# codegen development note

February 25, 2025

## 1 Notations

### 1.1 Parenthesis

We use parenthesis to represent grouping of syntax trees. This is not restricted for expressions but for any syntax trees. For example, we may use parenthesis as $\lambda(x{:}T).\, t$ for $\lambda x{:}T.\, t$. ($x{:}T$ is not an expression.)

We don't include parenthesis in syntax definition in BNF.

### 1.2 Repetition

- We use overline to represent repetition: $\overline{x_i + y_i \leq}^{1 \leq i \leq n}$ means $x_1 + y_1 < \ldots < x_n + y_n$.
- If the operator part ($\leq$) is just a punctuation to separate each term, we consider the punctuation can be added at first and/or last if appropriate for the context. $\overline{x_i \,}^{1 \leq i \leq n}$ represents "$x_1, \ldots, x_n$", "$x_1, \ldots, x_n,$", "$, x_1, \ldots, x_n$", or "$, x_1, \ldots, x_n,$".
  Thus, ($\overline{x_i \,}^{1 \leq i \leq m}\ \overline{y_i \,}^{1 \leq i \leq n}$) can represent
  $(x_1, \ldots, x_m, y_1, \ldots, y_n)$ when $(0 < m) \wedge (0 < n)$,
  $(x_1, \ldots, x_m)$ when $(0 < m) \wedge (n = 0)$,
  $(y_1, \ldots, y_n)$ when $(m = 0) \wedge (0 < n)$, and
  () when $m = n = 0$.
- We may omit the operator part when the operator is just a punctuation to separate each term. We write $\overline{x_i + y_i}^{1 \leq i \leq n}$ for $x_1 + y_1, \ldots, x_n + y_n$ if comma is appropriate separator for the context. (Also, comma can be added at first and/or end as described above.)
- We may write the range part, $1 \leq i \leq n$, as $i = 1 \ldots n$.
- We may write the range part as only an index metavariable if the range is clear from the context: $\overline{x_i + y_i}^{i}$.
- We may omit the range part ($1 \leq i \leq n$) and index of metavariables ($i$ of $x_i$ and $y_i$) when the metavariables are sequences of same length. We write $\overline{x + y \leq}$ for $x_1 + y_1 < \ldots < x_n + y_n$ when $x$ and $y$ are $n$-element sequences.
- We omit both the operator and range part if appropriate. We write $\overline{x + y}$ for $x_1 + y_1, \ldots, x_n + y_n$ if $x$ and $y$ are $n$-element sequences and comma is appropriate separator for the context. We use this form in most case.
- We use underline to distinguish metavariables which index is added by overline or not. We write $\overline{x + \underline{y}}$ for $x_1 + y, \ldots, x_n + y$.
- We use nested overline to represent multi-dimensional indexes.
  `match` $t$ `with` $\overline{C\ \overline{x} \Rightarrow u}$ `end` means
  `match` $t$ `with` $C_1\ \overline{x_1} \Rightarrow u_1 \mid \ldots \mid C_n\ \overline{x_n} \Rightarrow u_n$ `end` and
  `match` $t$ `with` $C_1\ x_{11} \ldots x_{1m_1} \Rightarrow u_1 \mid \ldots \mid C_n\ x_{n1} \ldots x_{nm_n} \Rightarrow u_n$ `end`.

- Nested overline and underline can be combined.
  $\texttt{let}\, x := \texttt{fix}\, \overline{(f := t)}\, \texttt{for}\, f\, \texttt{in}$ means
  $\texttt{let}\, x_1 := \texttt{fix}\, \overline{f := t}\, \texttt{for}\, f_1\, \texttt{in}\, \ldots\, \texttt{let}\, x_h := \texttt{fix}\, \overline{f := t}\, \texttt{for}\, f_h\, \texttt{in}$ and
  $\texttt{let}\, x_1 := \texttt{fix}\, (f_1 := t_1) \ldots (f_n := t_n)\, \texttt{for}\, f_1\, \texttt{in}\, \ldots\, \texttt{let}\, x_h := \texttt{fix}\, (f_1 := t_1) \ldots (f_n := t_n)\, \texttt{for}\, f_h\, \texttt{in}$.
  Overlines and underlines must construct a nested structure. If an underline and an overline covers same range, we consider the underline covers the overline. For example, we consider $a\overline{\underline{b}}c$ as $\overline{a(\underline{b})c} = a_1\underline{b}c_1 \ldots a_n\underline{b}c_n$. We don't consider it as $\overline{a(\underline{b})c} = a_1\overline{(\underline{b_1})}c_1 \ldots a_n\overline{(\underline{b_n})}c_n$. We cannot define how to repeat $\overline{(\underline{b})}$ because it has no variable without underline.
- This notation is taken from [1].

## 1.3 Number of Elements

We use $|x|$ to represent the number of elements: $|x| = n$ if $x$ is an $n$-element sequence, $x_1, \ldots, x_n$.

## 1.4 Number of Arguments

- $\mathrm{NA}_t$ is the number of arguments of $t$: $\mathrm{NA}_t = m$ if $t : T_1 \to \cdots \to T_m \to T_0$ and $T_0$ is an inductive type.
- $\mathrm{NP}_I$ is the number of the parameters of the inductive type $I$:
  $\mathrm{NP}_I = p$ if $I$ is defined in an inductive definition $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$.
- $\mathrm{NI}_I$ is the number of the indexes of the inductive type $I$ (the number of arguments without the parameters for the inductive type):
  $\mathrm{NI}_I = |\Gamma_{\mathrm{Arr}(I)}|$ where $\Gamma_{\mathrm{Arr}(t)}$ is the arity of the inductive type $t$. It means $(I : \forall(\Gamma_P; \Gamma_{\mathrm{Arr}(t)}), S)$ is defined in $\Gamma_I$ of $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$ in the global environment where $|\Gamma_P| = p$ and $S$ is a sort.
- $\mathrm{NM}_C$ is the number of the members of the constructor $C$ (the number of arguments without the parameters for the inductive type):
  $\mathrm{NM}_C = |\Gamma|$ where $\Gamma$ is the non-parameter arguments of the constructor $C$. It means $(C : \forall(\Gamma_P; \Gamma), T)$ is defined in $\Gamma_C$ of $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$ in the global environment where $|\Gamma_P| = p$ and $T$ is an inductive type.

## 1.5 Substitution

$t\{x/u\}$ means a term in which variable $x$ in term $t$ is replaced by term $u$. This notation is taken from the Coq reference manual [2].

We use $t\{\overline{x/u}\}$ for parallel substituion.

# 2 Gallina

## 2.1 Gallina Syntax

| | |
|---|---|
| $t = x$ | v-variable |
| $\mid f$ | f-variable |
| $\mid c$ | constant |
| $\mid C$ | constructor |
| $\mid T$ | type |
| $\mid \lambda x{:}T.\, t$ | abstraction |
| $\mid t\ u$ | application |
| $\mid \texttt{let}\ x := t : T\ \texttt{in}\ u$ | let-in |
| $\mid \texttt{match}\ t\ \texttt{with}\ \overline{C\ \overline{x{:}T} \Rightarrow u \mid}\ \texttt{end}$ | conditional |
| $\mid \texttt{fix}\ \overline{f/k{:}T := t\ \underline{\texttt{with}}}\ \texttt{for}\ f_j$ | fixpoint |

Note:

- $u, a, b$ represents a term as $t$.
  $v, w, y, z$ represent a v-variable as $x$.
  $g$ represent a f-variable as $f$.
  $U, V$ represents a type as $T$.
- We distinguish v-variables (such as $x$) and f-variables (such as $f$) in the syntax. V-variables are variables bounded by abstraction, let-in, and conditional. F-variables are variables bounded by fixpoint. They are treated differently in C code generation: V-variables has corresponding C variables but F-variables has no corresponding C variables.
- We write $(\cdots((t\ u_1)\ u_2)\cdots\ u_n)$ as $t\ u_1 \ldots u_n$ or $t\ \overline{u}$.
- We write $\lambda x_1{:}T_1.\, (\ldots(\lambda x_n{:}T_n.\, u)\ldots)$ as $\lambda\overline{x{:}T}.\, u$.
- We write $\texttt{let}\ x_1 := t_1{:}T_1\ \texttt{in}\ (\ldots(\texttt{let}\ x_n := t_n{:}T_n\ \texttt{in}\ u)\ldots)$ as $\texttt{let}\ \overline{x := t{:}T}\ \texttt{in}\ u$.
- $k$ is an integer.
  $k_i$ for fixpoint specify the decreasing argument for $f_i$.
- If it is unambiguous, we omit type annotations for the sake of simplicity. We also omit $k_i$ in fixpoints if they are not used.
- We omitted the elimination predicate (`as-in-return` clause of `match`-expression) in the syntax.
- We omitted the dummy parameters (underscores between $C$ and $\overline{x{:}T}$) in conditionals.
- We consider inductive types and constructor types has no let-in in binders.
- We omitted the detail of the types. Actual Gallina permits any Gallina term which evaluates to a type.

## 2.2 Global Context and Local Context

$E$ is a global environment which is a list of global assumptions ($c{:}T$), global definitions ($c := t{:}T$), and inductive definitions ($\mathsf{Ind}\ [p]\ (\Gamma_I := \Gamma_C)$).

$\Gamma$ is a local context which is a list of local assumptions ($x{:}T$), ($f{:}T$) and local definitions ($x := t{:}T$). The local assumptions ($x{:}T$) represent v-variables bounded by outer abstractions and conditionals. The local assumptions ($f{:}T$) represent f-variables bounded by outer fixpoints. The local definitions represent variables bounded by outer let-in.

## 2.3 Gallina Conversion Rules

beta: $E[\Gamma] \vdash ((\lambda x.\, t)\ u) \triangleright t\{x/u\}$

delta-local: $$\frac{(x := t) \in \Gamma}{E[\Gamma] \vdash x \triangleright t}$$

delta-global: $$\frac{(c := t) \in E}{E[\Gamma] \vdash c \triangleright t}$$

zeta: $E[\Gamma] \vdash \texttt{let}\ x := t\ \texttt{in}\ u \triangleright u\{x/t\}$

iota-match: $$\frac{E[\Gamma] \vdash C_j\ \overline{a}\ \overline{b} : T \quad |a| = \mathrm{NP}_T}{E[\Gamma] \vdash \texttt{match}\ (C_j\ \overline{a}\ \overline{b})\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end} \triangleright (\lambda \overline{x_j}.\, t_j)\ \overline{b}}$$

iota-fix: $$\frac{u_{k_j} = C\ \overline{a} \quad |u| = k_j}{E[\Gamma] \vdash\ (\texttt{fix}\ \overline{f/k := t}\ \texttt{for}\ f_j)\ \overline{u} \triangleright\ t_j\{\overline{f/\texttt{fix}\ \underline{(\overline{f/k := t})}\ \texttt{for}\ f}\}\ \overline{u}}$$

eta expansion: $$\frac{E[\Gamma] \vdash t : \forall x{:}T.\, U}{E[\Gamma] \vdash t \triangleright \lambda x{:}T.\, (t\ x)}$$

Note:

- The rules shown here are reductions, except the eta expansion.
- Variables cannot conflict because Coq uses de Bruijn's indexes to represent variables.
- If it is unambiguous, we omit type annotations in these definitions for the sake of simplicity.
- Iota-match reduces $\texttt{match}\ @\texttt{cons}\ \texttt{nat}\ \texttt{1}\ \texttt{nil}\ \texttt{with}\ (\texttt{nil} \Rightarrow t_1)\ |\ (\texttt{cons}\ \texttt{h}\ \texttt{t} \Rightarrow t_2)\ \texttt{end}$ to $(\lambda \texttt{h}.\, \lambda \texttt{t}.\, t_2)\ \texttt{1}\ \texttt{nil}$ because `list` has one parameter ($\mathrm{NP}_{\texttt{nat}} = 1$) and `cons` has two members ($\mathrm{NM}_{\texttt{cons}} = 2$).

