LocTypeLang: a layout description language

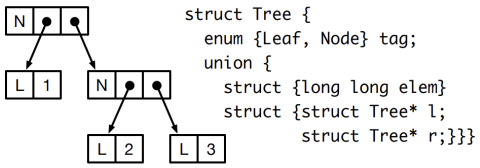
Dongxu Zhou

Peking University
School of Mathematical Sciences

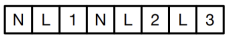
1 Overview

Contents

1.1 Motivation



(a) Standard representation of a tree structure in C: by default, word-sized tags *and* pointers.



(b) Serialized version of the same tree. Not to scale: tags take one byte and integers eight.

Figure 1.1: Standard and serialized representations of trees

Figure 1: Two kinds of Tree Representations

As shown in Figure 1, sometimes we need to store a tree as a serialized representation and operate on this representation to achieve better performance. However, the C implementation for this task can be error-prone and cumbersome, especially when using pointers to operate on the serialized tree.

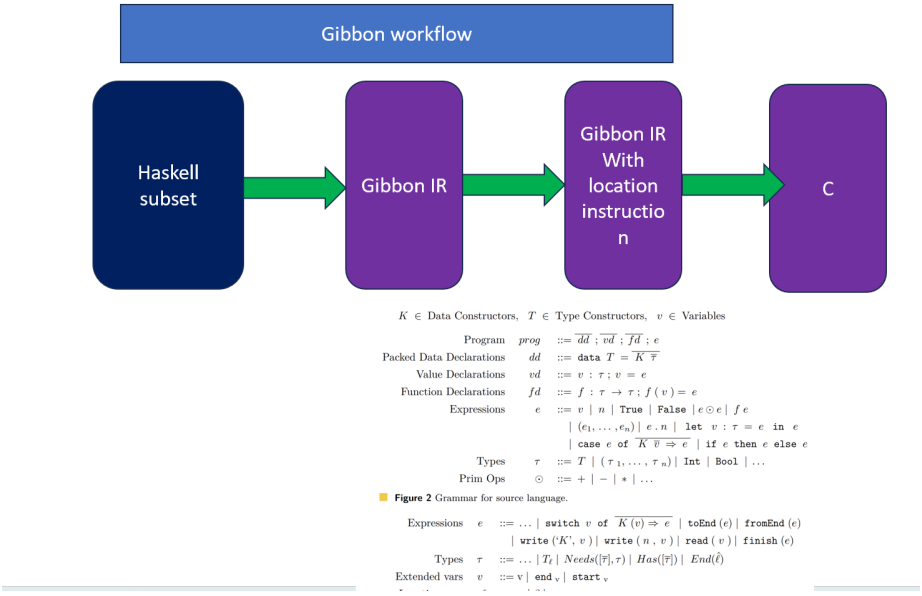
To address this issue, Can we implement the same task using a type-safe language like Haskell, as shown below? By using Haskell's strong type system, we can ensure that the serialized tree is always well-formed and avoid common errors that can occur in C. Additionally, with proper optimization techniques, we want to achieve performance similar to that of C while still maintaining type safety.

```
data Tree = Leaf Int | Node Tree Tree
 sum : : Tree -> Int
 sum t = case t of
         Leaf (n : Int) => n,
         Node (a : Tree) (b : Tree) => sum a + sum b
```

```
int sumPacked (byte * &ptr){
    int ret = 0 ;
     if(*ptr == LEAF){
        ptr++;
        ret = *(int*)ptr;
        ptr += sizeof(int);
    }else{
        ptr++;
```

```
        ret = sumPacked(ptr);
        ret += sumPacked(ptr);
    }
    return ret;
}
```

1.2 Previous Work



$K \in$ Data Constructors, $T \in$ Type Constructors, $v \in$ Variables

Program	$prog$	$::= \overline{dd}\ ;\ \overline{vd}\ ;\ \overline{fd}\ ;\ e$
Packed Data Declarations	dd	$::= \texttt{data}\ T = \overline{K\ \tau}$
Value Declarations	vd	$::= v : \tau ; v = e$
Function Declarations	fd	$::= f : \tau \to \tau ; f(v) = e$
Expressions	e	$::= v \mid n \mid \texttt{True} \mid \texttt{False} \mid e \odot e \mid f\ e$ $\mid (e_1, \ldots, e_n) \mid e.n \mid \texttt{let}\ v : \tau = e\ \texttt{in}\ e$ $\mid \texttt{case}\ e\ \texttt{of}\ \overline{K\ v \Rightarrow e} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
Types	τ	$::= T \mid (\tau_1, \ldots, \tau_n) \mid \texttt{Int} \mid \texttt{Bool} \mid \ldots$
Prim Ops	$\odot$	$::= + \mid - \mid * \mid \ldots$

Figure 2 Grammar for source language.

