Type checking data structures more complex than trees

SWoPP 2022, PRO–3

July 28, 2022

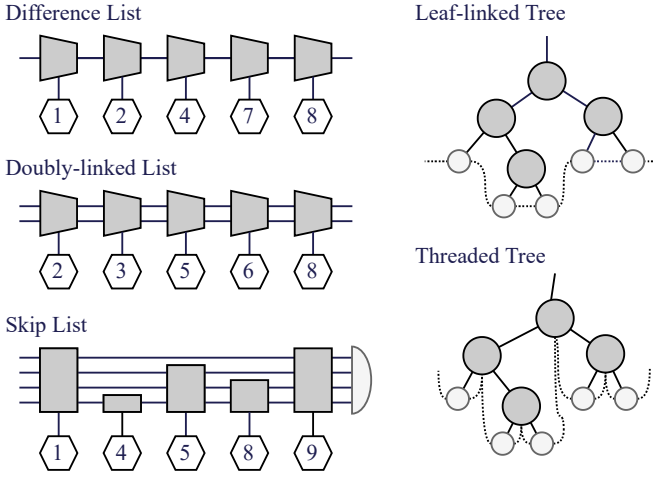
Waseda University, Tokyo, Japan

Jin SANO Naoki YAMAMOTO Kazunori UEDA

Overview

We propose a new purely functional language λ_{GT},
which handles graphs as immutable, first-class data
with pattern matching based on Graph Transformation
and developed a new type system F_{GT} for the language.

Data structures more complex than trees



There are several important data structures (graphs) that are beyond trees.

How Programming Paradigms handle data

Imperative

- ✓ Heaps and pointers
- ✗ Not Immutable

Purely Functional

- ✓ Algebraic Data Types (ADT)
- ✓ Immutable, First-class functions
- ✓ Type system
- ✗ Complex data structures are difficult to handle

Graph Transformations[Ehr+06]

- ✓ Graphs and pattern matching on them
- ✗ Not Immutable, No First-class functions

Our proposing language λ_{GT} is

a functional language with graphs as first-class data

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

Key ideas to achieve λ_{GT}

1. To establish the *semantics* of pattern matching and (re) construction of graphs, we incorporated **HyperLMNtal**[SU21]; a syntax-directed **graph transformation** formalism.
2. To *verify* the shape of the structure, we used **Graph Grammar**, which extends Tree Grammar, on which ADT is based.

1. Syntax and Semantics of λ_{GT}
2. The type system
3. Extending the type system
4. Related work and Summary

HyperLMNtal: A syntax-directed graph transformation formalism

Since graphs and their operations are more complex than trees,
there are diverse formalisms.

- Most of them use **graph isomorphism** or **bisimulation** to establish the equivalence of graphs and DPO/SPO for the matching and rewriting.
- × They are NOT syntax-directed.

In the previous study, we proposed **HyperLMNtal**[SU21].

- HyperLMNtal uses **structural congruence rules** to define the equivalence of graphs and exploit them in matching and rewriting.
- ✓ Syntax-directed → easier to adopt to the λ-calculus (defined structurally) and good for local reasoning

Syntactic conventions

For some syntactic entity E,
$\overrightarrow{E_i}^i = E_1, \ldots, E_n$ where $|\overrightarrow{E_i}^i| = n \geq 0$.

We omit the index i when there is no ambiguity.

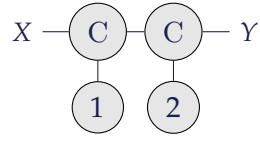
HyperLMNtal: Syntax

Graph

G	::=	$\mathbf{0}$	Null	*empty graph*
	|	$p(\vec{X})$	Atom	*vertex with name p and links $\vec{X}$*
	|	(G, G)	Molecule	*multiset of vertices*
	|	$\nu X.G$	Hyperlink Creation	*scope of link names*

For example, *Difference List* (*List Segment*) can be represented as

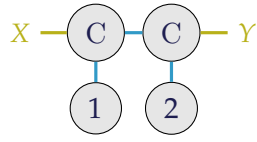
$$
\begin{aligned}
&\nu Z.(\\
&\quad \nu Z_1.(\mathrm{Cons}(Z_1, Z, X), 1(Z_1)), \\
&\quad \nu Z_2.(\mathrm{Cons}(Z_2, Y, Z), 2(Z_2)) \\
&)
\end{aligned}
$$



Free names and substitutions of hyperlinks

Links bound by ν are called *Local Links* and others are called *Free Links*

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



- $fn(G)$ denotes the set of all free links in G
- $G\langle Y/X\rangle$ replaces all free occurrences of X with Y.