## 2.4 Free Variables

$\mathrm{FV}(t)$ is the free variables of $t$.

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(f) &= \{f\} \\
\mathrm{FV}(c) &= \varnothing \\
\mathrm{FV}(C) &= \varnothing \\
\mathrm{FV}(\lambda x{:}T.\, t) &= \mathrm{FV}(t) - \{x\} \\
\mathrm{FV}(t\ u) &= \mathrm{FV}(t) \cup \mathrm{FV}(u) \\
\mathrm{FV}(\texttt{let}\ x := t : T\ \texttt{in}\ u) &= \mathrm{FV}(t) \cup \mathrm{FV}(u) - \{x\} \\
\mathrm{FV}(\texttt{match}\ t\ \texttt{with}\ \overline{C\ \overline{x{:}T} \Rightarrow u}\ \texttt{end}) &= \mathrm{FV}(t) \cup \textstyle\bigcup_i (\mathrm{FV}(u_i) - \{\overline{x_i}\}) \\
\mathrm{FV}(\texttt{fix}\ \overline{f/k{:}T := t}\ \texttt{for}\ f_j) &= \left(\textstyle\bigcup_i \mathrm{FV}(t_i)\right) - \{\overline{f}\}
\end{aligned}$$

Note:

- We ommitted $\mathrm{FV}(T)$ here because it needs details of types. It is defined as usual.

## 2.5 Syntactic Context

We use syntactic context $K$. $K$ is a single-hole context: a Gallina term with a subterm is sustituted with a hole, $[]$. $K[u]$ is $K$ with the hole is substituted with $u$.

We call syntactic context just as context if it is not ambiguous.

$$
\begin{aligned}
K = {} & [] \\
| {} & \lambda x{:}T.\ K \\
| {} & K\ t \\
| {} & t\ K \\
| {} & \texttt{let}\ x := K : T\ \texttt{in}\ t \\
| {} & \texttt{let}\ x := t : T\ \texttt{in}\ K \\
| {} & \texttt{match}\ K\ \texttt{with}\ \overline{C\ \overline{x{:}T} \Rightarrow u}\ \texttt{end} \\
| {} & \texttt{match}\ t\ \texttt{with}\ \overline{(C_l\ \overline{x_l{:}T_l} \Rightarrow u_l)}^{1 \le l < i}\ (C_i\ \overline{x_i{:}T_i} \Rightarrow K)\ \overline{(C_l\ \overline{x_l{:}T_l} \Rightarrow u_l)}^{i < l \le h}\ \texttt{end} \\
| {} & \texttt{fix}\ \overline{(f_l/k_l{:}T_l := t_l)}^{1 \le l < i}\ (f_i/k_i{:}T_i := K)\ \overline{(f_l/k_l{:}T_l := t_l)}^{i < l \le h}\ \texttt{for}\ f_j
\end{aligned}
$$

### 2.6 Local Context of Syntactic Context

$\mathrm{LC}(K)$ is the local context of the hole of $K$.

$$
\begin{aligned}
\mathrm{LC}([]) &= \text{empty} \\
\mathrm{LC}(\lambda x{:}T.\ K) &= (x{:}T); \mathrm{LC}(K) \\
\mathrm{LC}(K\ t) &= \mathrm{LC}(K) \\
\mathrm{LC}(t\ K) &= \mathrm{LC}(K) \\
\mathrm{LC}(\texttt{let}\ x := K : T\ \texttt{in}\ t) &= \mathrm{LC}(K) \\
\mathrm{LC}(\texttt{let}\ x := t : T\ \texttt{in}\ K) &= (x := t{:}T); \mathrm{LC}(K) \\
\mathrm{LC}(\texttt{match}\ K\ \texttt{with}\ \overline{C\ \overline{x{:}T} \Rightarrow u}\ \texttt{end}) &= \mathrm{LC}(K) \\
\mathrm{LC}(\texttt{match}\ t\ \texttt{with}\ \overline{(C_l\ \overline{x_l{:}T_l} \Rightarrow u_l)}^{1 \le l < i}\ (C_i\ \overline{x_i{:}T_i} \Rightarrow K)\ \overline{(C_l\ \overline{x_l{:}T_l} \Rightarrow u_l)}^{i < l \le h}\ \texttt{end}) &= \overline{(x_i{:}T_i)}; \mathrm{LC}(K) \\
\mathrm{LC}(\texttt{fix}\ \overline{(f_l/k_l{:}T_l := t_l)}^{1 \le l < i}\ (f_i/k_i{:}T_i := K)\ \overline{(f_l/k_l{:}T_l := t_l)}^{i < l \le h}\ \texttt{for}\ f_j) &= \overline{(f_i{:}T_i)}; \mathrm{LC}(K)
\end{aligned}
$$

$\mathrm{KV}(K)$ is the bound variables usable in the hole of the context $K$.

$$\mathrm{KV}(K) = \{\, x \mid (\exists T.(x{:}T) \in \mathrm{LC}(K)) \,\} \cup \{\, f \mid (\exists T.(f{:}T) \in \mathrm{LC}(K)) \,\} \cup \{\, x \mid (\exists t, T.(x := t{:}T) \in \mathrm{LC}(K)) \,\}$$

# 3 CodeGen

- Gallina-to-Gallina Transformations
  - Inlining
  - Strip Cast
  - Eta Expansion for Functions
  - V-Normalization
  - S-Normalization
  - Type Normalization
  - Static Argument Normalization
  - Unused let-in Deletion
  - Call Site Replacement
  - Eta Reduction to Expose Fixpoint
  - Argument Completion
  - Unreachable Fixfunc Deletion

  - Monomorphism Check
  - Borrow Check
  - C Variable Allocation
- C Code Generation
  - C Code Generation

# 4 Gallina-to-Gallina Transformations

We define transformations as a judgement $E[\Gamma] \vdash t \triangleright u$. This means a subterm $t$ is substituted to $u$ where $E$ and $\Gamma$ are the global environment and the local context of them.

We also use $E[\Gamma] \vdash_K t \triangleright u$ to represent transformations restricted with a syntactical context $K$.

$E[\Gamma] \vdash_K t \triangleright u$ is similar to $E[\Gamma] \vdash K[t] \triangleright K[u]$ but $\Gamma$ is the local context of $t$ (not $K[t]$).

Also, we use $E[] \vdash_\$ t \triangleright u$ which defines a tranformation of an entire term (not subterm). (The local context is empty because an entire term has no local context.)

When we define a new constant in a transformation, We use $E[\Gamma] \vdash t \triangleright (E; (c := a{:}T))[\Gamma] \vdash u$.

## 4.1 Inlining

Codegen apply delta-global reductions to inline definitions.

Two command, `CodeGen GlobalInline` and `CodeGen LocalInline`, specifies what definitions will be inlined.

```
CodeGen GlobalInline QUALID...
CodeGen LocalInline QUALID : QUALID...
```

`CodeGen GlobalInline` $c_1 \dots c_n$ specifies global constants $c_1 \dots c_n$ will be expanded.

`CodeGen LocalInline` $c_0$ `:` $c_1 \dots c_n$ specifies global constants $c_1 \dots c_n$ will be expanded in $c_0$.

## 4.2 Strip Cast

Codegen removes cast expressions. For example, `(1 : nat) + 2` is transformed to `1 + 2`.

Note that we ignore casts in this document except this section. Even the Gallina syntax in Section 2.1 has no rule for casts.

## 4.3 Eta Expansion for Functions

We apply eta-expansion to functions of top-level functions, fix-bounded functions, and closure generating lambdas. We consider explicit lambdas are closure generation. This makes beta-var applicable for partial applications without worrying to expose computation. (CIC [2] uses lambdas for match-branches but our syntax uses no lambdas for them. Thus match-branches doesn't trigger the eta expansion.)

$$\text{etaex-top: } \frac{E[] \vdash t : \forall x{:}T.\, U \quad t \text{ is neither an abstraction nor fixpoint}}{E[] \vdash_\$ t \triangleright \lambda x{:}T.\, (t\ x)}$$

$$\text{etaex-fix: } \frac{\begin{array}{l} E[\Gamma] \vdash t_i : \forall x{:}T.\, U \quad t_i \text{ is neither an abstraction nor fixpoint} \\ K = \texttt{fix}\ \overline{(f_l := t_l)}^{1 \le l < i} (f_i := []) \overline{(f_l := t_l)}^{i < l \le h}\ \texttt{for}\ f_j \end{array}}{E[\Gamma] \vdash_K t_i \triangleright \lambda x{:}T.\, (t_i\ x)}$$

$$\text{etaex-abs: } \frac{\begin{array}{l} E[\Gamma] \vdash t : \forall x{:}T.\, U \quad t \text{ is neither an abstraction nor fixpoint} \\ K = \lambda y.\, [] \end{array}}{E[\Gamma] \vdash_K t \triangleright \lambda x{:}T.\, (t\ x)}$$

This transformations makes a term in following syntax. The entire term is represented as $t^{\mathrm{DL}}$. ($t^{\mathrm{D}}$ for functions and $t^{\mathrm{E}}$ for non-function constants.)

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda x{:}T.\, t^{\mathrm{DL}} \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j \\
t^{\mathrm{DL}} &= t^{\mathrm{D}} \\
&\mid t^{\mathrm{E}} && \text{the type of } t^{\mathrm{E}} \text{ is an inductive type} \\
t^{\mathrm{E}} &= x \\
&\mid f \\
&\mid c \\
&\mid C \\
&\mid T \\
&\mid t^{\mathrm{E}}\ t^{\mathrm{E}} \\
&\mid \texttt{let}\ x := t^{\mathrm{E}} : T\ \texttt{in}\ t^{\mathrm{E}} \\
&\mid \texttt{match}\ t^{\mathrm{E}}\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{E}}}\ \texttt{end} \\
&\mid t^{\mathrm{D}}
\end{aligned}
$$

The type of $t^{\mathrm{E}}$ in $t^{\mathrm{DL}}$ is an inductive type because the eta expansions (etaex-fix and etaex-abs) transform the body of abstraction and fixpoint until its type is not function type.

## 4.4 V-Normalization

### 4.4.1 V-Reductions

$$\text{zeta-arg: } \frac{E[\Gamma] \vdash u : U \quad t \text{ is not an application} \quad u \text{ is not a v-variable} \quad y \text{ is a fresh v-variable}}{E[\Gamma] \vdash t\ \overline{x}\ u\ \overline{a} \triangleright \texttt{let}\ y := u : U\ \texttt{in}\ t\ \overline{x}\ y\ \overline{a}}$$

$$\text{zeta-item: } \frac{E[\Gamma] \vdash u : U \quad u \text{ is not a v-variable} \quad y \text{ is a fresh v-variable}}{E[\Gamma] \vdash \texttt{match}\ u\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end} \triangleright \texttt{let}\ y := u : U\ \texttt{in}\ \texttt{match}\ y\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end}}$$

### 4.4.2 V-Normal Form

V-normal form restricts Gallina terms that (1) application arguments and (2) match items to variables.

$$
\begin{aligned}
t = x &\mid f \mid c \mid C \mid T \mid \lambda x{:}T.\, t \mid \texttt{let}\ x := t : T\ \texttt{in}\ u \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t}\ \texttt{for}\ f_j \\
&\mid t\ x && \leftarrow (1) \\
&\mid \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end} && \leftarrow (2)
\end{aligned}
$$

Since we apply V-reductions for a eta-expanded term, the result term can be represented in following syntax.