Expressions	e	$::= \ldots \mid \texttt{switch}\ v\ \texttt{of}\ \overline{K\ (v) \Rightarrow e} \mid \texttt{toEnd}\ (e) \mid \texttt{fromEnd}\ (e)$ $\mid \texttt{write}\ (\text{'}K\text{'}, v) \mid \texttt{write}\ (n, v) \mid \texttt{read}\ (v) \mid \texttt{finish}\ (e)$
Types	τ	$::= \ldots \mid T_t \mid Needs(\overline{\tau}, \tau) \mid Has(\overline{\tau}) \mid End(\hat{\ell})$
Extended vars	v	$::= \text{v} \mid \texttt{end}_{\text{v}} \mid \texttt{start}_{\text{v}}$

Figure 2: Gibbon workflow

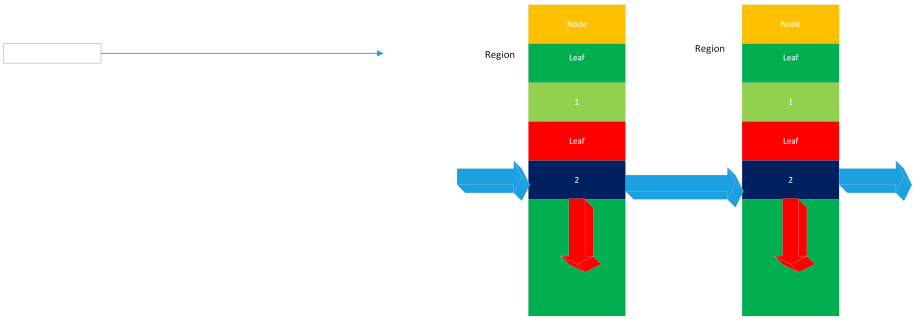


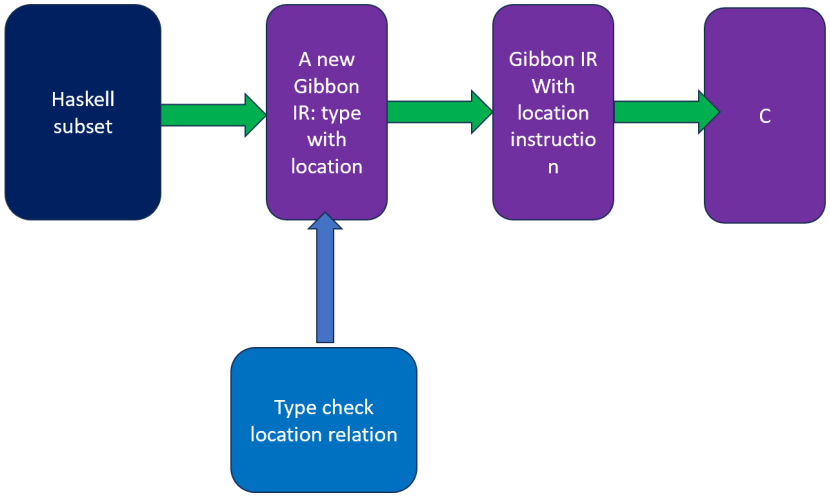
Figure 3: Gibbon memory model

Gibbon workflow : Translate subset of Haskell into Gibbon IR, then into C code.

Gibbon memory model:

1. Algebric data types are stored as serialized representation within a region
2. There are a lot of regions and each region consists of an array of cells, which contain store values (data constructor tags)
3. Each cell in a region is only initialized and written once
4. The locations of all the fields of the constructor must also match the expected constraints. That is, the location of the first field should be immediately after the constructor tag, and there should be appropriate after constraints for other fields in the location constraint environment

2 LocTypeLang: Add location information to Type



2.1 Syntax

$K \in$ Data Constructors, $\tau \in$ Type Constructors,
$x, y, f \in$ Variables, $l, l^r \in$ Location Variables,
$r \in$ Regions, $i, j \in$ Region Indices,
$\langle r, i \rangle^l \in$ Concrete Locations

