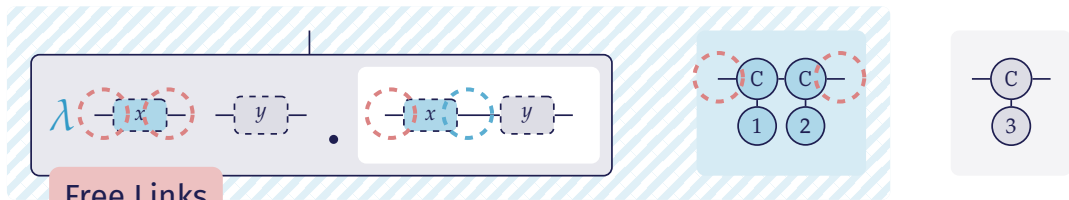
Demo Difference Lists Concatenation in λ_{GT}



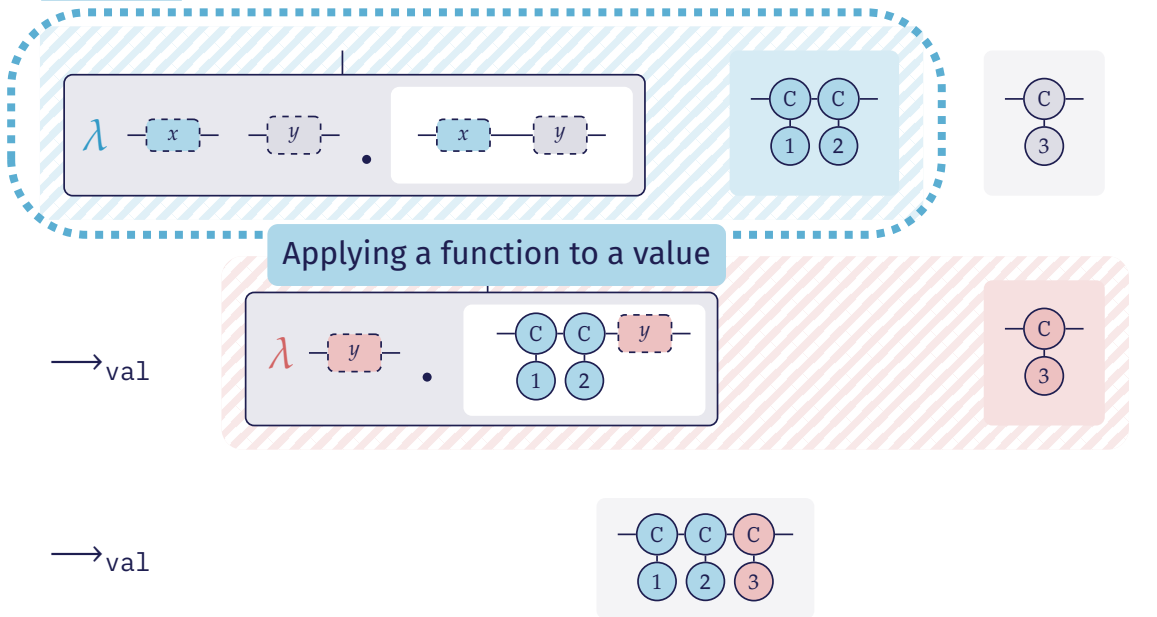
Demo Difference Lists Concatenation in λ_{GT}



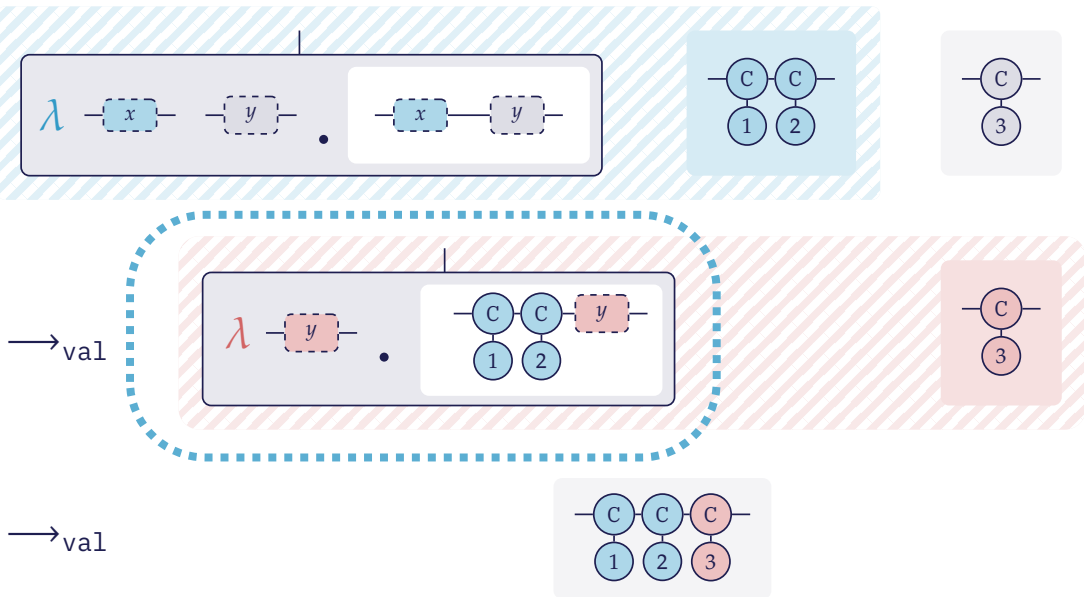
Demo Difference Lists Concatenation in λ_{GT}