The notion of locality of (link) names is NOT common in graph formalisms but in the formalisms for PLs; λ-calculus, π-calculus, ...

Structural Congruence: Axioms of graph equivalences

(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 $X \in fn(G) \vee Y \in fn(G)$		
(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 $X \notin fn(G_2)$		

For example,

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

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

✓ Notice the rules are defined compositionally.

Abbreviation schemes in HyperLMNtal

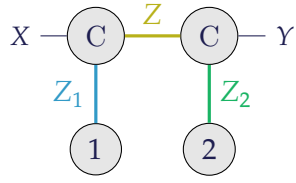
1. A nullary atom $p()$ can be simply written as p.
2. Term Notation:
   $\nu X_n.(p(\dots, X_n, \dots), q(X_1, \dots, X_n))$ can be written as $p(\dots, q(X_1, \dots, X_{n-1}), \dots)$

For example,

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

can be abbreviated as

$\mathrm{Cons}(1, \mathrm{Cons}(2, Y), X)$



Syntax of λ_{GT}

Expression	e	::=	$T \mid (\mathbf{case}\ e\ \mathbf{of}\ T \to e \mid \mathbf{otherwise} \to e) \mid (e\ e)$
Graph Template	T	::=	$\mathbf{0} \mid v(\vec{X}) \mid (T, T) \mid \nu X.T \mid x[\vec{X}]$
Atom Name	v	::=	$\bowtie \mid C \mid \lambda x[\vec{X}].e$
Value	G	::=	$\mathbf{0} \mid v(\vec{X}) \mid (G, G) \mid \nu X.G$

- ✓ λ_{GT} is designed to be a **small** language focusing on handling graphs.
- Value in λ_{GT} is a graph in HyperLMNtal
  - We allow $\bowtie$, *Constructor*, and λ-*abstraction* for the atoms' names

Syntax of λ_{GT}: Graph Template

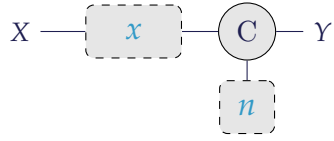
Graph Template

$$T ::= \mathbf{0} \mid v(\vec{X}) \mid (T, T) \mid \nu X.T$$
$$\mid x[\vec{X}] \quad \text{Graph context} \quad \textit{wildcard in pattern matching; variable}$$

Since the value in λ_{GT} is Graph, we use **Template** of graphs to represent data with variables.

For example,

$\nu Z.($
$\quad x[Z, X],$
$\quad \nu Z_2.(\mathrm{Cons}(Z_2, Y, Z), n[Z_2])$
$)$



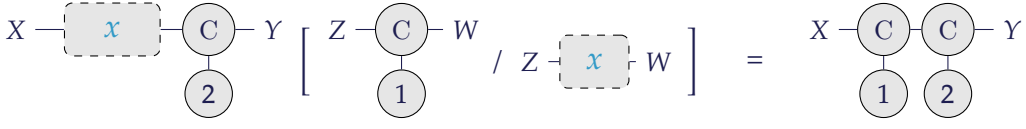
Graph Substitution

We define capture-avoiding substitution θ of a graph context $x[\vec{X}]$ with a template T in e, written $e[T/x[\vec{X}]]$.

- The definition is standard except that it handles the substitution of the free links of graph contexts as follows.

$(x[\vec{X}])[T/y[\vec{Y}]] \quad =$
$\quad$ if $x/|\vec{X}| = y/|\vec{Y}|$ then $T\langle X_1/Y_1\rangle \ldots \langle X_{|\vec{X}|}/Y_{|\vec{Y}|}\rangle$ *reconnect free links*
$\quad$ else $x[\vec{X}]$