Top-Level Programs top $::= \overrightarrow{dd}; \overrightarrow{fd}; e$
Datatype Declarations dd $::=$ data $\tau = \overrightarrow{K\hat{\tau}}$
Function Declarations fd $::= f : ts; f\vec{x} = e$
Located Types $\hat{\tau}$ $::= \tau a t l^r$
Type Scheme ts $::= \forall_{l^r} \cdot \overrightarrow{\hat{\tau}} \to \hat{\tau}$
Values v $::= x \mid \langle r, i \rangle^{l^r}$
Expressions e $::= v$ x
$\mid f \left[\overrightarrow{l^r}\right] \vec{v}$
$\mid K l^r \vec{v}$
$\mid$ let $x : \hat{\tau} = e$ in e
$\mid$letloc $l^r = le$ in e
$\mid$letregion r in e
$\mid$case v of $\overrightarrow{pat}$
Pattern pat $:= K(\overrightarrow{x : \hat{\tau}}) \to e$
Location Expressions le $::=$ (start r)
$\mid (l^r + 1)$
$\mid$ (after $\hat{\tau}$)

$\langle r, i \rangle^l$ consisting of a region, an index, and symbolic location corresponding to its binding site. The first two components are sufficient to fully describe an address in the store.

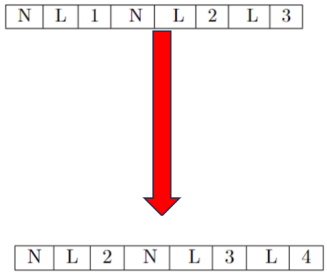
2.2 Design Idea

LocTypeLang is designed to augment Type in Gibbon to track locations within regions (e.g., byte offsets). The basic idea is to first establish what data share which logical memory regions, and in what order those data reside.

As below code, data constructor applications, such as Leaf 1, take an extra location argument, specifying where the data constructor should place the resulting value in memory:(Leaf l_r 1). This location became part of the type of the value Tree@l_r. Every location resides in a region, and when we want to name that region, we write add r to location l, namely, l_r.

```
data Tree  = Node Tree Tree | Leaf Int

buildtree : forall [l_r]. Int      Tree@la_r
buildtree [l_r] n =
    if n == 0 then (leaf l_r 1) -- write to output
    else
    let loc la_r = l_r + 1 in

    let left : Tree@la_r = buildtree [la_r] (n-1) in

    let loc lb_r = after (Tree@la_r) in

    let right : Tree@lb_r = buildtree [lb_r] (n-1) in
    -- write Node tag at l_r
    (Node l_r left right )

add1 :  forall [la_r,lb_r]. Tree@la_r      Tree@lb_r
add1 [la_r, lb_r ] t =
    case t of
    Leaf (n : Int) => Leaf (n+1 : Int)
    Node (a : Tree@la_r) (b : Tree@lb_r) =>
    Node (add1 [la_r, lb_r] a) (add1 [la_r, lb_r] b)
```

Build tree produces seriazation tree



Add1 produce another tree from seriazation tree aove

3 Contstrain Based Store typing

Contents

In Fig. 2.2, I extend the grammar with some extra details necessary for describing the type system. The typing rules for expressions are of the form as follows:

$$\Gamma; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$$

$$\begin{array}{ll}
\text{Typing Env.} & \Gamma \quad ::= \{x_1 \mapsto \hat{\tau}_1, \ldots, x_n \mapsto \hat{\tau}_n\} \\
\text{Store Typing} & \Sigma \quad ::= \{l_1^{r_1} \mapsto \tau_1, \ldots, l_n^{r_n} \mapsto \tau_n\} \\
\text{Constraint Env. } C & ::= \{l_1^{r_1} \mapsto le_1, \ldots, l_n^{r_n} \mapsto le_n\} \\
\text{Allocation Pointers } A & ::= \{r_1 \mapsto ap_1, \ldots, r_n \mapsto ap_n\} \\
\text{Nursery } N & ::= \{l_1^{r_1}, \ldots, l_n^{r_n}\}
\end{array}$$

The five letters to the left of the turnstile are different environments.

Γ is a standard typing environment.

Σ is a store-typing environment, which maps all materialized symbolic locations to their types. That is, every location in Σ has been written and contains a value of type $\Sigma\,(l^r)$.

C is a constraint environment, which keeps track of how symbolic locations relate to each other. e.g.