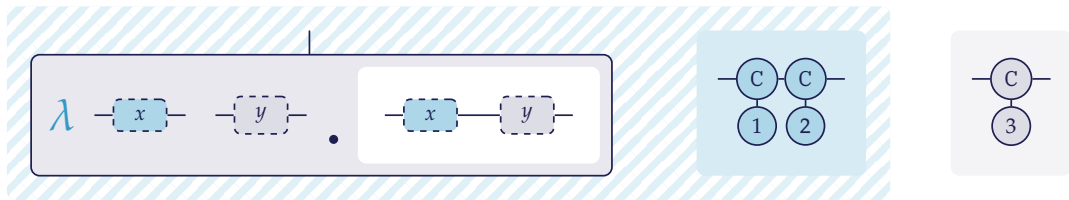
Demo Difference Lists Concatenation in λ_{GT}



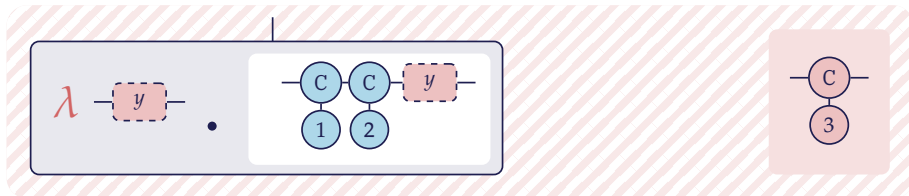
Demo Difference Lists Concatenation in λ_{GT}



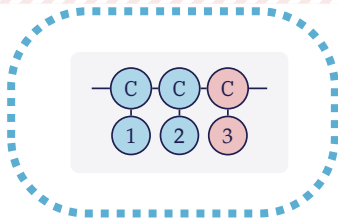
Demo Difference Lists Concatenation in λ_{GT}



\longrightarrow_{val}

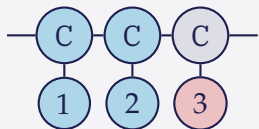


\longrightarrow_{val}



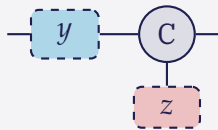
Demo Pattern Matching in λ_{GT}

If $\text{---}[x]\text{---}$ is bound to



, then

case $\text{---}[x]\text{---}$ **of**



→

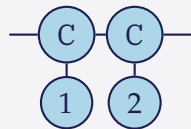


| **otherwise**

→



→_{val}



Remove the last element of a difference list

Formalising λ_{GT}

In the previous work, we gave a formal syntax and semantics to λ_{GT} and designed a new type system.

- ✓ To formalise λ_{GT} , we incorporated graph transformation to simply typed call-by-value λ -calculus.

We employed the formulation of HyperLMNtal instead of ordinary GTs.

LMNtal
(port) graph
+
rewrite rules

HyperLMNtal
(port) hypergraph
+
rewrite rules

λ_{GT}
(port) hypergraph
+
functional lang.
with pattern matching

\subseteq

$=$

$=$

\subseteq

HyperLMNtal: A Graph Transformation (GT) Formalism

	Ordinary GTs	HyperLMNtal / λ_{GT}
Graphs	...	Port hypergraphs with free links
Equivalence of graphs	Graph Isomorphism	Structural Congruence (SC) also found in process algebra
Matching and Rewriting	Double Pushout, ...	Matching modulo SC + explicit handling of contexts

The syntax-directed approach of HyperLMNtal has high affinity with standard/mainstream programming languages.

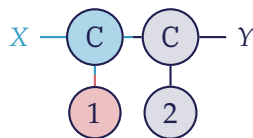
Syntax of Graphs in HyperLMNtal

Graph

$G ::= \mathbf{0}$	Null	<i>empty graph</i>	$\vec{X} = X_1, \dots, X_n$
$p(\vec{X})$	Atom	<i>vertex with a name p and links \vec{X}</i>	
$X \bowtie Y$	Fusion	<i>link connection</i>	
(G, G)	Molecule	<i>composition of subgraphs</i>	
$\nu X. G$	Hyperlink Creation	<i>hiding of link names</i>	

For example, a *difference list* can be represented as

$\nu Z. ($
 $\quad \nu Z_1. (\text{Cons}(Z_1, Z, X), 1(Z_1)),$
 $\quad \nu Z_2. (\text{Cons}(Z_2, Y, Z), 2(Z_2))$
 $)$