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda x{:}T.\, t^{\mathrm{DL}} \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j \\
t^{\mathrm{DL}} &= t^{\mathrm{D}} \\
&\mid t^{\mathrm{E}} && \text{the type of } t^{\mathrm{E}} \text{ is an inductive type} \\
t^{\mathrm{E}} &= x \\
&\mid f \\
&\mid c \\
&\mid C \\
&\mid T \\
&\mid t^{\mathrm{E}}\ x && \leftarrow (1) \\
&\mid \texttt{let}\ x := t^{\mathrm{E}} : T\ \texttt{in}\ t^{\mathrm{E}} \\
&\mid \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{E}}}\ \texttt{end} && \leftarrow (2) \\
&\mid t^{\mathrm{D}}
\end{aligned}
$$

## 4.5 S-Normalization

### 4.5.1 S-Reductions

beta-var: $E[\Gamma] \vdash (\lambda x.\, t)\ y \triangleright t\{x/y\}$

delta-vvar: $\dfrac{(x := y) \in \Gamma}{E[\Gamma] \vdash x \triangleright y}$

delta-fvar: $\dfrac{(x := f) \in \Gamma \quad x \text{ occur at non-argument position}}{E[\Gamma] \vdash x \triangleright f}$

delta-fun: $\dfrac{0 \le |x| \quad 0 < |y| \quad (z := t\ \overline{x}) \in \Gamma \quad t \text{ is one of v-variable, f-variable, constant, constructor, abstraction, or fixpoint}}{E[\Gamma] \vdash z\ \overline{y} \triangleright t\ \overline{x}\ \overline{y}}$

zeta-flat: $E[\Gamma] \vdash \texttt{let}\ y := (\texttt{let}\ x := t_1\ \texttt{in}\ t_2)\ \texttt{in}\ t_0 \triangleright \texttt{let}\ x := t_1\ \texttt{in}\ (\texttt{let}\ y := t_2\ \texttt{in}\ t_0)$

zeta-app: $E[\Gamma] \vdash (\texttt{let}\ y := t\ \texttt{in}\ u)\ \overline{x} \triangleright \texttt{let}\ y := t\ \texttt{in}\ (u\ \overline{x})$

iota-match-var: $\dfrac{(v := C_j\ \overline{y}\ \overline{z} : T) \in \Gamma \quad |y| = \mathrm{NP}_T}{E[\Gamma] \vdash \texttt{match}\ v\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end} \triangleright (\lambda \overline{x_j}.\, t_j)\ \overline{z}}$

iota-fix-var: $\dfrac{(x_{k_j} := C\ \overline{y}) \in \Gamma \quad \overline{z} \text{ are fresh v-variables} \quad E[\Gamma] \vdash (\texttt{fix}\ \overline{f/k := t}\ \texttt{for}\ f_j)\ \overline{x} : T \quad T \text{ is an inductive type}}{E[\Gamma] \vdash (\texttt{fix}\ \overline{f/k := t}\ \texttt{for}\ f_j)\ \overline{x} \triangleright \texttt{let}\ z := \texttt{fix}\ \underline{(\overline{f/k := t})}\ \texttt{for}\ f\ \texttt{in}\ t_j\{\overline{f/z}\}\ \overline{x}}$

iota-fix-var': $\dfrac{(x_{k_j} := C\ \overline{y}) \in \Gamma \quad \overline{(z := \texttt{fix}\ \underline{(\overline{f/k := t})}\ \texttt{for}\ f) \in \underline{\Gamma}} \quad E[\Gamma] \vdash (\texttt{fix}\ \overline{f/k := t}\ \texttt{for}\ f_j)\ \overline{x} : T \quad T \text{ is an inductive type}}{E[\Gamma] \vdash (\texttt{fix}\ \overline{f/k := t}\ \texttt{for}\ f_j)\ \overline{x} \triangleright t_j\{\overline{f/z}\}\ \overline{x}}$

match-app: $\dfrac{E[\Gamma] \vdash z : T}{E[\Gamma] \vdash \texttt{match}\ x\ \texttt{as}\ x'\ \texttt{in}\ I\ \overline{y}\ \texttt{return}\ T \to P\ \overline{y}\ x'\texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end}\ z \triangleright \texttt{match}\ x\ \texttt{as}\ x'\ \texttt{in}\ I\ \overline{y}\ \texttt{return}\ P\ \overline{y}\ x'\texttt{with}\ \overline{C\ \overline{x} \Rightarrow t\ \underline{z}}\ \texttt{end}}$

Note:

- match-app is not convertible
- delta-fvar does not forbid at a match item position but it does not break V-normalization. It is because f-variable, which is always function type, cannot occur at match item, which must be inductive type.

### 4.5.2 S-Normal Form

S-reductions transform applications to restrict function positions.

- beta-var removes an abstraction at the function position of an application.
- zeta-app removes a let-in at the function position of an application.
- match-app removes a conditional at the function position of an application.

Also, types cannot be a function. We treat multi-arguments application as single application, application is not occur at a function position. Thus, function position can be v-variable, f-variable, constant, construcor, or fixpoint in the S-normal form.

Also, zeta-flat removes a let-in at the binder term of a let-in.

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda x{:}T.\, t^{\mathrm{DL}} \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j \\
t^{\mathrm{DL}} &= t^{\mathrm{D}} \\
&\mid t^{\mathrm{L}} && \text{the type of } t^{\mathrm{L}} \text{ is an inductive type} \\
t^{\mathrm{L}} &= \texttt{let}\ \overline{x := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}} \\
t^{\mathrm{M}} &= \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} \\
&\mid t^{\mathrm{E}} \\
t^{\mathrm{E}} &= x \mid f \mid c \mid C \mid T \\
&\mid t^{\mathrm{F}}\ \overline{x} && 0 < |x| \\
&\mid t^{\mathrm{D}} \\
t^{\mathrm{F}} &= x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j
\end{aligned}
$$

## 4.6 Type Normalization

We normalize type annotations in the term.

This transformation makes that types contain no variable bounded by let-ins because such variables are redex of delta reduction. Thus, this transformation makes Unused let-in Deletion (Section 4.8) more effective.

$$
\begin{array}{rll}
t^{\mathrm{D}} = \lambda x{:}\boxed{T}.\, t^{\mathrm{DL}} & & \leftarrow \\
\quad \mid \texttt{fix}\ \overline{f/k{:}\boxed{T} := t^{\mathrm{D}}}\ \texttt{for}\ f_j & & \leftarrow \\
t^{\mathrm{DL}} = t^{\mathrm{D}} & & \\
\quad \mid t^{\mathrm{L}} & \text{the type of } t^{\mathrm{L}} \text{ is an inductive type} & \\
t^{\mathrm{L}} = \texttt{let}\ \overline{x := t^{\mathrm{M}} : \boxed{T}}\ \texttt{in}\ t^{\mathrm{M}} & & \leftarrow \\
t^{\mathrm{M}} = \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} & & \\
\quad \mid t^{\mathrm{E}} & & \\
t^{\mathrm{E}} = x \mid f \mid c \mid C \mid T & & \\
\quad \mid t^{\mathrm{F}}\ \overline{x} & 0 < |x| & \\
\quad \mid t^{\mathrm{D}} & & \\
t^{\mathrm{F}} = x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{f/k{:}\boxed{T} := t^{\mathrm{D}}}\ \texttt{for}\ f_j & & \leftarrow
\end{array}
$$

We also normalize types in `match`-expressions to make less free variables. Gallina internal representation of `match`-expressions contains parameters for the inductive type, return clause, and SProp inversion data.

## 4.7 Static Argument Normalization

We normalize static arguments. We assume the normalized static arguments have no free variables.

It makes the syntax as follows.

$$
\begin{array}{rll}
t^{\mathrm{D}} = \lambda x{:}T.\, t^{\mathrm{DL}} & & \\
\quad \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j & & \\
t^{\mathrm{DL}} = t^{\mathrm{D}} & & \\
\quad \mid t^{\mathrm{L}} & \text{the type of } t^{\mathrm{L}} \text{ is an inductive type} & \\
t^{\mathrm{L}} = \texttt{let}\ \overline{x := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}} & & \\
t^{\mathrm{M}} = \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} & & \\
\quad \mid t^{\mathrm{E}} & & \\
t^{\mathrm{E}} = x \mid f \mid c \mid C \mid T & & \\
\quad \mid t^{\mathrm{F}}\ \overline{x} & 0 < |x| & \\
\quad \mid t^{\mathrm{C}}\ \overline{t^{\mathrm{A}}} & 0 < |t^{\mathrm{A}}| & \\
\quad \mid t^{\mathrm{D}} & & \\
t^{\mathrm{F}} = x \mid f \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j & & \\
t^{\mathrm{C}} = c \mid C & & \\
t^{\mathrm{A}} = x \mid u & u \text{ is a static argument (normal Gallina term without FV)} &
\end{array}
$$

The application in previous section, $t^{\mathrm{F}}\ \overline{x}$, is changed to $t^{\mathrm{C}}\ \overline{t^{\mathrm{A}}}$ for constant and constructor applications. (This is not V-normal form because $t^{\mathrm{A}}$ can be non-variable.)

Static arguments are defined as follows by default.

- non-monomorphic arguments for constant functions. (The non-monomorphic argument means an argument which type is a sort or a polymorphic function type.)
- parameters for constructors.

The static arguments can be customized with `CodeGen StaticArgs` command.

This transformation makes that static arguments contain no variable bounded by let-ins because such variables are redex of delta reduction. Thus, this transformation makes Unused let-in Deletion (Section 4.8) more effective.

## 4.8 Unused let-in Deletion

$$\text{zeta-del: } \frac{x \text{ does not occur in } u \quad x \text{ is not linear} \quad \mathrm{FV}(t) \text{ does not contain linear variable}}{E[\Gamma] \vdash \texttt{let}\ x := t\ \texttt{in}\ u \triangleright u}$$

## 4.9 Call Site Replacement

$$\text{replace: } \frac{\begin{array}{c} t \text{ is a constant or constructor} \\ a_{|a|} \text{ is not a v-variable} \\ \overline{a} = \mathrm{merge}_t(\overline{u}, \overline{x}) \\ \overline{y} \text{ are fresh v-variables} \quad |x| = |y| \\ \overline{b} = \mathrm{merge}_t(\overline{u}, \overline{y}) \\ c \text{ is a fresh constant} \end{array}}{E[\Gamma] \vdash_K t\ \overline{a}\ \overline{z} \triangleright (E; (c := \lambda \overline{y}.\ t\ \overline{b}))[\Gamma] \vdash c\ \overline{x}\ \overline{z}}$$

$K$ is a non-application context to restrict $t\ \overline{a}\ \overline{z}$ is not at a function position of an application. ($K = \$, (\lambda x.\ []), (t\ []), (\texttt{let}\ x := []\ \texttt{in}\ u), (\texttt{let}\ x := t\ \texttt{in}\ []),$ or ... but NOT $([]\ u)$.)