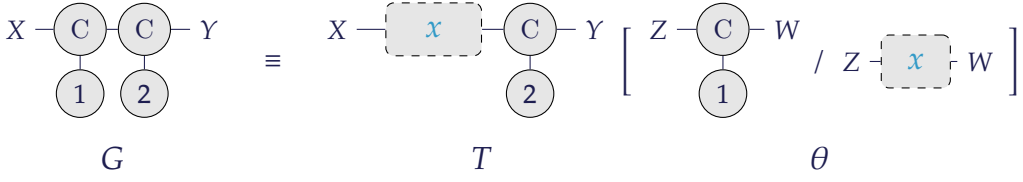
For example,



Graph Matching is defined with Graph Substitution

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

For example,



G $\quad$ T $\quad$ θ

Here, G can be matched to T with θ

Reduction of λ_{GT}

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

match succeeded

$$\frac{\neg\exists\theta.G \equiv T\theta}{(\textbf{case } G \textbf{ of } T \to e_2 \mid \textbf{otherwise} \to e_3) \longrightarrow_{\texttt{val}} e_3}\ \text{Rd-Case2}$$

match failed

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

beta reduction

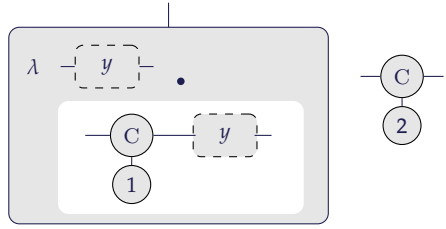
$$\frac{e_1 \longrightarrow_{\texttt{val}} e_1'}{(e_1\ e_2) \longrightarrow_{\texttt{val}} (e_1'\ e_2)}\ \text{Rd-App1} \qquad \frac{e \longrightarrow_{\texttt{val}} e'}{(G\ e) \longrightarrow_{\texttt{val}} (G\ e')}\ \text{Rd-App2} \qquad \frac{e \longrightarrow_{\texttt{val}} e'}{E[e] \longrightarrow_{\texttt{val}} E[e']}\ \text{Rd-Ctx}$$

where $E ::= [] \mid (\textbf{case } E \textbf{ of } T \to e \mid \textbf{otherwise} \to e) \mid (E\ e) \mid (G\ E) \mid T$

Example of the β-reduction

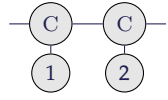
We can describe a program to append two singleton difference lists as follows.

$$(\lambda\, y[Y, X].\, \mathrm{Cons}(1, y[Y], X))(Z)\ \ \mathrm{Cons}(2, Y, X)$$



$$\longrightarrow_{\mathtt{val}}\ \mathrm{Cons}(1, y[Y], X)[\mathrm{Cons}(2, Y, X)/y[Y, X]]$$
$$=\ \ \mathrm{Cons}(1, \mathrm{Cons}(2, Y), X)$$

$\longrightarrow_{\mathtt{val}}$



1. Syntax and Semantics of λ_{GT}
2. The type system
3. Extending the type system
4. Related work and Summary

F_{GT}: Type System for λ_{GT}

We propose a new type system, F_{GT}, for the λ_{GT} language.

- The type in F_{GT} is a *type atom* $\tau\,(\vec{X})$.
  - → We extend the λ-expression to $\lambda\, x[\vec{X}]\colon \tau\,(\vec{X}).e$.
- We define the type of graphs using **graph grammar**.
- We focus on the handling of graph structures and keep the language small, e.g. No $\mathbf{let\ rec}$ nor $\mathbf{fix}$.

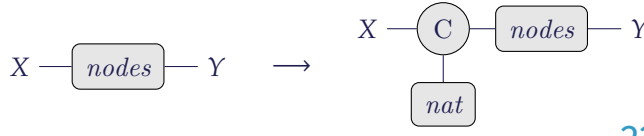
Syntax of F_{GT}: The type in F_{GT} is a type atom $\tau(\vec{X})$ where ...

Atom Name for types	τ	::=	α	Type Variable
		|	$\tau(\vec{X}) \to \tau(\vec{X})$	Arrow
Type Graph	$\mathscr{T}$	::=	$\tau(\vec{X}) \mid C(\vec{X}) \mid X \bowtie Y \mid (\mathscr{T}, \mathscr{T}) \mid \nu X.\mathscr{T}$	
Production Rule	r	::=	$\alpha(\vec{X}) \longrightarrow \mathscr{T}$	

For example, production rules of difference lists, r_1 and r_2, are

$nodes(Y, X)$
$\longrightarrow X \bowtie Y$

$nodes(Y, X)$
$\longrightarrow \mathrm{Cons}(nat, nodes(Y), X)$



Typing relation in F_{GT}

We introduce **typing environment**

$$\Gamma = \{x_1\,[\overrightarrow{X_1}] : \tau_1\,(\overrightarrow{X_1}), \ldots, x_n\,[\overrightarrow{X_n}] : \tau_n\,(\overrightarrow{X_n})\}$$

where the x_i's are mutually distinct.