$C' = C \cup \{l^r \mapsto (\text{ after } \tau'@{l_1}^r)\}$

A maps each region in scope to a location, and is used to symbolically track the allocation and incremental construction of data structures; A can be thought of as representing the focus within a region of the computation.

N is a nursery of all location VARs that have been allocated, but not yet used. Locations are removed from N upon being written to, as the purpose is to prevent multiple writes to a location.

$$\begin{array}{ll}
\text{Store } S & ::= \{r_1 \mapsto h_1, \ldots, r_n \mapsto h_n\} \\
\text{Heap} \quad h & ::= \{i_1 \mapsto K_1, \ldots, i_n \mapsto K_n\} \\
\text{Location Map } M & ::= \{l_1^{r_1} \mapsto \langle r_1, i_1 \rangle, \ldots, l_n^{r_n} \mapsto \langle r_n, i_n \rangle\}
\end{array}$$

S. The store is a map from regions to heaps H, where each heap consists of an array of cells, which contain store values (data constructor tags. e.g. Tree).

M , a map from location var to concrete locations,(r, i).

3.0.1 Typing Rule

T-LETREGION

$$\frac{\Gamma; \Sigma; C; A'; N \vdash A''; N'; e : \hat{\tau}}{\Gamma; \Sigma; C; A; N \vdash A''; N'; \texttt{letregion}\ r\ \texttt{in}\ e : \hat{\tau} \qquad \texttt{where}\ A' = A \cup \{r \mapsto \varnothing\}}$$

T-VAR

$$\frac{\Gamma(x) = \tau @ l^r \quad \Sigma\,(l^r) = \tau}{\Gamma; \Sigma; C; A; N \vdash A; N; x : \tau @ l^r}$$

T-CONCRETE-LOC

$$\frac{\Sigma\,(l^r) = \tau}{\Gamma; \Sigma; C; A; N \vdash A; N; \langle r, i \rangle^l : \tau @ l^r}$$

The T-VAR rule ensures that the variable is in scope, and the symbolic location of the variable has been written to. T-CONCrete-LOC is very similar, and also just ensures that the symbolic location has been written to. T-LET is straightforward, but note that along with Γ, it also extends Σ to signify that the location l has materialized.

In T-LetRegion, extending A with an empty allocation pointer brings the region r in scope, and also indicates that a symbolic location has not yet been allocated in this region.

T-LETLOC-START

$$\frac{A(r) = \varnothing \quad l^r \notin N'' \quad l^{r'} \neq l^r \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau' @ l^{r'}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \text{ letloc } l^r = (\text{ start } r) \text{ in } e : \tau' @ l^{r'}}$$

where $C' = C \cup \{l^r \mapsto (\text{start}\, r)\} \qquad A' = A \cup \{r \mapsto l^r\} \qquad N' = N \cup \{l^r\}$

T-LetLoc-TAG

$$\frac{A(r) = l'^r \quad l'^r \in N \quad l^r \notin N'' \quad l^r \neq l'^{r''} \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau'' @ l'^{r''}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \text{ letloc } l^r = (l'^r + 1) \text{ in } e : \tau'' @ l''''}$$

`where` $C' = C \cup \{l^r \mapsto (l'^r + 1)\} \qquad A' = A \cup \{r \mapsto l^r\} \qquad N' = N \cup \{l^r\}$

T-LETLOC-AFTER

$$\frac{A(r) = {l_1}^r \quad \Sigma\,({l_1}^r) = \tau' \quad {l_1}^r \notin N \quad l^r \notin N'' \quad l^r \neq l^{r'} \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau' @ l^{r'}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \text{letloc}\, l^r = (\text{ after } \tau' @ {l_1}^r) \text{ in } e : \tau' @ l^{r'}}$$

where $C' = C \cup \{l^r \mapsto (\text{ after } \tau' @ {l_1}^r)\} \qquad A' = A \cup \{r \mapsto l^r\}$
$N' = N \cup \{l^r\}$

T-LEtLoc-StART, T-LETLOC-TAG and TLetLoc-After are corresponding to three ways of allocating a new location in a region. A new location is either: at the start of a region, one cell after an existing location, or after the data structure rooted at an existing location. Additionally, in A, each region is mapped to either the right-most allocated symbolic location in that region (if it is unwritten), or to the symbolic location of the most recently materialized data structure. This mapping in A is used by the typing rules to ensure that: (1) T-LetLoc-StarT may only introduce a location at the start of a region once; (2) T-LETLoc-TAG may only introduce a location if an unwritten location has just been allocated in that region (to correspond to the tag of some soon-to-be-built data structure); and (3) T-LETLOC-AfTER may only introduce a location if a data structure has just been materialized at the end of the region, and the programmer wants to allocate after it.