$\mathrm{merge}_t(\overline{u}, \overline{x})$ represents a sequence of terms which two sequences of terms are merged according to the static arguments definition of $t$. The first argument $\overline{u}$ specifies static arguments. The second argument $\overline{x}$ specifies dynamic arguments. For example, assuming the 1st and 4th arguments are static for $t$, $\mathrm{merge}_t((u_1, u_2), (x_1, x_2, x_3)) = (u_1, x_1, x_2, u_2, x_3)$.

This transformation removes non-variable arguments from applications. Thus the result will be V-normal form again.

$$\begin{array}{rcll}
t^{\mathrm{D}} & = & \lambda x{:}T.\ t^{\mathrm{DL}} & \\
& | & \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j & \\
t^{\mathrm{DL}} & = & t^{\mathrm{D}} & \\
& | & t^{\mathrm{L}} & \text{the type of } t^{\mathrm{L}} \text{ is an inductive type} \\
t^{\mathrm{L}} & = & \texttt{let}\ \overline{x := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}} & \\
t^{\mathrm{M}} & = & \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} & \\
& | & t^{\mathrm{E}} & \\
t^{\mathrm{E}} & = & x \mid f \mid c \mid C \mid T & \\
& | & t^{\mathrm{F}}\ \overline{x} & 0 < |x| \\
& | & t^{\mathrm{D}} & \\
t^{\mathrm{F}} & = & x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j &
\end{array}$$

## 4.10 Eta Reduction to Expose Fixpoint

$$\text{etared-fix: } \frac{E[\Gamma] \vdash t : \forall \overline{x{:}T}.\ U \quad \overline{x} \text{ does not occur in } t \quad t \text{ is a fixpoint}}{E[\Gamma] \vdash \lambda \overline{x{:}T}.\ t\ \overline{x} \triangleright t}$$

Note:

- We require the types of arguments of $t$ as $\overline{T}$ to prevent this transfomation changes the type.

- The premise "$t$ is a fixpoint" guarantees the result is not partial application.

This transformation is intended to remove eta-redexes introduced by static arguments and eta expansion (Section 4.3). For example, assume the standard list concatenation function, $\texttt{app} : \forall \texttt{A}, \texttt{list A} \to \texttt{list A} \to \texttt{list A}$, is monomorphized to `bool`.

$$
\begin{aligned}
& \texttt{app bool} \\
\triangleright_{\text{inline}} \ & (\lambda \texttt{A}.\ (\texttt{fix} \ldots))\ \texttt{bool} \\
\triangleright_{\text{eta-expansion}} \ & \lambda \texttt{l}.\ \lambda \texttt{m}.\ (\lambda \texttt{A}.\ (\texttt{fix} \ldots))\ \texttt{bool l m} \\
\triangleright_{\text{V-normalization}} \ & \lambda \texttt{l}.\ \lambda \texttt{m}.\ \texttt{let T} := \texttt{bool in}\ (\lambda \texttt{A}.\ (\texttt{fix} \ldots))\ \texttt{T l m} \\
\triangleright_{\text{S-normalization}} \ & \lambda \texttt{l}.\ \lambda \texttt{m}.\ \texttt{let T} := \texttt{bool in}\ (\texttt{fix} \ldots)\ \texttt{l m} \\
\triangleright_{\text{Type-nomalization}} \ & \lambda \texttt{l}.\ \lambda \texttt{m}.\ \texttt{let T} := \texttt{bool in}\ (\texttt{fix} \ldots)\ \texttt{l m} \qquad \text{(expand T in the fix-term)} \\
\triangleright_{\text{zeta-del}} \ & \lambda \texttt{l}.\ \lambda \texttt{m}.\ (\texttt{fix} \ldots)\ \texttt{l m} \\
\triangleright_{\text{etared-fix}} \ & (\texttt{fix} \ldots)
\end{aligned}
$$

The code generator (Section 5) generates multiple C functions (Section 5.10) from the pre-eta-reduction term, $\lambda \texttt{l}.\ \lambda \texttt{m}.\ (\texttt{fix} \ldots)\ \texttt{l m}$. This eta-reduction avoid this. The code generator generates single C function (Section 5.11) from post-eta-reduction term, $(\texttt{fix} \ldots)$.

## 4.11 Argument Completion

Argument completion removes partial applications by applying eta expansions.

$$
\text{argcomp-papp:}\ \frac{\begin{array}{c} t \text{ is not an application} \\ 0 < |x| \\ E[\Gamma] \vdash t\ \overline{x} : \forall \overline{y{:}T}, U \\ U \text{ is an inductive type} \end{array}}{E[\Gamma] \vdash_K t\ \overline{x} \triangleright \lambda \overline{y{:}T}.\ t\ \overline{x}\ \overline{y}}
$$

$$
\text{argcomp-fvar-cnst-cstr:}\ \frac{\begin{array}{c} t \text{ is a f-variable, constant, or constructor} \\ E[\Gamma] \vdash t : \forall \overline{y{:}T}, U \\ U \text{ is an inductive type} \end{array}}{E[\Gamma] \vdash_K t \triangleright \lambda \overline{y{:}T}.\ t\ \overline{y}}
$$

$K$ is a non-application context as in Section 4.9.

This transformation makes the result of an application inductive type. Also, constants and constructors are always fully applied to arguments. F-variables cannot occur in $t^{\mathrm{E}}$ because they cannot be a inductive type.

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda x{:}T.\, t^{\mathrm{DL}} \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j \\
t^{\mathrm{DL}} &= t^{\mathrm{D}} \\
&\mid t^{\mathrm{L}} && \text{the type of } t^{\mathrm{L}} \text{ is an inductive type} \\
t^{\mathrm{L}} &= \texttt{let}\ \overline{x := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}} \\
t^{\mathrm{M}} &= \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} \\
&\mid t^{\mathrm{E}} \\
t^{\mathrm{E}} &= x \mid T \\
&\mid c \mid C && \text{The type of } c \text{ and } C \text{ are inductive type} \\
&\mid t^{\mathrm{F}}\ \overline{x} && 0 < |x|, \text{ the type of } t^{\mathrm{F}}\ \overline{x} \text{ is inductive type} \\
&\mid t^{\mathrm{D}} \\
t^{\mathrm{F}} &= x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j
\end{aligned}
$$

## 4.12 Unreachable Fixfunc Deletion

Codegen deletes unreachable fix-bounded functions.

## 4.13 Monomorphism Check

We check the transformed term is a monomorphic term.

Although our transformations removes many rank-1 polymorphism, it still possible to retain polymorphic term. For example, our transformations don't remove polymorphic recursion unless the recursion is completely unrolled.

This step checks (1) all type annotations are inductive or function types without free variables, and (2) types and sorts doesn't occur at expression.

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda x{:}T.\, t^{\mathrm{DL}} \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j \\
t^{\mathrm{DL}} &= t^{\mathrm{D}} \\
&\mid t^{\mathrm{L}} && \text{the type of } t^{\mathrm{L}} \text{ is an inductive type} \\
t^{\mathrm{L}} &= \texttt{let}\ \overline{x := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}} \\
t^{\mathrm{M}} &= \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} \\
&\mid t^{\mathrm{E}} \\
t^{\mathrm{E}} &= x && \leftarrow T \text{ is removed} \\
&\mid c \mid C && \text{The type of } c \text{ and } C \text{ are inductive type} \\
&\mid t^{\mathrm{F}}\ \overline{x} && 0 < |x|, \text{ the type of } t^{\mathrm{F}}\ \overline{x} \text{ is inductive type} \\
&\mid t^{\mathrm{D}} \\
t^{\mathrm{F}} &= x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j
\end{aligned}
$$

## 4.14 Borrow Check

We define two judgements $E[\Gamma] \vdash t : T \mid B$ and $E[\Gamma] \vdash t : T \mid (L, B^{\mathrm{used}}, B^{\mathrm{result}})$ for borrow check. We extend $\Gamma$ in this section. $\Gamma$ is an annotated local context. It is a list of $(x^B{:}T)$, $(f{:}T)$ and $(x^B := t{:}T)$. They are same as local assumption and local definitions except that the v-variable $x$ is annotated with a borrow information $B$. (The f-variable $f$ is not-annotated.) $B$ is a set of pair of borrow type and linear

variable, such as $\{\overline{(T, x)}\}$. $B^{\text{used}}$ and $B^{\text{result}}$ are also borrow information. $L$ is a set of linear variables. $T$ is the type of $t$.

We write $\mathrm{B}_\Gamma\, x$ to refer the borrow information for $x$ in $\Gamma$. $\mathrm{B}_\Gamma\, x = B$ if $\Gamma$ contains $(x^B{:}T)$ or $(x^B := t{:}T)$.

We omit $: T$ in a rule which does not use $T$.

The borrow information $B = \{\overline{(T, x)}\}$ represents a linear variable $x_i$ is used via borrow type $T_i$. $(x^{\{(T', y)\}}{:}T) \in \Gamma$ represents $x$ may contain a value of type $T'$ which is a (part of) content of the linear variable $y$.

$E[\Gamma] \vdash t \mid B$ means a function $t$ may use linear variables via borrow $B$.

$E[\Gamma] \vdash t \mid (L, B^{\text{used}}, B^{\text{result}})$ means an expression $t$ (1) consumes linear variables $L$, (2) may use linear values via borrow $B^{\text{used}}$, (3) result value may contain linear values via borrow $B^{\text{result}}$.

For example, assume linear list `lseq`, borrow list `bseq` which has constructors `bnil` and `bcons`, borrow function `borrow : lseq nat → bseq nat`. In a code fragment
`let y := borrow x in match y with bnil ⇒ true | bcons h t ⇒ false end` contains variables `x : lseq nat`, `y : bseq nat`, `h : nat`, and `t : bseq nat`. `y` and `t` contain a `bseq nat` value borrowed from `x`. It is represented as $\texttt{y}^{\{(\texttt{bseq nat},\texttt{x})\}} : \texttt{bseq nat}$ and $\texttt{t}^{\{(\texttt{bseq nat},\texttt{x})\}} : \texttt{bseq nat}$. `h` is annotated as $\texttt{h}^{\varnothing}$ which means `h` does not contain borrowed values. The type of `h` is `nat`. Since `nat` is not a borrow type, `h` lives even after `x` is consumed.