Structural Congruence (\equiv): Axioms of graph equivalence

- (E1) $(\mathbf{0}, G) \equiv G$
- (E2) $(G_1, G_2) \equiv (G_2, G_1)$
- (E3) $(G_1, (G_2, G_3)) \equiv ((G_1, G_2), G_3)$
- (E4) $G_1 \equiv G_2 \Rightarrow (G_1, G_3) \equiv (G_2, G_3)$
- (E5) $G_1 \equiv G_2 \Rightarrow \nu X. G_1 \equiv \nu X. G_2$
- (E6) $\nu X. (X \bowtie Y, G) \equiv \nu X. G \langle Y/X \rangle$
where $X \in \text{fn}(G) \vee Y \in \text{fn}(G)$
- (E7) $\nu X. \nu Y. X \bowtie Y \equiv \mathbf{0}$
- (E8) $\nu X. \mathbf{0} \equiv \mathbf{0}$
- (E9) $\nu X. \nu Y. G \equiv \nu Y. \nu X. G$
- (E10) $\nu X. (G_1, G_2) \equiv (\nu X. G_1, G_2)$
where $X \notin \text{fn}(G_2)$

For example,

$$\begin{aligned} & \nu Z. (\\ & \quad \nu Z_1. (\text{Cons}(Z_1, Z, X), 1(Z_1)), \\ & \quad \nu Z_2. (\text{Cons}(Z_2, Y, Z), 2(Z_2)) \\ &) \\ & \equiv \\ & \nu Z. (\\ & \quad \nu Z_1. (1(Z_1), \text{Cons}(Z_1, Z, X)), \\ & \quad \nu Z_2. (\text{Cons}(Z_2, Y, Z), 2(Z_2)) \\ &) \end{aligned}$$

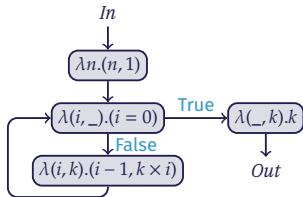
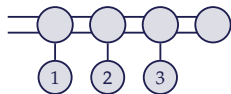
by (E2), (E4) and (E5)

Syntax and Semantics of λ_{GT}

λ_{GT} is a simple call-by-value λ -calculus extended with HyperLMNtal.

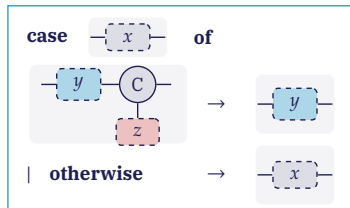
Extending graphs

Graphs
augmented with
 λ -abstractions
(embedded in node names)



Extending λ -expressions

λ -expressions
augmented with
Graph templates
(graphs with variables)



Syntax of λ_{GT}

Value $G ::= \mathbf{0} \mid p(\vec{X}) \mid X \bowtie Y \mid (G, G) \mid \nu X.G$

Atom Name $p ::= C \mid \lambda x[\vec{X}].e$

Expression $e ::= (\mathbf{case} \ e \ \mathbf{of} \ T \rightarrow e \mid \mathbf{otherwise} \rightarrow e) \mid (e \ e) \mid T$

Graph Template $T ::= \mathbf{0} \mid p(\vec{X}) \mid X \bowtie Y \mid (T, T) \mid \nu X.T \mid x[\vec{X}]$

Variable

- ✓ Value in λ_{GT} is a **graph** in HyperLMNtal.
We allow *Constructor* and *λ -abstraction* for the atoms' names.
- ✓ λ_{GT} program is an **expression**.
- ✓ We use **Template = Value + Variables (Wildcards)** for pattern matching.

Overview of the Semantics of λ_{GT}

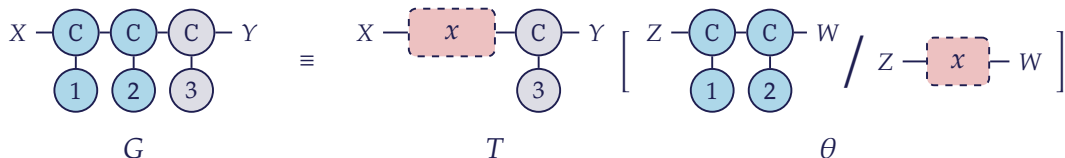
The semantics of λ_{GT} is that of a simple call-by-value λ -calculus extending its

- substitution to substitute graphs and
- case expression to pattern match graphs.

Pattern Matching in λ_{GT}

$$\frac{G \equiv T\vec{\theta}}{(\text{case } G \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3) \longrightarrow_{\text{val}} e_2\vec{\theta}} \text{ Rd-Case1}$$

For example,



The graph G can be matched with the template T with a substitution θ .