The typing relation $(\Gamma, P) \vdash e : \tau\,(\vec{X})$ denotes that
e has the type $\tau\,(\vec{X})$
under the type environment Γ and a set P of production rules.

For example,

$$(\{n[Z_1] : \mathrm{nat}\,(Z_1)\}, \{r_1, r_2\}) \vdash \mathrm{Cons}(n, Y, X) : nodes\,(Y, X)$$

Rules of F_{GT} ⟨1/2⟩: typing rules as in functional languages

- These are basically the same as the type system of the other ordinary functional languages, except that **the type in $\mathbf{F_{GT}}$ is an atom**.

$$\frac{(\Gamma, P) \vdash e_1 : (\tau_1(\vec{X}) \to \tau_2(\vec{Y}))(\vec{Z}) \qquad (\Gamma, P) \vdash e_2 : \tau_1(\vec{X})}{(\Gamma, P) \vdash (e_1\ e_2) : \tau_2(\vec{Y})}\ \text{Ty-App}$$

$$\frac{((\Gamma, x[\vec{X}] : \tau_1(\vec{X})), P) \vdash e : \tau_2(\vec{Z})}{(\Gamma, P) \vdash (\lambda\, x[\vec{X}] : \tau_1(\vec{Y}).e)(\vec{W}) : (\tau_1(\vec{Y}) \to \tau_2(\vec{Z}))(\vec{W})}\ \text{Ty-Arrow}$$

$$\frac{}{(\Gamma\{x[\vec{X}] : \tau\,(\vec{Y})\}, P) \vdash x[\vec{X}] : \tau\,(\vec{Y})}\ \text{Ty-Var}$$

$$\frac{(\Gamma, P) \vdash e_1 : \tau_1(\vec{X}) \qquad ((\Gamma, \Gamma'^{*}), P) \vdash e_2 : \tau_2(\vec{Y}) \qquad (\Gamma, P) \vdash e_3 : \tau_2(\vec{Y})}{(\Gamma, P) \vdash (\mathbf{case}\ e_1\ \mathbf{of}\ T \to e_2 \mid \mathbf{otherwise} \to e_3) : \tau_2(\vec{Y})}\ \text{Ty-Case}$$

* We gave a detailed explanation of Γ' in the paper.

Rules of $F_{GT}\ \langle 2/2 \rangle$: typing rules for graphs

$$\frac{(\Gamma, P) \vdash T : \tau\,(\vec{X}) \qquad T \equiv T'}{(\Gamma, P) \vdash T' : \tau\,(\vec{X})}\ \text{Ty-Cong}$$

$$\frac{(\Gamma, P) \vdash T : \tau\,(\vec{X})}{(\Gamma, P) \vdash T\langle Z/Y\rangle : \tau\,(\vec{X})\langle Z/Y\rangle}\ \text{Ty-Alpha}$$

where $Z \notin fn(T)$

$$\frac{(\Gamma, P) \vdash T_1 : \tau_1(\overrightarrow{X_1}) \quad \dots \quad (\Gamma, P) \vdash T_n : \tau_n(\overrightarrow{X_n})}{(\Gamma, P\{\alpha(\vec{X}) \longrightarrow \mathscr{T}\}) \vdash \mathscr{T}\ [T_1/\tau_1(\overrightarrow{X_1}), \dots, T_n/\tau_n(\overrightarrow{X_n})] : \alpha(\vec{X})}\ \text{Ty-Prod}$$

where $\tau_i(\overrightarrow{X_i})$ *are all the type variable or arrow atoms appearing in* $\mathscr{T}$

Ty-Prod Example

$$\frac{(\Gamma, P) \vdash T_1 : \tau_1(\overrightarrow{X_1}) \quad \dots \quad (\Gamma, P) \vdash T_n : \tau_n(\overrightarrow{X_n})}{(\Gamma, P\{\alpha(\vec{X}) \longrightarrow \mathscr{T}\}) \vdash \mathscr{T}\,[T_1/\tau_1(\overrightarrow{X_1}), \dots, T_n/\tau_n(\overrightarrow{X_n})] : \alpha(\vec{X})}\ \text{Ty-Prod}$$

where $\tau_i(\overrightarrow{X_i})$ are all the type variable or arrow atoms appearing in $\mathscr{T}$