borrow-lvar: $\dfrac{(x^B{:}T) \in \Gamma \quad x \text{ is linear}}{E[\Gamma] \vdash x \mid (\{x\}, B, B)}$

borrow-vvar: $\dfrac{(x^B{:}T) \in \Gamma \quad x \text{ is not linear}}{E[\Gamma] \vdash x \mid (\varnothing, B, B)}$

borrow-fvar: $E[\Gamma] \vdash f \mid (\varnothing, \varnothing, \varnothing)$

borrow-constant: $\dfrac{c \text{ is not a borrow function}}{E[\Gamma] \vdash c \mid (\varnothing, \varnothing, \varnothing)}$

borrow-constructor: $E[\Gamma] \vdash C \mid (\varnothing, \varnothing, \varnothing)$

borrow-letin: $$\frac{\begin{gathered} E[\Gamma] \vdash t_1 \mid (L_1, B_1^{\text{used}}, B_1^{\text{result}}) \\ E[\Gamma; (x^{B_1^{\text{result}}} := t_1{:}T)] \vdash t_2 \mid (L_2, B_2^{\text{used}}, B_2^{\text{result}}) \\ L_1 \cap L_2 = \varnothing \\ x \text{ is linear} \to x \in L_2 \\ L_1 \cap B_2^{\text{used}} = \varnothing \end{gathered}}{E[\Gamma] \vdash \texttt{let}\ x := t_1 : T\ \texttt{in}\ t_2 \mid (L_1 \cup L_2 - \{x\}, B_1^{\text{used}} \cup B_2^{\text{used}} - \{x\}, B_2^{\text{result}} - \{x\})}$$

borrow-match: $$\frac{\begin{gathered} E[\Gamma] \vdash y \mid (L_{\text{item}}, B_{\text{item}}^{\text{used}}, B_{\text{item}}^{\text{result}}) \\ B_{ij} = B_{\text{item}}^{\text{result}} \cap \text{components}_E(T_{ij}) \\ \overline{\Gamma' = \overline{(x^B{:}T)}} \\ \overline{\underline{E}[\underline{\Gamma}; \Gamma'] \vdash t \mid (L^{\text{branch}}, B^{\text{used}}, B^{\text{result}})} \\ \overline{L^{\text{branch}} \cap \{\overline{x}\} = \{z \mid z \in \{\overline{x}\} \ \wedge\ z \text{ is linear}\}} \\ \overline{L^{\text{branch'}} = L^{\text{branch}} - \{\overline{x}\}} \quad \forall i, \forall j, \left(L_i^{\text{branch'}} = L_j^{\text{branch'}}\right) \quad L_{\text{item}} \cap L_1^{\text{branch'}} = \varnothing \\ B_{\text{branches}}^{\text{used}} = \bigcup_i (B_i^{\text{used}} - \{\overline{x_i}\}) \quad B_{\text{branches}}^{\text{result}} = \bigcup_i (B_i^{\text{result}} - \{\overline{x_i}\}) \quad L_{\text{item}} \cap B_{\text{branches}}^{\text{used}} = \varnothing \end{gathered}}{E[\Gamma] \vdash \texttt{match}\ y\ \texttt{with}\ \overline{C\ \overline{x{:}T} \Rightarrow t}\ \texttt{end} \mid (L_{\text{item}} \cup L_1^{\text{branch'}}, B_{\text{item}}^{\text{used}} \cup B_{\text{branches}}^{\text{used}}, B_{\text{branches}}^{\text{result}})}$$

borrow-vvar-app: $\dfrac{(y^B{:}T') \in \Gamma \quad \text{APP}(E, \Gamma, B, \overline{x}, T, L, B^{\text{used}}, B^{\text{result}})}{E[\Gamma] \vdash y\ \overline{x} : T \mid (L, B^{\text{used}}, B^{\text{result}})}$

borrow-fvar-app: $\dfrac{(f{:}T') \in \Gamma \quad \text{APP}(E, \Gamma, \varnothing, \overline{x}, T, L, B^{\text{used}}, B^{\text{result}})}{E[\Gamma] \vdash f\ \overline{x} : T \mid (L, B^{\text{used}}, B^{\text{result}})}$

borrow-constant-app: $\dfrac{c \text{ is not a borrow function} \quad \text{APP}(E, \Gamma, \varnothing, \overline{x}, T, L, B^{\text{used}}, B^{\text{result}})}{E[\Gamma] \vdash c\ \overline{x} : T \mid (L, B^{\text{used}}, B^{\text{result}})}$

borrow-constructor-app: $\dfrac{\text{APP}(E, \Gamma, \varnothing, \overline{x}, T, L, B^{\text{used}}, B^{\text{result}})}{E[\Gamma] \vdash C\ \overline{x} : T \mid (L, B^{\text{used}}, B^{\text{result}})}$

borrow-fix-app: $\dfrac{E[\Gamma] \vdash \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \mid B \quad \text{APP}(E, \Gamma, B, \overline{x}, T, L, B^{\text{used}}, B^{\text{result}})}{E[\Gamma] \vdash (\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j)\ \overline{x} : T \mid (L, B^{\text{used}}, B^{\text{result}})}$

borrow-borrow: $$\frac{\begin{gathered} c \text{ is a borrow function} \\ E[\Gamma] \vdash c : T^{\text{arg}} \to T^{\text{result}} \quad T^{\text{arg}} \text{ is a linear type} \quad T^{\text{result}} \text{ is a borrow type} \\ T^{\text{result}} \text{ does not contain function} \\ \{\overline{T}\} \text{ is the set of borrow types contained in } T^{\text{result}} \\ B = \{\overline{(T, \underline{x})}\} \end{gathered}}{E[\Gamma] \vdash c\ x \mid (\varnothing, B, B)}$$

$$\text{borrow-fix-clo: } \frac{E[\Gamma] \vdash \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \mid B}{E[\Gamma] \vdash \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \mid (\varnothing, B, B)}$$

$$\text{borrow-abs-clo: } \frac{E[\Gamma] \vdash \lambda x.\, t \mid B}{E[\Gamma] \vdash \lambda x.\, t \mid (\varnothing, B, B)}$$

$$\text{borrow-abs-fun: } \frac{\begin{array}{c} t \text{ is not an abstraction} \\ t \text{ is not a fixpoint} \\ E[\Gamma; \overline{(x^{\varnothing}{:}T)}] \vdash t \mid (L, B^{\text{used}}, B^{\text{result}}) \\ \{z | z \in \{\overline{x}\} \wedge z \text{ is linear}\} = L \\ B' = (B^{\text{used}} - \{\overline{x}\}) \end{array}}{E[\Gamma] \vdash \lambda \overline{x{:}T}.\, t \mid B'}$$

$$\text{borrow-abs-fix: } \frac{\begin{array}{c} t \text{ is a fixpoint} \\ E[\Gamma; \overline{(x^{\varnothing}{:}T)}] \vdash t \mid B \\ \forall z \in \{\overline{x}\},\ z \text{ is not linear} \end{array}}{E[\Gamma] \vdash \lambda \overline{x{:}T}.\, t \mid B}$$

$$\text{borrow-fix-fun: } \frac{\overline{\underline{E[\Gamma; \overline{(f{:}T)}]} \vdash t \mid B}}{E[\Gamma] \vdash \texttt{fix}\ \overline{f{:}T := t}\ \texttt{for}\ f_j \mid \bigcup_i B_i}$$

$$\begin{aligned} &\text{APP}(E, \Gamma, B^{\text{func}}, \overline{x}, T, L, B^{\text{used}}, B^{\text{result}}) \\ &= 1 \leq |x| \\ &\wedge \forall i, \forall j, (i \neq j \rightarrow \neg(x_i = x_j \wedge x_i \text{ is linear})) \\ &\wedge L = \{z | z \in \{\overline{x}\} \ \wedge\ z \text{ is linear}\} \\ &\wedge B^{\text{used}} = B^{\text{func}} \cup \bigcup_i (\mathrm{B}_\Gamma\, x_i) \\ &\wedge B^{\text{result}} = B^{\text{used}} \cap \text{components}_E(T) \\ &\wedge B^{\text{used}} \cap L = \varnothing \end{aligned}$$

We use a function $\text{components}_E(T)$ to obtain the component types of a type $T$ under the global environment $E$. It returns a set of types or $\top$. $\top$ is a set which contains all types. $\text{components}_E(T)$ is

defined as the minimum set which satisfy following equations.

$$
\begin{aligned}
&\mathrm{components}_E(I\,\overline{t}) = \{I\,\overline{t}\} \cup \textstyle\bigcup_{(x:T)\in\Gamma_B} \mathrm{components}_E(T)\\
&\quad \text{where } E[] \vdash I\,\overline{t} : S\\
&\qquad\quad S \text{ is a sort}\\
&\qquad\quad \mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E\\
&\qquad\quad p \text{ the is number of recursively uniform parameters of } \mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)\\
&\qquad\quad I \in \Gamma_I\\
&\qquad\quad |t| = p\\
&\qquad\quad (C : \forall\Gamma_P, \forall\Gamma_A, I\,\overline{u}) \in \Gamma_C\\
&\qquad\quad (I\,\overline{u})\{\Gamma_P/\overline{t}\} = I\,\overline{t}\\
&\qquad\quad \Gamma_B = \Gamma_A\{\Gamma_P/\overline{t}\}\\
&\mathrm{components}_E(\forall T, U) = \top\\
&\mathrm{components}_E(S) = \top\\
&\quad \text{where } S \text{ is a sort}
\end{aligned}
$$

We extend substitution for local contexts here. $t\{\Gamma/\overline{u}\}$ represents the term $t$ which variables in $\Gamma$ are substituted with terms $\overline{u}$. $\Gamma'\{\Gamma/\overline{u}\}$ represents the local context $\Gamma'$ which variables in $\Gamma$ are substituted with terms $\overline{u}$. If $\Gamma$ contains a local definition, its variable is substituted with the corresponding definition. ($\epsilon$ is the empty local context. $\varepsilon$ is the empty list of terms.)

$$
\begin{aligned}
t\{\epsilon/\varepsilon\} &= t\\
t\{((x{:}T);\Gamma)/(a\,\overline{u})\} &= t\{x/a\}\{\Gamma\{x/a\}/\overline{u}\}\\
t\{((x := a{:}T);\Gamma)/\overline{u}\} &= t\{x/a\}\{\Gamma\{x/a\}/\overline{u}\}\\
\epsilon\{\Gamma/\overline{u}\} &= \epsilon\\
((x{:}T);\Gamma')\{\Gamma/\overline{u}\} &= (x{:}T\{\Gamma/\overline{u}\});\Gamma'\{\Gamma/\overline{u}\}\\
((x := a{:}T);\Gamma')\{\Gamma/\overline{u}\} &= (x := a\{\Gamma/\overline{u}\}{:}T\{\Gamma/\overline{u}\});\Gamma'\{\Gamma/\overline{u}\}
\end{aligned}
$$

We mix borrow information and set of variables in set-operations. Assume $L = \{x_1, \ldots, x_n\}$ and $B = \{(T_1, y_1), \ldots, (T_m, y_m)\}$.

$$
\begin{aligned}
B \cap L = L \cap B &= \{(T_i, y_i) \in B \mid 1 \le i \le m,\ y_i \in L\}\\
B - L &= \{(T_i, y_i) \in B \mid 1 \le i \le m,\ y_i \notin L\}
\end{aligned}
$$

We also mix borrow information and set of types (including $\top$) in set-operations.

$$
\begin{aligned}
&B \cap D = D \cap B = \{(T_i, y_i) \in B \mid 1 \le i \le m,\ T_i \in D\} \qquad\qquad D = \{U_1, \ldots, U_n\}\\
&B \cap \top = \top \cap B = B
\end{aligned}
$$

Note:

- We don't annotate f-variables. This is not correct because invoking $f_1 \ldots f_n$ may refer borrowed values via free variables in `fix` $\overline{f{:}T := t}$ `for` $f_j$. However, it is harmless because corresponding linear value cannot be consumed in the fix-term.

## 4.15 C Variable Allocation

We rename variables to be unique and approproate for C.

Since Gallina variables are represented by de Bruijn's indexes, we only need to change variable names in binders: (1) variable of abstraction, (2) functions of fixpoint, (3) variable of let-in, and (4) variables of constructor members of conditional.