Small-Step Contextual Semantics ($\longrightarrow_{\text{val}}$) of λ_{GT}

$$\frac{G \equiv T\vec{\theta}}{(\text{case } G \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3) \longrightarrow_{\text{val}} e_2\vec{\theta}} \text{ Rd-Case1} \quad \text{Successful matching}$$

$$\frac{\neg \exists \vec{\theta}. (G \equiv T\vec{\theta})}{(\text{case } G \text{ of } T \rightarrow e_2 \mid \text{otherwise} \rightarrow e_3) \longrightarrow_{\text{val}} e_3} \text{ Rd-Case2} \quad \text{Failed matching}$$

$$\frac{fn(G) = \{\vec{X}\}}{((\lambda x[\vec{X}].e)(\vec{Y}) G) \longrightarrow_{\text{val}} e[G/x[\vec{X}]]} \text{ Rd-}\beta \quad \beta\text{-reduction}$$

$$\frac{e \longrightarrow_{\text{val}} e'}{E[e] \longrightarrow_{\text{val}} E[e']} \text{ Rd-Ctx}$$

where context $E ::= [] \mid (\text{case } E \text{ of } T \rightarrow e \mid \text{otherwise} \rightarrow e) \mid (E e) \mid (G E) \mid T$

Challenge in the Implementation

λ_{GT} has new syntax and semantics.

- We built a concise reference implementation that (we believe) reflects the semantics precisely.
- In order to implement the language correctly, we need to handle **corner cases**.

Removing the Last Element of a Difference List

```
let pop[Z] = ( $\lambda$  x[Y, X].
```

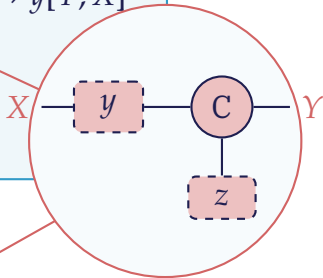
```
  case x[Y, X] of
```

```
     $\nu WZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z]) \rightarrow y[Y, X]$ 
```

```
  | otherwise  $\rightarrow$  x[Y, X]
```

```
)(Z)
```

```
in pop[Z] G
```



We need to find substitutions such that

$G \equiv \nu WZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z]) [? / y[W, X]] [? / z[Z]]$

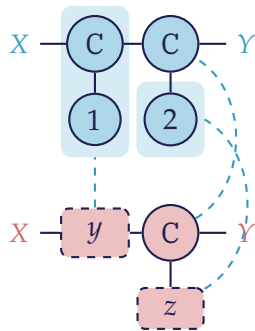
Pattern Matching of Graphs

If the graph G is $\nu WZ.(\nu U.(\text{Cons}(U, W, X), 1(U)), \text{Cons}(Z, Y, W), 2(Z))$,
then it can be matched with the template

$\nu WZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z])$ as follows.

$$\begin{aligned} & \nu WZ.(\nu U.(\text{Cons}(U, W, X), 1(U)), \text{Cons}(Z, Y, W), 2(Z)) \\ \equiv & \nu WZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z]) \\ & \quad [\nu U.(\text{Cons}(U, W, X), 1(U))/y[W, X]][2(Z)/z[Z]] \end{aligned}$$

...seems not that difficult?

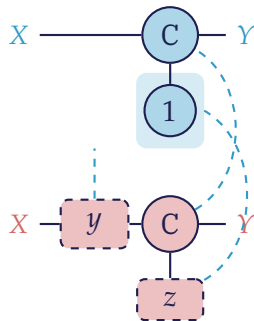


The Corner Case in Graph Pattern Matching

If the graph G is $\nu Z.(\text{Cons}(Z, Y, X), 1(Z))$, then it can NOT be matched with the template $\nu WZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z])$ as follows.

$$\begin{aligned} & \nu Z.(\text{Cons}(Z, Y, X), 1(Z)) \\ \equiv & \nu WZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z]) \\ & [\text{?} / y[W, X]][1(Z) / z[Z]] \end{aligned}$$

...fails matching?



Pattern Matching Using a Fusion

This time, we need to first *supply a fusion* using the structural congruence rule

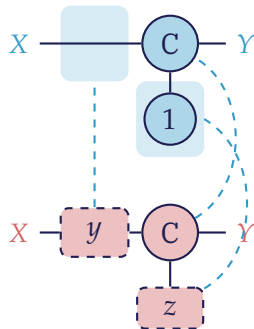
$$(E6) \nu X.(X \bowtie Y, G) \equiv \nu X.G\langle Y/X \rangle \text{ where } X \in fn(G) \vee Y \in fn(G).$$

as follows.

$$\nu Z.(\text{Cons}(Z, Y, X), 1(Z))$$

$$\equiv \nu WZ.(\boxed{W \bowtie X}, \text{Cons}(Z, Y, W), \boxed{1(Z)})$$

$$= \nu WZ.(\boxed{y[W, X]}, \text{Cons}(Z, Y, W), \boxed{z[Z]}) \\ \quad \llbracket W \bowtie X / y[W, X] \rrbracket \llbracket 1(Z) / z[Z] \rrbracket$$



Reference Interpreter

We built a reference interpreter in 500 lines of OCaml code.

- ✓ In addition to the core syntax in the previous slide, we implemented standard constructs including `let` and `let rec` expressions and arithmetic operations.
- ✓ We also implemented a visualiser using the interpreter in Elm, a purely functional language that compiles into JavaScript.