For example, for

$$nodes(Y, X) \longrightarrow \nu Z_1.\nu Z_2.(\mathrm{Cons}(Z_1, Z_2, X), nat(Z_1), nodes(Y, Z_2)) \quad \cdots \quad r_2$$

the Ty-Prod is

$$\frac{(\Gamma, P) \vdash T_1 : nat(Z_1) \qquad (\Gamma, P) \vdash T_2 : nodes(Y, Z_2)}{\begin{array}{l}(\Gamma, P\{P_2\}) \vdash \\ \quad \nu Z_1.\nu Z_2.(\mathrm{Cons}(Z_1, Z_2, X), nat(Z_1), nodes(Y, Z_2))[T_1/nat(Z_1), T_2/nodes(Y, Z_2)] \\ \quad = \nu Z_1.\nu Z_2.(\mathrm{Cons}(Z_1, Z_2, X), T_1, T_2) : nodes(Y, X)\end{array}}\ \text{Ty-Prod}$$

Example: Typing difference list

$$(\{n[Z_1] : \mathrm{nat}\,(Z_1)\}, \{r_1, r_2\}) \vdash \mathrm{Cons}(n, Y, X) : nodes\,(Y, X)$$

where r_1 and r_2 are the followings.

$$nodes\,(Y, X) \longrightarrow X \bowtie Y$$
$$nodes\,(Y, X) \longrightarrow \mathrm{Cons}(nat, nodes\,(Y), X)$$

can be shown as follows.

$$\dfrac{\dfrac{}{(\Gamma, P) \vdash n[Z_1] : \mathrm{n}\,(Z_1)}\ \text{Ty-Var} \quad \dfrac{\dfrac{}{(\Gamma, P\{r_1\}) \vdash X \bowtie Y : nodes\,(Y, X)}\ \text{Ty-Prod}}{(\Gamma, P) \vdash Z_2 \bowtie Y : nodes\,(Z_2, X)}\ \text{Ty-Alpha}}{\dfrac{(\Gamma, P\{r_2\}) \vdash T' : nodes\,(Y, X) \quad \text{where} \quad T' = \nu Z_1 Z_2.(\mathrm{Cons}(Z_1, Z_2, X), n[Z_1], Z_2 \bowtie Y)}{(\Gamma, P) \vdash T : nodes\,(Y, X) \quad \text{where} \quad T = \mathrm{Cons}(succ, Y, X)}\ \text{Ty-Prod} \quad T \equiv T'}\ \text{Ty-Cong}$$

Theorems of F_{GT}

We have proved some properties of F_{GT}.

Theorem 4.1 Soundness of F_{GT}.

If $(\varnothing, P) \vdash e : \tau\,(\vec{X})$, and $e \longrightarrow^{*}_{\mathtt{val}} e'$ then e' is a value or $\exists e''.e' \longrightarrow_{\mathtt{val}} e''$.

- This can be proved in a same manner as in ordinary type systems.

Theorem 4.2 Relation between F_{GT} and HyperLMNtal reduction.

Typig relation in F_{GT} corresponds to the transitive closure of HyperLMNtal reduction

- This allows us to take advantage of research of Graph Transformations[FM98; FM97; YU21; Bj21].

F_{GT} and HyperLMNtal reduction

Theorem 4.1

$$(\Gamma, P) \vdash T : \tau\,(\vec{X})$$
$$\Leftrightarrow \tau\,(\vec{X}) \rightsquigarrow_P^* T[\overrightarrow{\tau_i(\vec{Y_i})/x_i[\vec{X_i}]}^{i}]\,[\overrightarrow{\tau_i(\vec{Z_i})/(\lambda \ldots)_i(\vec{W_i})}^{i}]$$

where

- $\Gamma = \overrightarrow{x_i[\vec{X_i}] : \tau_i(\vec{X_i})}^{i}$,
- $(\lambda \ldots)_i(\vec{W_i})$ are all the λ-abstraction atoms in T, and
- $(\Gamma, P) \vdash (\lambda \ldots)_i(\vec{W_i}) : \tau_i(\vec{Z_i})$.

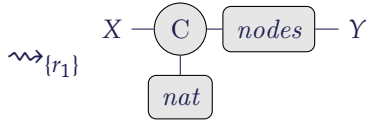
where $\rightsquigarrow_P$ is a reduction relation with rules P in HyperLMNtal.