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda\boxed{x}{:}T.\, t^{\mathrm{DL}} && \leftarrow (1)\\
&\mid \texttt{fix}\ \overline{\boxed{f}/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j && \leftarrow (2)\\
t^{\mathrm{DL}} &= t^{\mathrm{D}}\\
&\mid t^{\mathrm{L}} && \text{the type of } t^{\mathrm{L}} \text{ is an inductive type}\\
t^{\mathrm{L}} &= \texttt{let}\ \overline{\boxed{x} := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}} && \leftarrow (3)\\
t^{\mathrm{M}} &= \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{\boxed{x}} \Rightarrow t^{\mathrm{L}}}\ \texttt{end} && \leftarrow (4)\\
&\mid t^{\mathrm{E}}\\
t^{\mathrm{E}} &= x\\
&\mid c \mid C && \text{The type of } c \text{ and } C \text{ are inductive type}\\
&\mid t^{\mathrm{F}}\ \overline{x} && 0 < |x|, \text{ the type of } t^{\mathrm{F}}\ \overline{x} \text{ is inductive type}\\
&\mid t^{\mathrm{D}}\\
t^{\mathrm{F}} &= x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{\boxed{f}/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j && \leftarrow (2)
\end{aligned}
$$

# 5 C Code Generation

## 5.1 The Gallina Subset for C Code Generation

$$
\begin{aligned}
t^{\mathrm{D}} &= \lambda x{:}T.\, t^{\mathrm{DL}}\\
&\mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j\\
t^{\mathrm{DL}} &= t^{\mathrm{D}}\\
&\mid t^{\mathrm{L}} && \text{the type of } t^{\mathrm{L}} \text{ is an inductive type}\\
t^{\mathrm{L}} &= \texttt{let}\ \overline{x := t^{\mathrm{M}} : T}\ \texttt{in}\ t^{\mathrm{M}}\\
t^{\mathrm{M}} &= \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t^{\mathrm{L}}}\ \texttt{end}\\
&\mid t^{\mathrm{E}}\\
t^{\mathrm{E}} &= x\\
&\mid c \mid C && \text{The type of } c \text{ and } C \text{ are inductive type}\\
&\mid t^{\mathrm{F}}\ \overline{x} && 0 < |x|, \text{ the type of } t^{\mathrm{F}}\ \overline{x} \text{ is inductive type}\\
&\mid t^{\mathrm{D}}\\
t^{\mathrm{F}} &= x \mid f \mid c \mid C \mid \texttt{fix}\ \overline{f/k{:}T := t^{\mathrm{D}}}\ \texttt{for}\ f_j
\end{aligned}
$$

## 5.2 Detection of Higher Order Fixfuncs

We call the set of f-variables of higher order fixfuncs HigherOrderFixfunc.

## 5.3 Detection of Inlinable Fixpoints

We detect inlinable fixpoints. "Inlinable fixpoint" means a fixpoint, $\texttt{fix}\,\overline{f := t}\,\texttt{for}\,f_j$, which all applications of $\overline{f}$ are located at the tail positions of $\overline{f}$. In this case, the continuation of the applications to $\overline{f}$ in $\texttt{let}\,x := (\texttt{fix}\,\overline{f := t}\,\texttt{for}\,f_j)\,\overline{y}\,\texttt{in}\,u$ are always $\texttt{let}\,x := \square\,\texttt{in}\,u$. Thus, we can translate the tail positions of $\overline{f}$ to (1) assignments to the arguments of $f_i$ and $\texttt{goto}\ f_i$ for application of $f_i$ and (2) assignment to $x$ and $\texttt{goto}\ u$ otherwise. This translation is equivalent to inlining a tail-recursive function, which means generating a loop at a non-tail position.

$\mathrm{TR}_{\mathrm{FUN}}[\![t]\!]$ is the second component of $\mathrm{IFIX}_{\mathrm{FUN}}[\![t]\!]$. It is a set of fixfuncs which are translatable without actual functions.

$\mathrm{IFIX}_{\mathrm{FUN}}[\![t]\!]$ and $\mathrm{IFIX}_{\mathrm{EXP}}[\![t]\!]$ are defined mutually recursive. $\mathrm{IFIX}_{\mathrm{FUN}}[\![t]\!]$ is four-tuple. $\mathrm{IFIX}_{\mathrm{EXP}}[\![t]\!]$ is three-tuple. We introduce 7 functions to refer the components of them.

$$\mathrm{IFIX}_{\mathrm{FUN}}[\![t]\!] = (\mathrm{INL}[\![t]\!], \mathrm{TR}_{\mathrm{FUN}}[\![t]\!], \mathrm{HEAD}_{\mathrm{FUN}}[\![t]\!], \mathrm{TAIL}_{\mathrm{FUN}}[\![t]\!])$$
$$\mathrm{IFIX}_{\mathrm{EXP}}[\![t]\!] = (\mathrm{TR}_{\mathrm{EXP}}[\![t]\!], \mathrm{HEAD}_{\mathrm{EXP}}[\![t]\!], \mathrm{TAIL}_{\mathrm{EXP}}[\![t]\!])$$

The components means as follows.

$$\begin{aligned}
\mathrm{INL}[\![t]\!] &: t \text{ is inlinable or not (T or F)} \\
\mathrm{TR}_{\mathrm{FUN}}[\![t]\!], \mathrm{TR}_{\mathrm{EXP}}[\![t]\!] &: \text{set of tail-recursive fixfuncs in } t \\
\mathrm{HEAD}_{\mathrm{FUN}}[\![t]\!], \mathrm{HEAD}_{\mathrm{FUN}}[\![t]\!] &: \text{free f-variables at head positions of } t \\
\mathrm{TAIL}_{\mathrm{FUN}}[\![t]\!], \mathrm{TAIL}_{\mathrm{FUN}}[\![t]\!] &: \text{free f-variables at tail positions of } t
\end{aligned}$$

$\mathrm{IFIX}_{\mathrm{FUN}}[\![t]\!]$ traverses abstractions and fixpoints ($t^{\mathrm{D}}$ and $t^{\mathrm{DL}}$). $\mathrm{IFIX}_{\mathrm{EXP}}[\![t]\!]$ traverses other constructs ($t^{\mathrm{L}}$, $t^{\mathrm{M}}$, $t^{\mathrm{E}}$, and $t^{\mathrm{F}}$).

"tail position" is extended to the function position of the application at a tail position.
$\mathrm{TR}_{\mathrm{FUN}}$ distinguishes fixpoint bounded functions translatable without actual functions (but with `goto`) or not.

$$\mathrm{IFIX}_{\mathrm{FUN}}[\![\lambda x.\, t]\!] = (\mathrm{INL}[\![t]\!], \mathrm{TR}_{\mathrm{FUN}}[\![t]\!], \mathrm{HEAD}_{\mathrm{FUN}}[\![t]\!], \mathrm{TAIL}_{\mathrm{FUN}}[\![t]\!])$$

$$\mathrm{IFIX}_{\mathrm{FUN}}[\![\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j]\!] = \begin{cases} (\mathrm{T}, & \left(\bigcup_i \mathrm{HEAD}_{\mathrm{FUN}}[\![t_i]\!] \cap \{\overline{f}\} = \varnothing\right) \wedge \\ \ \bigcup_i \mathrm{TR}_{\mathrm{FUN}}[\![t_i]\!] \cup \{\overline{f}\}, & \left(\mathrm{HigherOrderFixfunc} \cap \{\overline{f}\} = \varnothing\right) \wedge \\ \ \bigcup_i \mathrm{HEAD}_{\mathrm{FUN}}[\![t_i]\!] - \{\overline{f}\}, & \forall i,\ \ \mathrm{INL}[\![t_i]\!] \\ \ \bigcup_i \mathrm{TAIL}_{\mathrm{FUN}}[\![t_i]\!] - \{\overline{f}\}) & \\ (\mathrm{F}, & \text{otherwise} \\ \ \bigcup_i \mathrm{TR}_{\mathrm{FUN}}[\![t_i]\!], & \\ \ \bigcup_i (\mathrm{HEAD}_{\mathrm{FUN}}[\![t_i]\!] \cup \mathrm{TAIL}_{\mathrm{FUN}}[\![t_i]\!]) - \{\overline{f}\}, & \\ \ \varnothing) & \end{cases}$$

$$\mathrm{IFIX}_{\mathrm{FUN}}[\![t]\!] = (\mathrm{T}, \mathrm{TR}_{\mathrm{EXP}}[\![t]\!], \mathrm{HEAD}_{\mathrm{EXP}}[\![t]\!], \mathrm{TAIL}_{\mathrm{EXP}}[\![t]\!])$$
$$\text{if } t \text{ is not abstraction nor fixpoint}$$

$$\begin{aligned}
\mathrm{IFIX}_{\mathrm{EXP}}[\![x\ \overline{y}]\!] &= (\varnothing, \varnothing, \varnothing) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![f\ \overline{y}]\!] &= (\varnothing, \varnothing, \{f\}) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![c\ \overline{y}]\!] &= (\varnothing, \varnothing, \varnothing) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![C\ \overline{y}]\!] &= (\varnothing, \varnothing, \varnothing) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![\texttt{let}\ x := t\ \texttt{in}\ u]\!] &= (\ \mathrm{TR}_{\mathrm{EXP}}[\![t]\!] \cup \mathrm{TR}_{\mathrm{EXP}}[\![u]\!], \\
&\qquad \mathrm{HEAD}_{\mathrm{EXP}}[\![t]\!] \cup \mathrm{TAIL}_{\mathrm{EXP}}[\![t]\!] \cup \mathrm{HEAD}_{\mathrm{EXP}}[\![u]\!], \\
&\qquad \mathrm{TAIL}_{\mathrm{EXP}}[\![u]\!]) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![\texttt{match}\ y\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end}]\!] &= \left(\bigcup_i \mathrm{TR}_{\mathrm{EXP}}[\![t_i]\!], \bigcup_i \mathrm{HEAD}_{\mathrm{EXP}}[\![t_i]\!], \bigcup_i \mathrm{TAIL}_{\mathrm{EXP}}[\![t_i]\!]\right) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![t]\!] &= (\mathrm{TR}_{\mathrm{FUN}}[\![t]\!], \mathrm{HEAD}_{\mathrm{EXP}}[\![t]\!] \cup \mathrm{TAIL}_{\mathrm{EXP}}[\![t]\!], \varnothing) \\
&\qquad \text{if } (t = \lambda x.\, u) \vee (t = \texttt{fix}\ \overline{f := u}\ \texttt{for}\ f_j) \\
\mathrm{IFIX}_{\mathrm{EXP}}[\![t\ \overline{y}]\!] &= (\mathrm{TR}_{\mathrm{FUN}}[\![t]\!], \mathrm{HEAD}_{\mathrm{EXP}}[\![t]\!], \mathrm{TAIL}_{\mathrm{EXP}}[\![t]\!]) \\
&\qquad \text{if } (t = \texttt{fix}\ \overline{f := u}\ \texttt{for}\ f_j) \wedge (|y| > 0)
\end{aligned}$$

Note:

- The variables in $t$ are unique. Codegen uses de Bruijn's indexes for HEAD and TAIL; the variables renamed by Section 4.15 for TR.
- $y$ of $\texttt{match}\ y\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t}\ \texttt{end}$ is not counted because $y$ is not a function and does not affect the final TR.
- INL guarantees that consecutive fixpoints and abstractions ($t^{\mathrm{D}}$, traversed by $\mathrm{IFIX}_{\mathrm{LF}}$) synthesizes non-inlinableness: an outer fixpoint is not inlinable if an inner fixpoint is not-inlinable. This property holds in most case without INL but an curious fixpoint (such as non-recursive fixpoint) can break this property. Thus we added INL.