Visualiser

 λ_{GT}

Run

Proceed

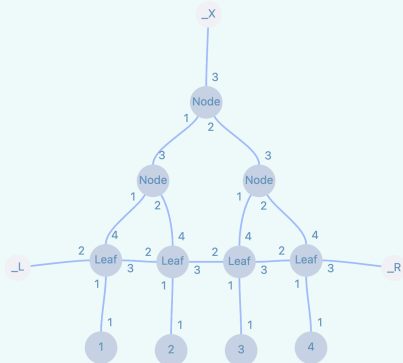
Examples ▾ About

Settings

```

Let succ[_X] x[_X] = {x[_X]} + {1(_X)} in
let map[_X] f[_X] x[_L, _R, _X] =
  let rec helper[_X] x2[_L, _R, _X] =
    case {Log} {x2[_L, _R, _X]} of
      {nu _L2 _R2 _X2 _X3. (
        y [_L, _R, _X, _L2, _R2, _X2],
        M (_L2)), Leaf (_X3, _L2, _R2, _X2), z [_X3]}
      → let z2[_X] = {f[_X]} {z[_X]} in
        {helper[_X]}
      {nu _L2 _R2 _X2 _X3 _X4. (
        y [_L, _R, _X, _L2, _R2, _X2],
        Leaf (_X3, _L2, _R2, _X2), z2 [_X3], M (_R2))}
      | otherwise → case {x2[_L, _R, _X]} of
        { y[_L, _R, _X], M (_R) } → { y[_L, _R, _X] }
        | otherwise → {Error, x2[_L, _R, _X]}
    in {helper [_X]} {x[_L, _R, _X], M (_L)}
in
{map[_X]} {succ[_X]} ({Log} {
  nu _X1 _X2 _X10. (Node (_X1, _X2, _X),
    nu _X3 _X4 _X7. (Node (_X3, _X4, _X1),
      nu _X8. (Leaf (_X8, _L, _X7, _X3), 1 (_X8)),
      nu _X9. (Leaf (_X9, _X7, _X10, _X4), 2 (_X9))
    ),
    nu _X5 _X6 _X11. (Node (_X5, _X6, _X2),
      nu _X12. (Leaf (_X12, _X10, _X11, _X5), 3 (_X12)),
      nu _X13. (Leaf (_X13, _X11, _R, _X6), 4 (_X13))
    ))})

```



Visualiser



Run

Proceed

Examples ▾

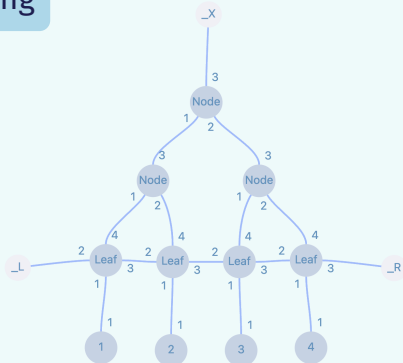
About

Settings

```
let succ[_X] _X[_X] = {x[_X]} + {1(_X)} in
```

Start running a program from the beginning

```
{(no _L2 _R2 _X2 _X3. (
  y [_L, _R, _X, _L2, _R2, _X2],
  M (_L2)), Leaf (_X3, _L2, _R2, _X2), z [_X3])
→ let z2[_X] = {f[_X]} {z[_X]} in
  {helper[_X]}
{nu _L2 _R2 _X2 _X3 _X4. (
  y [_L, _R, _X, _L2, _R2, _X2],
  Leaf (_X3, _L2, _R2, _X2), z2 [_X3], M (_R2))}
| otherwise → case {x2[_L, _R, _X]} of
  { y[_L, _R, _X], M (_R) } → { y[_L, _R, _X] }
| otherwise → {Error, x2[_L, _R, _X]}
in {helper [_X]} {x[_L, _R, _X], M (_L)}
in
{map[_X]} {succ[_X]} ({Log} {
  nu _X1 _X2 _X10. (Node (_X1, _X2, _X),
  nu _X3 _X4 _X7. (Node (_X3, _X4, _X1),
  nu _X8. (Leaf (_X8, _L, _X7, _X3), 1 (_X8)),
  nu _X9. (Leaf (_X9, _X7, _X10, _X4), 2 (_X9))
  ),
  nu _X5 _X6 _X11. (Node (_X5, _X6, _X2),
  nu _X12. (Leaf (_X12, _X10, _X11, _X5), 3 (_X12)),
  nu _X13. (Leaf (_X13, _X11, _R, _X6), 4 (_X13))
  )))
}))
```