T-DataConstructor

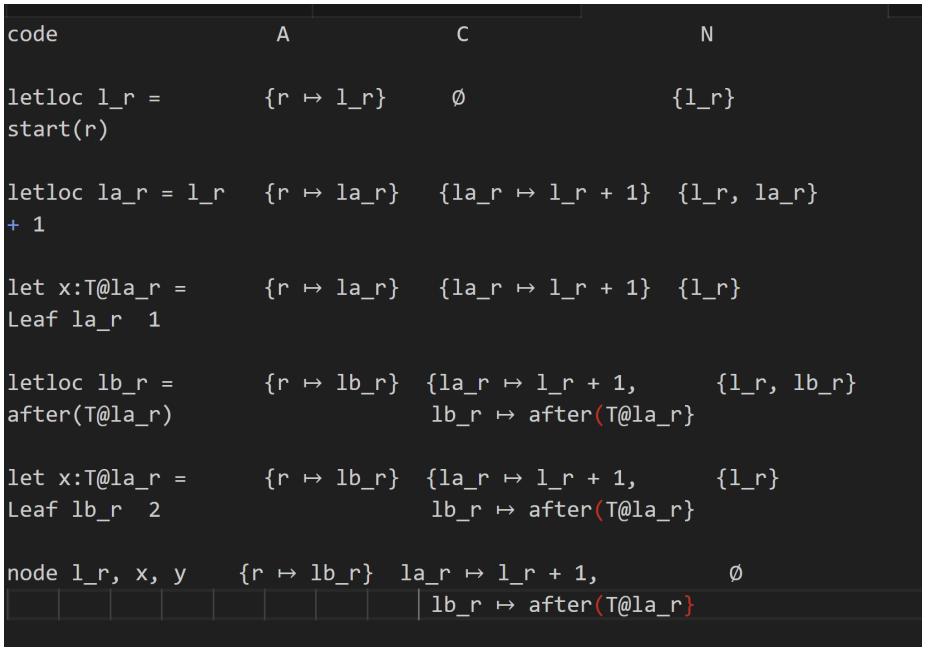
$$\frac{\begin{array}{c}\text{TypeOfCon}(K) = \tau \quad \text{TypeOfField } (K, i) = \overrightarrow{\hat{\tau}_i} \\ l^r \in N \quad A(r) = \overrightarrow{l_n}^r \text{ if } n \neq 0 \quad \text{else } l^r \\ C\left(\overrightarrow{l_1}^r\right) = l^r + 1 \quad C\left(\overrightarrow{l_{j+1}}^r\right) = \left(\text{after}\left(\overrightarrow{\tau_j'} @ \overrightarrow{l_j'}\right)\right) \\ \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau_i' @ l_i}^r\end{array}}{\begin{array}{c}\Gamma; \Sigma; C; A; N \vdash A'; N'; K l^r \vec{v} : \tau @ l^r \\ \text{where } A' = A \cup \{r \mapsto l^r\}; N' = N - \{l^r\} \\ n = |\vec{v}|; i \in I = \{1, \ldots, n\}; j \in I - \{n\}\end{array}}$$

T-LET

$$\frac{\Gamma; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1 @ {l_1}^{r_1} \qquad \Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \tau_2 @ {l_2}^{r_2}}{\begin{array}{c}\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{let } x : \tau_1 @ l_1^{r_1} = e_1 \texttt{ in } e_2 : \tau_2 @ l_2^{r_2} \\ \texttt{where } \Gamma' = \Gamma \cup \{x \mapsto \tau_1 @ l_1^{r_1}\}; \Sigma' = \Sigma \cup \left\{l_1^{r_1} \mapsto \tau_1\right\}\end{array}}$$

T-DATACONSTRUCTOR starts by ensuring that the tag being written and all the fields have the correct type. Along with that, the locations of all the fields of the constructor must also match the expected constraints. That is, the location of the first field should be immediately after the constructor tag, and there should be appropriate after constraints for other fields in the location constraint environment. After the tag has been written, the location l is removed from the nursery to prevent multiple writes to a location.

As demonstrated by T-DataConstructor, the type system enforces a particular ordering of writes to ensure the resulting tree is serialized in a certain order.