## 5.4 Head Position and Tail Position

$$
\begin{aligned}
t_p^{\mathrm{DL}} &= t_p^{\mathrm{D}} \\
&\mid t_p^{\mathrm{L}} \\
t_p^{\mathrm{D}} &= \lambda x{:}T.\, t_p^{\mathrm{DL}} \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t_p^{\mathrm{D}}}\ \texttt{for}\ f_j \\
t_p^{\mathrm{L}} &= \texttt{let}\ \overline{x := t_{\mathrm{HEAD}}^{\mathrm{M}} : T}\ \texttt{in}\ t_p^{\mathrm{M}} \\
t_p^{\mathrm{M}} &= \texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{x} \Rightarrow t_p^{\mathrm{L}}}\ \texttt{end} \\
&\mid t_p^{\mathrm{E}} \\
t_p^{\mathrm{E}} &= x \\
&\mid c \mid C \\
&\mid t_p^{\mathrm{F}}\ \overline{x} \\
&\mid t_{\mathrm{TAIL}}^{\mathrm{D}} \\
t_p^{\mathrm{F}} &= x \mid f \mid c \mid C \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t_p^{\mathrm{D}}}\ \texttt{for}\ f_j && f_j \in TR \\
&\mid \texttt{fix}\ \overline{f/k{:}T := t_{\mathrm{TAIL}}^{\mathrm{D}}}\ \texttt{for}\ f_j && f_j \notin TR
\end{aligned}
$$

Note:

- $TR = \mathrm{TR}_{\mathrm{FUN}}[\![t]\!]$ where the translating function is defined as `Definition` $c := t$.

## 5.5 Top-Level Functions Detection

If a fixpoint needs recursive call in C, we need a real C function for it. Codegen detects such fixpoints by simulating $A_K$ and $B_K$ in Section 5.7 and Section 5.8 to collect application of fixpoint-bounded functions.

## 5.6 Fix-lifting

We use lambda-lifting-like technique to translate fixpoint expressions without closures.

Consider following (artificial) example.

```
Definition c x y :=
  fix f n :=
  match n with
  | O ⇒ x                    (* invocation of f needs x *)
  | S n' ⇒
    (fix g m :=
    match m with
    | O ⇒ y + f n'           (* invocation of g needs y and f *)
    | S m' ⇒ S (g m')
    end) n
  end.
```

Codegen translate a Gallina application to C function call (if `goto` is not usable). In this scenario, Codegen generates C functions corresponding to internal functions `f` and `g`. The application in Gallina, `g m'`, is translated to `g(m')` in C. But it does not work because `g` needs `y` but the global C function `g` does not know `y`. Also, `g` needs `f`. Although there is the C function `f`, `f` needs `x`. Thus, Codegen need to add extra arguments as `g(x, y, m')` which is similar to lambda-lifting. We call this translation, adding extra arguments for functions bounded by fixpoints, fix-lifting.

We define FIXFUNCS, FIXFV, EXARGS′, and EXARGS for each definition `Definition` $c := u$.

EXARGS($f$) is the extra arguments for the fix-bounded function $f$ in the definition.

FIXFUNCS is the set of f-variables in $u$.

$$\text{FIXFUNCS} = \bigcup_{K[\texttt{fix}\ \overline{f:=t}\ \texttt{for}\ f_j]=u} \{\overline{f}\}$$

FIXK($f$) is the context of the fixpoint which bounds f-variable $f$.

$$\text{FIXK}(f) = K \quad \text{where } u = K[\texttt{fix}\ \overline{(g_l := t_l)}^{1\le l<i}(f := t_i)\overline{(g_l := t_l)}^{i<l\le h}\ \texttt{for}\ g_j]$$

FIXFV($f$) is the set of free variables of the fixpoint which bounds f-variable $f$. FIXFV($x$) is also defined as empty set for v-variable $x$.

$$\text{FIXFV}(f) = \text{FV}(t) \quad \text{where } u = \text{FIXK}(x)[t]$$
$$\text{FIXFV}(x) = \varnothing$$

EXARGS$'(f)$ is a set which satisfy the following conditions. EXARGS$'(f)$ is similar to FIXFV($f$). But if it contains a function bounded by an outer fixpoint, the free variable of the outer fixpoint are also contained.

$$\text{EXARGS}'(f) \supseteq \text{FIXFV}(f) \cup \bigcup_{t\in\text{FIXFV}(f)} \text{EXARGS}'(t)$$
$$\text{EXARGS}'(f) \subseteq \text{KV}(\text{FIXK}(f))$$
$$\text{EXARGS}'(x) = \varnothing$$

Codegen chooses the minimal set for EXARGS$'(f)$ if a dedicated internal C function is generated for $f$. But if $f$ is callable via $c$ like follows, Codegen generate a call to $c$ for $f$ to avoid generating the dedicated C function for `f`. This means `f y'` is translated to `c(x, y')`. In this case, the arguments of the external C function for $c$ must corresponds to the type of $c$. Thus Codegen chooses the maximal set (all bound variables) for EXARGS$'(f)$. (This means that `x` is passed even if `f` does not use `x`.)

```
Definition c := λx. fix f := λy. ... f y' ... for f
```

EXARGS($f$) is defined as follows. It is EXARGS$'(f)$ except fix-bounded functions.

$$\text{EXARGS}(f) = \text{EXARGS}'(f) - \text{FIXFUNCS}$$

When EXARGS($f$) is used in a context which the order matters, we consider it is a list of variables from declared ouside to inside. (used in Section 5.7)

When EXARGS($f$) is used in a context which require types, we consider it is a set of pairs of variable and its type. (used in Section 5.10)

## 5.7 Translation to C for a Non-Tail Position

$\mathrm{A}_K[\![t]\!]$ generates C code for $t$ in a non-tail position. The result expression is passed to $K$. $K(e) =$ "$v = e$;" in simple situations.

$\mathrm{A}_K[\![x]\!] = K(\text{“}x\text{”})$

$\mathrm{A}_K[\![f\ \overline{x}]\!] =$ “passign(fvars$'[\![f]\!], \overline{x})$
  goto entry_$f$;”
  $(|x| > 0) \wedge f \in TR$

$\mathrm{A}_K[\![f\ \overline{x}]\!] = K(\text{“}f(\overline{y},\overline{x})\text{”})$ $\quad (|x| > 0) \wedge f \notin TR$ where $\overline{y} = \mathrm{EXARGS}(f)$

$\mathrm{A}_K[\![c\ \overline{x}]\!] = K(\text{“}c(\overline{x})\text{”})$ $\quad |x| \geq 0$

$\mathrm{A}_K[\![C\ \overline{x}]\!] = K(\text{“}C(\overline{x})\text{”})$ $\quad |x| \geq 0$

$\mathrm{A}_K[\![\texttt{let}\ x := t_1\ \texttt{in}\ t_2]\!] =$ “$\mathrm{A}_{K'}[\![t_1]\!]$ $\mathrm{A}_K[\![t_2]\!]$” where $K'(e) =$ “$x$ = $e$;”

$\mathrm{A}_K[\![\texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{y} \Rightarrow t}\ \texttt{end}]\!]$ where $x : T$

“switch (*swfunc*$_T(x)$) {
  ...
  *caselabel*$_{C_i}$: $\overline{y_{ij} = \mathit{get\_member}_{C_i j}(x);}^{\,j}$
    *linear_dealloc*$_T(x)$;
    $\mathrm{A}_K[\![t_i]\!]$
    break;
  ...
  }”

$\mathrm{A}_K[\![(\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j)\ \overline{x}]\!] =$ $\quad f_j \in TR$
  “passign(fvars$[\![t_j]\!], \overline{x})$ where
  GENBODY$^{\mathrm{AT}}_{K'}[\![\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j]\!]$
  exit_$f_j$:” $\quad K'(e) = \begin{cases} K(e) & K(e) \text{ contains goto} \\ \text{“}K(e) \\ \quad \texttt{goto exit\_}f_j\texttt{;}\text{”} & \text{otherwise} \end{cases}$

$\mathrm{A}_K[\![(\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j)\ \overline{x}]\!] =$ $\quad f_j \notin TR \quad |x| > 0$
  “$K(f_j(\overline{y},\overline{x}))$ where
  goto skip_$f_j$;
  $\overline{y} = \mathrm{EXARGS}(f_j)$
  GENBODY$^{\mathrm{AN}}[\![\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j]\!]$
  skip_$f_j$:”

Note:

- “...” means a string. A string can contain characters in typewriter font and expressions starting in italic or roman font. The former is preserved as-is. The latter embeds the value of the expression (with name translation from Gallina to C).
- Gallina types, constants, and constructors have corresponding (user-configurable) C names and they are implicitly translated. Gallina variables are translated by the mapping defined in Section 4.15.
- $TR = \mathrm{TR}_{\mathrm{FUN}}[\![t]\!]$ where the translating function is defined as `Definition` $c := t$.
- *swfunc*$_T$, *caselabel*$_{C_i}$, and *get_member*$_{C_i j}$ are defined by a user to translate `match`-expressions for the inductive type $T$.
- *linear_dealloc*$_T(x)$ is the deallocation function for the linear type $T$. It is empty for unrestricted types.
- passign$(\overline{y}, \overline{x})$ is a parallel assignment. It is translated to a sequence of assignments to assign $x_1 \ldots x_n$ into $y_1 \ldots y_n$. It may require temporary variables.
- We do not define $\mathrm{A}_K[\![\lambda x.\ t]\!]$ because we do not support closures yet.
- Actual Codegen generates GENBODY$^{\mathrm{AN}}[\![]\!]$ in a different position to avoid the label skip_$f_j$ and gotoskip_$f_j$;.

## 5.8 Translation to C for a Tail Position

$\mathrm{B}_K[\![t]\!]$ generates C code for $t$ in a tail position. The result expression is passed to $K$.
$K(e) =$ "`return` $e$`;`" in simple situations.

$$
\begin{array}{lll}
\mathrm{B}_K[\![x]\!] = K(\text{“}x\text{”}) & & \\
\mathrm{B}_K[\![f\ \overline{x}]\!] = \text{“passign}(\mathrm{fvars}'[\![f]\!], \overline{x}) & |x| > 0 & \\
\quad \texttt{goto entry\_}f\texttt{;}\text{”} & & \\
\mathrm{B}_K[\![c\ \overline{x}]\!] = K(\text{“}c\texttt{(}\overline{x}\texttt{)}\text{”}) & |x| \geq 0 & \\
\mathrm{B}_K[\![C\ \overline{x}]\!] = K(\text{“}C\texttt{(}\overline{x}\texttt{)}\text{”}) & |x| \geq 0 & \\
\mathrm{B}_K[\![\texttt{let}\ x := t_1\ \texttt{in}\ t_2]\!] = \text{“}\mathrm{A}_{K'}[\![t_1]\!]\ \mathrm{B}_K[\![t_2]\!]\text{”} & \text{where} & K'(e) = \text{“}x\texttt{ = }e\texttt{;}\text{”}
\end{array}
$$

$\mathrm{B}_K[\![\texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{y} \Rightarrow t}\ \texttt{end}]\!] =$ where $x : T$

```
"switch (swfunc_T(x)) {
  ...
  caselabel_{C_i}:  y_{ij} = get_member_{C_i j}(x);   (j)
                    linear_dealloc_T(x);
                    B_K[[t_i]]
  ...
}"
```

$\mathrm{B}_K[\![(\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j)\ \overline{x}]\!] =$ where $|x| > 0$

"passign(fvars$[\![t_j]\!], \overline{x}$)
GENBODY$^{\mathrm{B}}_K[\![\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j]\!]$"

Note:

- We do not define $\mathrm{B}_K[\![\lambda x.\ t]\!]$ because a tail position cannot be a function after the argument completion.