Visualiser

λ_{GT}

Run

Proceed

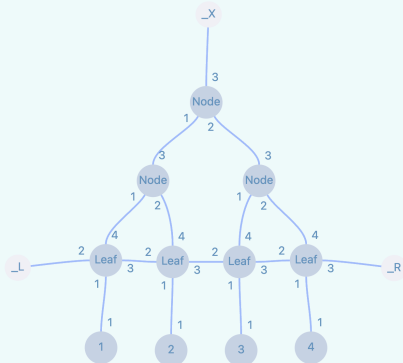
Examples ▾

About

Settings

```
let succ[_X] <[_X] = {f[_X]} + {1(_X)} in
let map[_X] <[_X] = {f[_X]} + {1(_X)} in
let rec <[_X] = {f[_X]} + {1(_X)} in
case
  {nu _L2 _R2 _X2 _X3. (
    y [_L, _R, _X, _L2, _R2, _X2],
    M (_L2)), Leaf (_X3, _L2, _R2, _X2), z [_X3]}
  → let z2[_X] = {f[_X]} {z[_X]} in
    {helper[_X]}
    {nu _L2 _R2 _X2 _X3 _X4. (
      y [_L, _R, _X, _L2, _R2, _X2],
      Leaf (_X3, _L2, _R2, _X2), z2 [_X3], M (_R2))}
  | otherwise → case {x2[_L, _R, _X]} of
    { y[_L, _R, _X], M (_R) } → { y[_L, _R, _X] }
    | otherwise → {Error, x2[_L, _R, _X]}
in {helper [_X]} {x[_L, _R, _X], M (_L)}
in
{map[_X]} {succ[_X]} ({Log} {
  nu _X1 _X2 _X10. (Node (_X1, _X2, _X),
  nu _X3 _X4 _X7. (Node (_X3, _X4, _X1),
  nu _X8. (Leaf (_X8, _L, _X7, _X3), 1 (_X8)),
  nu _X9. (Leaf (_X9, _X7, _X10, _X4), 2 (_X9))
  ),
  nu _X5 _X6 _X11. (Node (_X5, _X6, _X2),
  nu _X12. (Leaf (_X12, _X10, _X11, _X5), 3 (_X12)),
  nu _X13. (Leaf (_X13, _X11, _R, _X6), 4 (_X13))
  )))
))}
```

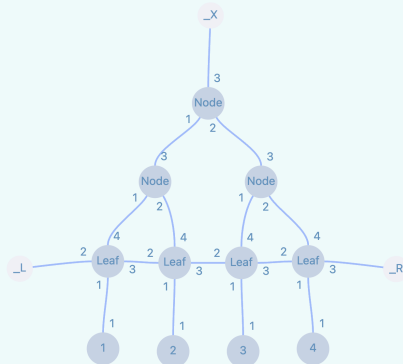
Proceed 1 step from breakpoints



```
let succ[_X] x[_X] = {x[_X]} + {1(_X)} in
let map[_X] f[_X] x[_L, _R, _X] =
  let rec helper[_X] x2[_L, _R, _X] =
    case {Log} {x2[_L, _R, _X]} of
    | nu _L2 _R2 _X2 _X3. (
      y[_L, _R, _X, _L2, _R2, _X2],
      M (_L2)), Leaf (_X3, _L2, _R2, _X2), z [_X3]}
    | otherwise → case {x2[_L, _R, _X]} of
    { y[_L, _R, _X], M (_R) } → { y[_L, _R, _X] }
    | otherwise → {Error, x2[_L, _R, _X]}
  in {helper [_X]} {x[_L, _R, _X], M (_L)}
in
{map[_X]} {succ[_X]} ({Log} {
  nu _X1 _X2 _X10. (Node (_X1, _X2, _X),
  nu _X3 _X4 _X7. (Node (_X3, _X4, _X1),
  nu _X8. (Leaf (_X8, _L, _X7, _X3), 1 (_X8)),
  nu _X9. (Leaf (_X9, _X7, _X10, _X4), 2 (_X9))
  ),
  nu _X5 _X6 _X11. (Node (_X5, _X6, _X2),
  nu _X12. (Leaf (_X12, _X10, _X11, _X5), 3 (_X12)),
  nu _X13. (Leaf (_X13, _X11, _R, _X6), 4 (_X13))
  )))
}}
```