Example: Theorem 4.1 on the typing of difference list.

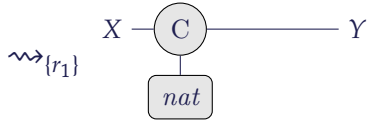
Recall that $(\{n[Z_1] : nat(Z_1)\}, \{r_1, r_2\}) \vdash \mathrm{Cons}(n, Y, X) : nodes(Y, X)$ holds in F_{GT}, which can also be shown using HyperLMNtal reduction as follows.

$nodes(Y, X)$

$\rightsquigarrow_{\{r_2\}} \mathrm{Cons}(nat, nodes(Y, Z_2), X)$



$\rightsquigarrow_{\{r_1\}} \nu Z.(\mathrm{Cons}(nat, Z, X), Z \bowtie Y)$
$\equiv \mathrm{Cons}(nat, Y, X)$
$= \mathrm{Cons}(n, Y, X)[nat(Z_1)/n[Z_1]]$



1. Syntax and Semantics of λ_{GT}
2. The type system
3. Extending the type system
4. Related work and Summary

Extending the type system

The type system was actually for **parsing** when dealing with graphs;

- it just checks if the graph can be generated from the annotated type variable atom, i.e., the start symbol.

Algebraic data types can be handled in this manner
because they only allow adding/removing a root constructor.

$\leftrightarrow$ However, in λ_{GT}, more powerful operations are possible,
for example the concatenation of difference lists.

Example: Difference list concatenation

As a running example, we consider a typed version of the program for appending two difference lists.

$$(\Gamma, P) \vdash \begin{array}{l} (\lambda\, x[Y,X] : nodes(Y,X).\\ \quad (\lambda\, y[Y,X] : nodes(Y,X).\\ \quad\quad \nu Z.(x[Z,X], y[Y,Z])\\ )(Z))(Z) \end{array} : \begin{array}{l} (nodes(Y,X) \rightarrow\\ \quad (nodes(Y,X) \rightarrow\\ \quad\quad nodes(Y,X)\\ )(Z))(Z) \end{array}$$

However, this program cannot be verified by directly using the rules in the type system.

Why typing the difference list concatenation fails

We need to prove

$$((x[Y,X] : nodes(Y,X), y[Y,X] : nodes(Y,X)), P)$$
$$\vdash \nu Z.(x[Z,X], y[Y,Z]) : nodes(Y,X)$$

to verify the present example.

Theorem 4.1 states that, if we can successfully prove the typing relation using F_{GT}, we should be able to prove

$$nodes(Y,X) \rightsquigarrow_P^* \nu Z.(nodes(Z,X), nodes(Y,Z)).$$

However, applying the production rules of difference lists cannot increase the number of $nodes$/2 atoms, contradiction.

Extending F_{GT}

We extend the previously defined F_{GT} to enable such verification.

For a graph template T
it is sufficient if the typing succeeds $T : \tau\,(\vec{X})$
after replacing each graph context in T
by all possible values of the types attached to the graph context.

Or more formally,

$$\frac{\forall\, G_i.((\overset{i}{\bigwedge}((\varnothing, P) \vdash G_i : \tau_i(\vec{Y_i}))) \Rightarrow (\varnothing, P) \vdash \overrightarrow{T[G_i/x_i[\vec{X_i}]]}^{\,i} : \tau\,(\vec{X}))}{(\overrightarrow{x_i[\vec{X_i}] : \tau_i(\vec{Y_i})}^{\,i}, P) \vdash T : \tau\,(\vec{X})}$$

Using the extension on F_{GT} in the example

In order to apply the rule to the present example,
we need to prove that,

- the substituted result of $\nu Z.(x[Z, X], y[Y, Z])$
  must have the type $nodes(Y, X)$
- for any graphs to which $x[Z, X]$ and $y[Y, Z]$ can be mapped.

that is,

$$\forall G_1, G_2.((\varnothing, P) \vdash G_1 : nodes(Y, X) \wedge (\varnothing, P) \vdash G_2 : nodes(Y, X))$$
$$\Rightarrow (\varnothing, P) \vdash \nu Z.(x[Z, X], y[Y, Z])[G_1/x[Y, X]][G_2/y[Y, X]]$$
$$= \nu Z.(G_1\langle Z/Y\rangle, G_2\langle Z/X\rangle : nodes(Y, X)).$$