## 5.9 Auxiliary Functions for Translation to C

$$
\mathrm{fvars}[\![t]\!] = \begin{cases}
\text{“}x;\ \mathrm{fvars}[\![u]\!]\text{”} & t = \lambda x.\ u \\
\mathrm{fvars}[\![t_j]\!] & t = \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \\
\text{“”} & \text{otherwise}
\end{cases}
$$

$\mathrm{fvars}'[\![f_i]\!] = \mathrm{fvars}[\![t_i]\!]$ for functions bounded by $\texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j$

$$
\mathrm{GENBODY}^{\mathrm{AT}}_K[\![t]\!] = \begin{cases}
\mathrm{GENBODY}^{\mathrm{AT}}_K[\![u]\!] & t = \lambda x.\ u \\
\text{“}\texttt{entry\_}f_i\texttt{:}\ \mathrm{GENBODY}^{\mathrm{AT}}_K[\![t_i]\!]\text{”} & t = \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \\
\quad \text{for } i = j, 1, \ldots, (j-1), (j+1), \ldots, |f| & \\
\mathrm{A}_K[\![t]\!] & \text{otherwise}
\end{cases}
$$

$$
\mathrm{GENBODY}^{\mathrm{AN}}[\![t]\!] = \begin{cases}
\mathrm{GENBODY}^{\mathrm{AN}}[\![u]\!] & t = \lambda x.\ u \\
\text{“}\texttt{entry\_}f_i\texttt{:}\ \mathrm{GENBODY}^{\mathrm{AN}}[\![t_i]\!]\text{”} & t = \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \\
\quad \text{for } i = 1, \ldots, |f| & \\
\mathrm{B}_K[\![t]\!] & \text{otherwise} \\
 & \text{where} \\
 & \quad t : T \\
 & \quad K(e) = \text{“}\texttt{*(}T\texttt{*)ret = }e\texttt{; return;}\text{”}
\end{cases}
$$

$$
\mathrm{GENBODY}^{\mathrm{B}}_K[\![t]\!] = \begin{cases}
\mathrm{GENBODY}^{\mathrm{B}}_K[\![u]\!] & t = \lambda x.\ u \\
\text{“}\texttt{entry\_}f_i\texttt{:}\ \mathrm{GENBODY}^{\mathrm{B}}_K[\![t_i]\!]\text{”} & t = \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \\
\quad \text{for } i = j, 1, \ldots, (j-1), (j+1), \ldots, |f| & \\
\mathrm{B}_K[\![t]\!] & \text{otherwise}
\end{cases}
$$

Note:

- fvars and fvars′ returns a list of variables: $x_1;\ldots;x_n;$. For simplicity, we omit ";" if not ambiguous.
- "$g(i)$" for $i = j_1,\ldots,j_n$ means "$g(j_1)\ \ldots\ g(j_n)$".

## 5.10 Translation for a Top-Level Function which is Translated to Multiple C Functions

$\text{GENFUN}^{\text{M}}[\![c]\!]$ translates the function (constant) $c$ with one or more auxiliary functions. We assume $c$ is defined as `Definition` $c := t$. The auxiliary functions $f_1 \ldots f_n$ are fixpoint bounded functions in $t$ which are invoked as functions. We assume the types of them:

$$c : T_{01} \to \cdots \to T_{0m_0} \to T_{00}$$
$$f_i : T_{i1} \to \cdots \to T_{im_i} \to T_{i0} \qquad i = 1 \ldots n$$

where $T_{i0}$ are inductive types $(i = 0 \ldots n)$

The formal arguments of $c$ are $x_{01} \ldots x_{0m_0} = \text{fvars}[\![t]\!]$ and the formal arguments of $f_i$ are $x_{i1} \ldots x_{im_i} = \text{fvars}'[\![f_i]\!]$.

$f_i$ invocation in C needs extra arguments, $\text{EXARGS}(f_i) = y_{i1}{:}U_{i1} \ldots y_{il_i}{:}U_{il_i}$, addition to the actual arguments in Gallina application because the free variables of the fixpoint should also be passed. If the free variables contain a function bounded by an outer fixpoint, the function itself is not passed but the free variables of the outer fixpoint are also passed. We iterate it until no fixpoint functions.

$\text{GENFUN}^{\text{M}}[\![c]\!]$ = "enum_entries$[\![c]\!]$ arg_structdefs$[\![c]\!]$ forward_decl$[\![c]\!]$ entry_functions$[\![c]\!]$ body_function$[\![c]\!]$"

enum_entries$[\![c]\!]$ = "`enum enum_func_`$c$ `{func_`$c$`,`$\overline{\texttt{func\_}f_i}^{i=1\ldots n}$`};`"

arg_structdefs$[\![c]\!]$ = "main_structdef$[\![c]\!]$ $\overline{\text{aux\_structdef}[\![c]\!]_i}^{i=1\ldots n}$"

main_structdef$[\![c]\!]$ = "`struct arg_`$c$ `{` $\overline{T_{0j}\ \texttt{arg}j\texttt{;}}^{j=1\ldots m_0}$ `};`"

aux_structdef$[\![c]\!]_i$ = "`struct arg_`$f_i$ `{` $\overline{U_{ij}\ \texttt{exarg}j\texttt{;}}^{j=1\ldots l_i}$ $\overline{T_{ij}\ \texttt{arg}j\texttt{;}}^{j=1\ldots m_i}$ `};`"

forward_decl$[\![c]\!]$ = "`static void body_function_`$c$`(enum enum_func_`$c$ `g,void *arg,void *ret);`"

entry_functions$[\![c]\!]$ = "main_function$[\![c]\!]$ $\overline{\text{aux\_function}[\![c]\!]_i}^{i=1\ldots n}$"

main_function$[\![c]\!]$ = "`static` $T_{00}$ `c(`$\overline{T_{0j}\ x_{0j}\texttt{,}}^{j=1\ldots m_0}$`) {`
`struct arg_`$c$ `arg = {`$\overline{x_{0i}\texttt{,}}^{i=1\ldots m_0}$`};` $T_{00}$ `ret;`
`body_function_`$c$`(func_`$c$`,&arg,&ret);return ret;`
`}`"

aux_function$[\![c]\!]_i$ = "`static` $T_{i0}$ $f_i$`(`$\overline{U_{ij}\ y_{ij}\texttt{,}}^{j=1\ldots l_i}$ $\overline{T_{ij}\ x_{ij}\texttt{,}}^{j=1\ldots m_i}$`) {`
`struct arg_`$f_i$ `arg = {`$\overline{y_{ij}\texttt{,}}^{j=1\ldots l_i}$ $\overline{x_{ij}\texttt{,}}^{j=1\ldots m_i}$`};` $T_{i0}$ `ret;`
`body_function_`$c$`(func_`$f_i$`,&arg,&ret); return ret;`
`}`"

body_function$[\![c]\!]$ = "`static void body_function_`$c$`(enum enum_func_`$c$ `g,void *arg,void *ret) {`
*decls*
`switch (g) {` $\overline{\text{aux\_case}[\![c]\!]_i}^{i=1\ldots n}$ main_case$[\![c]\!]$ `}`
$\text{GENBODY}^{\text{B}}_K[\![t]\!]$
`}`"

$$\text{aux\_case}[\![c]\!]_i = \text{“}\texttt{case func\_}f_i\texttt{:}\ \overline{y_{ij}\texttt{ = ((struct arg\_}f_i\texttt{ *)arg)->exarg}j\texttt{;}}^{j=1\ldots l_i}\ \overline{x_{ij}\texttt{ = ((struct arg\_}f_i\texttt{ *)arg)->arg}j\texttt{;}}^{j=1\ldots m_i}\ \texttt{goto entry\_}f_i\texttt{;”}$$

$$\text{main\_case}[\![c]\!] = \text{“}\texttt{default:;}\ \overline{x_{0j}\texttt{ = ((struct arg\_}c\texttt{ *)arg)->arg}j\texttt{;}}^{j=1\ldots m_0}\text{„}$$

where *decls* is local variable declarations for variables used in $\text{GENBODY}^{\text{B}}_{K}[\![t]\!]$.
$K(e) = \text{“}\texttt{*(}T_{00}\texttt{*)ret = }e\texttt{;return;”}$

## 5.11 Translation for a Top-Level Function which is Translated to a Single C Function

$\text{GENFUN}^{\text{S}}[\![c]\!]$ translates the function (constant) $c$ to a single C function.

$$\text{GENFUN}^{\text{S}}[\![c]\!] = \text{“}\texttt{static }T_0\ c\texttt{(}\ \text{fargs}'[\![t]\!]\texttt{)\ \{}\ \textit{decls}\ \text{GENBODY}^{\text{B}}_{K}[\![t]\!]\ \texttt{\}}\text{”}$$

where $c$ is defined as $\texttt{Definition}\ c : T_1 \to \cdots \to T_n \to T_0 := t$.
$T_0$ is an inductive type
*decls* is local variable declarations for variables used in $\text{GENBODY}^{\text{B}}_{K}[\![t]\!]$ excluding $\text{fargs}[\![t]\!]$.
$K(e) = \text{“}\texttt{return }e\texttt{;”}$

$$\text{fargs}[\![t]\!] = \begin{cases} \text{“}T\ x\texttt{,}\ \text{fargs}[\![u]\!]\text{”} & t = \lambda x{:}T.\ u \\ \text{fargs}[\![t_j]\!] & t = \texttt{fix}\ \overline{f := t}\ \texttt{for}\ f_j \\ \text{“”} & \text{otherwise} \end{cases}$$

$\text{fargs}'[\![t]\!] = \text{fargs}[\![t]\!]$ without the trailing comma

## 5.12 Translation for Top-Level Function

$$\text{GENFUN}[\![c]\!] = \begin{cases} \text{GENFUN}^{\text{M}}[\![c]\!] & t \text{ needs multiple functions} \\ \text{GENFUN}^{\text{S}}[\![c]\!] & \text{otherwise} \end{cases}$$

where $c$ is defined as $\texttt{Definition}\ c := t$.

# 6 Verification of Gallina-to-Gallina Transformations

Codegen verifies each step of the Gallina-to-Gallina transformations.

Since all transformations except match-app are convertible, Codegen checks convertibility of such transformations.

Assuming $t = (\texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{y} \Rightarrow t}\ \texttt{end}\ z)$, $u = (\texttt{match}\ x\ \texttt{with}\ \overline{C\ \overline{y} \Rightarrow t\ \underline{z}}\ \texttt{end})$, and some context $K$, match-app transformation is $K[t] \triangleright K[u]$. The match-app redex, $t$, is a subterm of a whole function, $K[t]$. Free variables in $t$ and $u$ are bounded by $K$.

Codegen verifies match-app transformation by proving $t = u$ internally. The proof term is not visible from a user.

Codegen does not verify $K[t] = K[u]$ because (1) this verification does not add big guarantee over above because we know same context $K$ is used in LHS and RHS, (2) induction is required if $x$ is a decreasing argument of fixpoint in $K$ and it is difficult to automate.

# References

[1] Guy L Steele Jr. It's time for a new old language. In *PPOPP*, page 1, 2017. https://labs.oracle.com/pls/apex/f?p=94065:10:175964141145:4986.

[2] The Coq Development Team. The coq reference manual: Release 8.12.0. 2020.