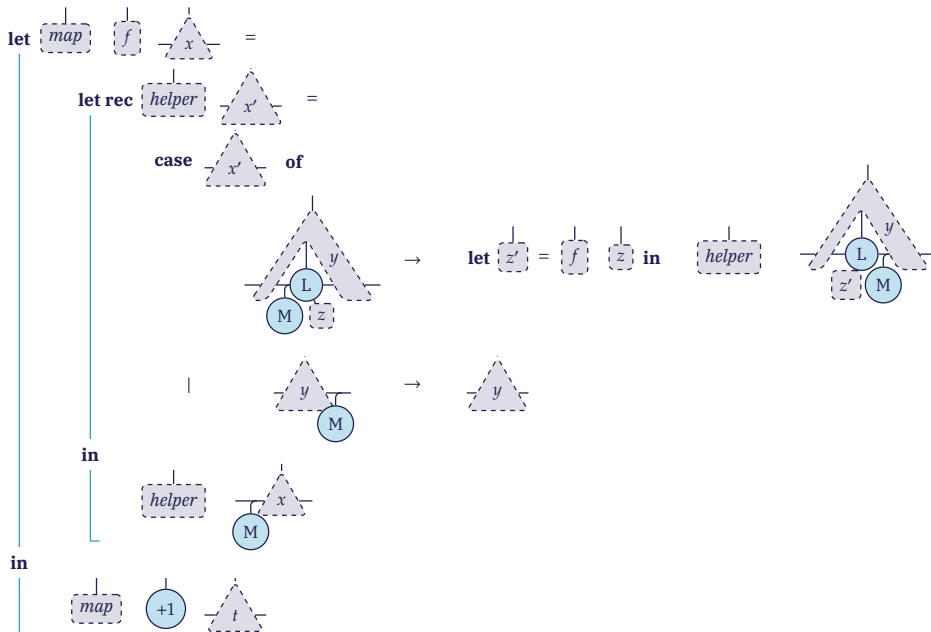
Breakpoint (identity function)

```
  y [_L, _R, _X, _L2, _R2, _X2],
  Leaf (_X3, _L2, _R2, _X2), z2 [_X3], M (_R2))}
| otherwise → case {x2[_L, _R, _X]} of
{ y[_L, _R, _X], M (_R) } → { y[_L, _R, _X] }
| otherwise → {Error, x2[_L, _R, _X]}
in {helper [_X]} {x[_L, _R, _X], M (_L)}
in
{map[_X]} {succ[_X]} ({Log} {
  nu _X1 _X2 _X10. (Node (_X1, _X2, _X),
  nu _X3 _X4 _X7. (Node (_X3, _X4, _X1),
  nu _X8. (Leaf (_X8, _L, _X7, _X3), 1 (_X8)),
  nu _X9. (Leaf (_X9, _X7, _X10, _X4), 2 (_X9))
  ),
  nu _X5 _X6 _X11. (Node (_X5, _X6, _X2),
  nu _X12. (Leaf (_X12, _X10, _X11, _X5), 3 (_X12)),
  nu _X13. (Leaf (_X13, _X11, _R, _X6), 4 (_X13))
  )))
}}
```



Visualised graph at the breakpoint

Demo Map a function on leaves of a leaf-linked tree



Related Work

There are several attempts on functional languages or term rewriting systems featuring graphs such as FUnCAL², Clean³ and Rho-Calculus⁴.

- As far as we know,
none of them generalises Algebraic Data Types to graphs as we do.
 - Handle graphs as first-class values, enable pattern matching, and let uses to define a custom data type to ensure the shape of structures.
- Enabling this required new semantics,
whose implementation is not trivial.

²Matsuda et al. “A Functional Reformulation of UnCAL Graph-Transformations: Or, Graph Transformation as Graph Reduction”. 2017.

³Brus et al. “CLEAN: A language for functional graph writing”. 1987.

⁴Cirstea et al. “Matching Power”. 2001.

Conclusion and Future Work

We implemented a new purely functional language λ_{GT} , which handles graphs as first-class data with pattern matching.

Future work:

1. We are planning to implement a type checker.
This allows us to implement more efficient matching.
2. We are going to add ownership checking to the type checker so that we can ensure safety over destructive rewriting for the efficiency.

Appendix

Implementation overview

The goal of this study is to implement as simple as possible. Our implementation consists of only 500 lines of OCaml code.

File	LOC
eval/match_ctxs.ml	79
parser/parser.mly	70
parser/lexer.mll	51
eval/syntax.ml	47
eval/eval.ml	43
eval/pushout.ml	42
eval/match_atoms.ml	36
eval/preprocess.ml	36
parser/syntax.ml	16
eval/match.ml	11
parser/parse.ml	4
bin/main.ml	3
SUM	438

The Concrete Syntax

$$\begin{aligned} e &::= \{T\} \mid \text{case } e \text{ of } \{T\} \rightarrow e \mid \text{otherwise } \rightarrow e \mid e e \mid e \text{ op } e \\ &\mid \text{let } x[\vec{X}] \ x_1[\vec{X}_1] \ \dots \ x_n[\vec{X}_n] = e \text{ in } e \quad (n \geq 0) \\ &\mid \text{let rec } x[\vec{X}] \ x_1[\vec{X}_1] \ \dots \ x_n[\vec{X}_n] = e \text{ in } e \quad (n \geq 1) \end{aligned}$$
$$\text{op} ::= + \mid * \mid - \mid < \mid =$$
$$T ::= C(\vec{X}) \mid (\backslash x[\vec{X}] . e) (\vec{Y}) \mid X > < Y \mid T, T \mid \text{nu } X . T \mid x[\vec{X}]$$
$$X ::= _ [A-Z] [a-zA-Z0-9_']^*$$
$$C ::= [A-Z] [a-zA-Z0-9_']^* \mid [1-9] [0-9]^* \mid 0$$
$$x ::= [a-z] [a-zA-Z0-9_']^*$$