Proof tree of the difference lists concatenation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{nodes}_2(Y,X) : nodes(Y,X)}}{\nu Z.(X \bowtie Z, \underline{nodes}_2(Y,Z)) : nodes(Y,X)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\underline{nat}_3(W_1) : nat(W_1)} \quad \dfrac{\dfrac{}{\nu Z.(\underline{nodes}_4(Z,X), \underline{nodes}_2(Y,Z)) : nodes(Y,X)}}{\nu Z.(\underline{nodes}_4(Z,W), \underline{nodes}_2(Y,Z)) : nodes(Y,W)}\,\text{Alpha}}{\nu W.(\text{Cons}(\underline{nat}_3, W, X), \nu Z.(\underline{nodes}_4(Z,W), \underline{nodes}_2(Y,Z))) : nodes(Y,X)}\,\text{Prod } P_2}{\nu Z.(\text{Cons}(\underline{nat}_3, \underline{nodes}_4(Z), X), \underline{nodes}_2(Y,Z)) : nodes(Y,X)}\,\text{Cong}}{\nu Z.(\underline{nodes}_1(Z,X), \underline{nodes}_2(Y,Z)) : nodes(Y,X)}\,\text{Case } \underline{nodes}_1$$

The concatenation of difference lists can be verified as shown where the arrow $\hookleftarrow$ refers to using the **induction hypothesis**.

The proposing algorithm

We have developed an algorithm that performs structural induction automatically.

- The proof obtained by the algorithm may contain **cycles** since it uses induction hypothesis.

→ We have proved that the algorithm is sound using infinite decent.

We implemented it in OCaml and tested with the examples in p3.

1. Syntax and Semantics of λ_{GT}
2. The type system
3. Extending the type system
4. Related work and Summary

Related work

Structured Gamma[FM98], Shape Types[FM97] provides a typing framework using graph grammar for graph transformation system

FUnCAL[MA17] is a functional language with Graph Transformation. The equality of graphs is defined with bisimulation. FUnCAL comes with its type system but does not support user-defined data types.

Initial algebra semantics for cyclic sharing tree structures[Ham10] discusses how to express graphs by lambda expressions.

Separation Logic[Rey02] is a verification framework for imperative programs with heaps and pointers.

Cyclic Proof, Inductive Predicate/SLRD[BGP12; IRS13; TNK19] discusses how to prove properties of heaps using induction.

Summary

We propose a new purely functional language λ_{GT},
which handles graphs as immutable, first-class data
with pattern matching based on Graph Transformation
and developed a new type system F_{GT} for the language.

1. To establish the *semantics* of pattern matching and
   (re) construction of graphs, we incorporated **HyperLMNtal**;
   a syntax-directed **graph transformation** formalism.
2. To *verify* the shape of the structure, we used **Graph Grammar**,
   which extends Tree Grammar, on which ADT is based.

References I

[FM97] Pascal Fradet and Daniel Le Métayer. “Shape types”. In: **Proc. POPL’97**. ACM. 1997, pp. 27–39. DOI: 10.1145/263699.263706.

[FM98] Pascal Fradet and Daniel Le Métayer. “Structured Gamma”. In: **Science of Computer Programming** 31.2 (1998), pp. 263–289. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(97)00023-3.

[Rey02] J.C. Reynolds. “Separation logic: a logic for shared mutable data structures”. In: **Proc. LICS 2002**. IEEE. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[Ehr+06] Hartmut Ehrig et al. **Fundamentals of Algebraic Graph Transformation**. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-31187-4. DOI: 10.1007/3-540-31188-2.

References II

[Ham10] Makoto Hamana. “Initial Algebra Semantics for Cyclic Sharing Tree Structures”. In: **Log. Methods Comput. Sci.** 6.3 (2010). URL: http://arxiv.org/abs/1007.4266.

[BGP12] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. “A Generic Cyclic Theorem Prover”. In: **APLAS**. Vol. 7705. Lecture Notes in Computer Science. Springer, 2012, pp. 350–367.

[IRS13] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. “The Tree Width of Separation Logic with Recursive Definitions”. In: **Automated Deduction – CADE-24**. Ed. by Maria Paola Bonacina. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–38. ISBN: 978-3-642-38574-2.