Preprocessing

In the formal syntax

$vZ.(\text{Cons}(Z, Y, X), 1(Z))$

\rightarrow

Data structure in the interpreter

$[\text{Cons}(L_0, F_Y, F_X), 1(L_0)]$

Host Graph

$vWZ.(y[W, X], \text{Cons}(Z, Y, W), z[Z])$

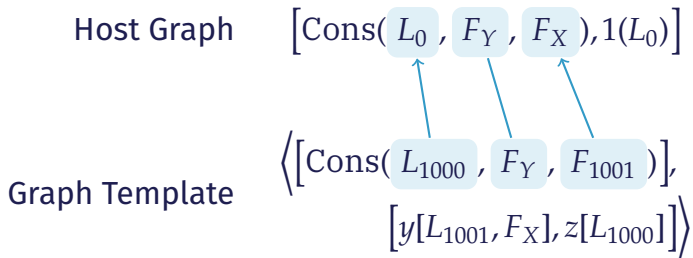
\rightarrow

$\langle [\text{Cons}(L_{1000}, F_Y, L_{1001})],$
 $[y[L_{1001}, F_X], z[L_{1000}]] \rangle$

Graph Template

where $\begin{cases} L_i & \text{Local Link} \\ F_X & \text{Free Link} \end{cases}$

Matching atoms



1. Match atoms with a mapping from the local links in the graph template to the links in the host graph.

$$\{L_{1000} \mapsto L_0, L_{1001} \mapsto F_X\}$$

2. Remove the matched atoms. Backtrack if fails.

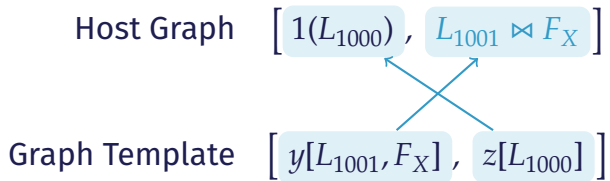
Supplying a fusion

The rest host graph $[1(L_0)] \rightarrow [1(L_{1000}), L_{1001} \bowtie F_X]$

$$\{L_{1000} \mapsto L_0, L_{1001} \mapsto F_X\}$$

1. Substitute link names in the host graph with the inverse of the obtained link mapping.
2. Supply the fusion atom $L_i \bowtie F_X$ to the host graph if there exists a mapping $L_i \mapsto F_X$

Matching graph contexts



- Finally, we obtain the graph substitution

$$[L_{1001} \bowtie F_X / y[L_{1001}, F_X], 1(L_{1000}) / z[L_{1000}]]$$

How Programming Paradigms handle data

Imperative

- ! Heaps and pointers
- × Not immutable
- × Tedious and error-prone
- × Security issues

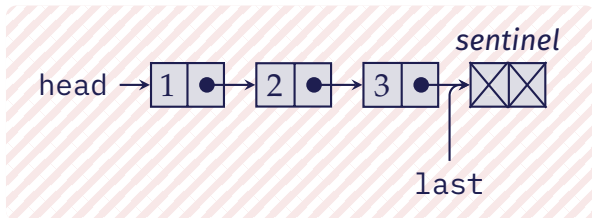
Purely Functional

- ✓ Algebraic Data Types (Trees)
- ✓ Immutable
- ✓ Pattern matching
- ✓ Type system
- × Complex data structures are difficult to handle

→ Incorporating **Graph Transformation** to purely functional languages will make them even better.

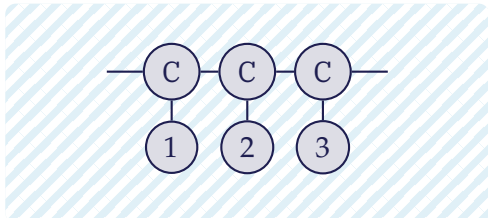
Example: Queues with Lists

Imperative



- ✗ Requires a **dummy** sentinel node.
- ✗ Low-level, tedious operations without a guarantee of the shape.

λ_{GT}



- ✓ Can define a *difference list*: a list with a link to the end.
- ✓ *Declarative operations*.
→ Next slides

Comparison between Imperative Languages with λ_{GT}

Imperative Languages

- ✓ Heaps and pointers
- ✗ Not Immutable

! **Verification techniques**

Hoare triple, Separation Logic,
Shape Analysis, ...



Proposing language λ_{GT}

- ✓ Graphs and pattern matching on them
- ✓ Immutable
- ✓ First-class functions
- ✓ **Type system**
simpler and automatic

Comparison between HyperLMNtal and Separation Logic

	Separation Logic	λ_{GT} /HyperLMNtal
Heap segment/Atom	$x \mapsto \vec{y}$	$C(\vec{X})$
Variable	x	$-$
Address/Hyperlink	$s(x)$	X
Separating Conjunction/Molecule	$*$	$,$
emp/null	emp	0
part of pure logic/fusion	$x = y$	$X \bowtie Y$
inductive predicate/non-terminal symbol	$P\vec{X}$	$\alpha(\vec{X})$
existence quantifier/hyperlink creation	\exists	ν