References III

[MA17] Kazutaka Matsuda and Kazuyuki Asada. “A Functional Reformulation of UnCAL Graph-Transformations: Or, Graph Transformation as Graph Reduction”. In: **Proc. POPL’97**. Paris, France: ACM, 2017, pp. 71–82. ISBN: 9781450347211. DOI: 10.1145/3018882.3018883. URL: https://doi.org/10.1145/3018882.3018883.

[TNK19] Makoto Tatsuta, Koji Nakazawa, and Daisuke Kimura. “Completeness of Cyclic Proofs for Symbolic Heaps with Inductive Definitions”. In: **Programming Languages and Systems**. Ed. by Anthony Widjaja Lin. Cham: Springer International Publishing, 2019, pp. 367–387. ISBN: 978-3-030-34175-6.

[Bj21] Henrik Bj’orklund et al. “Uniform parsing for hyperedge replacement grammars”. In: **J.Computer and System Sciences** 118 (2021), pp. 1–27. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2020.10.002.

References IV

[SU21] Jin Sano and Kazunori Ueda. “Syntax-driven and compositional syntax and semantics of Hypergraph Transformation System”. In: **Proc. 32nd JSSST Annual Conference (JSSST 2021)**. 2021.

[YU21] Naoki Yamamoto and Kazunori Ueda. “Engineering Grammar-based Type Checking for Graph Rewriting Languages”. In: **Proc. 12th Int. Workshop on Graph Computation Models (GCM 2021)**. 2021.

Appendix

5. Comparison with Separation Logic

6. HyperLMNtal reduction

7. Extension on F_{GT}

Comparison between Imperative Languages with λ_{GT}

Imperative Languages

- ✓ Heaps and pointers
- × Not Immutable
- ! **Verification techniques**
  Hoare triple, Separation Logic, Shape Analysis, ...

$\leftrightarrow$

Proposing language λ_{GT}

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

our contribution!

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	$\mathbf{emp}$	$\mathbf{0}$
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

Appendix

5. Comparison with Separation Logic

6. HyperLMNtal reduction

7. Extension on F_{GT}

HyperLMNtal reduction

For a set {P} of rewrite rules, the reduction relation $\rightsquigarrow_P$ on graphs is defined as the minimal relation satisfying the rules in the following.

(R1) $$\frac{G_1 \rightsquigarrow_P G_2}{(G_1, G_3) \rightsquigarrow_P (G_2, G_3)}$$

(R2) $$\frac{G_1 \rightsquigarrow_P G_2}{\nu X.G_1 \rightsquigarrow_P \nu X.G_2}$$

(R3) $$\frac{G_1 \equiv G_2 \qquad G_2 \rightsquigarrow_P G_3 \qquad G_3 \equiv G_4}{G_1 \rightsquigarrow_P G_4}$$

(R4) $$\frac{(G_1 \longrightarrow G_2) \in P}{G_1 \rightsquigarrow_P G_2}$$

Appendix

5. Comparison with Separation Logic

6. HyperLMNtal reduction

7. Extension on F_{GT}

Extension on F_{GT} in the example

In order to apply the rule to the present example,
we need to prove that,

- for any graphs to which $x[Y, X]$ and $y[Y, X]$ can be mapped,
- the substituted result must have the type $nodes(Y, X)$,

that is,

$$\forall G_1, G_2.((\varnothing, P) \vdash G_1 : nodes(Y, X) \land (\varnothing, P) \vdash G_2 : nodes(Y, X))$$
$$\Rightarrow (\varnothing, P) \vdash \nu Z.(x[Z, X], y[Y, Z])[G_1/x[Y, X]][G_2/y[Y, X]]$$
$$= \nu Z.(G_1\langle Z/Y\rangle, G_2\langle Z/X\rangle : nodes(Y, X)).$$

Abbreviation in the proof

From now on, we omit the "$(\varnothing, P) \vdash$" for brevity. Then the above can be rewritten using Ty-Alpha of F_{GT} as follows.

$$\begin{aligned} \forall G_1, G_2.(G_1 : nodes(Z, X) \wedge G_2 : nodes(Y, Z) \\ \Rightarrow \nu Z.(G_1, G_2) : nodes(Y, X)) \end{aligned} \tag{1}$$

For brevity, we denote the graph G of the type $\tau\,(\vec{X})$ as $\underline{\tau}\,(\vec{X})$ and omit $\forall G$. Then (1) can be rewritten as

$$\nu Z.(\underline{nodes}_1(Z, X), \underline{nodes}_2(Y, Z)) : nodes(Y, X)$$