A simple demonstration of the type system is shown by the figure above, which tracks how A, C, and N change after each line in a simple expression that builds a binary tree with leaf children. Introducing l at the top establishes that it is at the beginning of r, A maps r to l, and N contains l. The location for the left sub-tree, l_a, is defined to be $+1$ after it, which updates r to point to l_a in A and adds a constraint to C for l_a. Actually constructing the Leaf in the next line removes l_a to N , because it has been written to.

Once l_a has been written, the next line can introduce a new location l_b after it, which updates the mapping in A and adds a new constraint to C . Once l_b has been written and removed from N in the next line, the final Node can be constructed, which expects the constraints to establish that l is before l_a, which is before l_b.

T-PATTERN

$$\frac{\text{TypeOfCon}\ (K) = \tau'' \quad \text{ArgTysOfConstructor}\ (K) = \overrightarrow{\tau'} \quad \Sigma\left(l^r\right) = \tau \quad l^r \neq \overrightarrow{l_i^{r'}} \quad \Gamma'; \Sigma'; C; A; N \vdash A'; N'; e : \tau @ l^r}{\tau''; \Gamma; \Sigma; C; A; N \vdash_{\text{pat}} \ A'; N'; K\left(\overrightarrow{x : \tau' @ l^{r'}}\right) \rightarrow e : \tau @ l^r}$$

$$\Sigma' = \Sigma \cup \left\{\overrightarrow{l_1^{r'}} \mapsto \overrightarrow{\tau_1'}, \ldots, \overrightarrow{l_n^{r'}} \mapsto \overrightarrow{\tau_n'}\right\} \qquad i \in \{1, \ldots, n\}; n = \left|\overrightarrow{\tau'}\right| = \left|\overrightarrow{x : \tau' @ l'^r}\right|$$

T-CASE

$$\frac{\Gamma; \Sigma; C; A; N \vdash A; N; v : \tau' @ l^{r'} \qquad \tau'; \Gamma; \Sigma; C; A; N \vdash patA'; N'; \overrightarrow{pat_i} : \hat{\tau}}{\Gamma; \Sigma; C; A; N \vdash A'; N';\ \text{case}\ v\ \text{of}\ \overrightarrow{pat} : \hat{\tau}}$$

3.0.2 Operation Semantics

D-LETLOC-TAG

$$S; M;\ \text{letloc}\ l^r = l'^r + 1\ \text{in}\ e \Rightarrow S; M'; e \quad \text{where}\ M' = M \cup \left\{l^r \mapsto \langle r, i+1\rangle\right\}; \langle r, i\rangle = M\left(l'^r\right)$$

DataConstructor

$$S; M; K l^r \vec{v} \Rightarrow S'; M; \langle r, i\rangle^{l^r} \qquad \text{where}\ S' = S \cup \{r \mapsto (i \mapsto K)\}; \langle r, i\rangle = M\left(l^r\right)$$

D-LETLOC-START

$$S; M;\ \text{letloc}\ lr = (\ \text{start}\ r)\ \text{in}\ e \Rightarrow S; M\prime; e \qquad \text{where}\ M\prime = M \cup lr \mapsto \langle r, 0\rangle$$

D-LETLOC-AFTER

$$\frac{S; M;\ \text{letloc}\ l^r = (\ \text{after}\ \tau @ l_1^r)\ \text{in}\ e \Rightarrow S; M'; e \quad \text{where}\ M' = M \cup l^r \mapsto \langle r, j\rangle; \langle r, i\rangle = M\left(l_1^r\right) \qquad \tau; \langle r, i\rangle; S \vdash_{eW} \langle r, j\rangle}{}$$

D-LET-EXPR

$$\frac{S; M; e_1 \Rightarrow S'; M'; e_1' \quad e_1 \neq v}{S; M;\ \text{let}\ x : \hat{\tau} = e_1\ \text{in}\ e_2 \Rightarrow S'; M';\ \text{let}\ x : \hat{\tau} = e_1'\ \text{in}\ e_2}$$

D-CASE

$$S; M; \text{case}\ \langle r, i\rangle lr\ \text{of}\ \left[\ldots, K\left(\overrightarrow{x : \tau @ lr}\right) \rightarrow e, \ldots\right] \Rightarrow S; M\prime; e\left[\langle r, \vec{w}\rangle \overrightarrow{lr} \mid \vec{x}\right]$$

$$\text{where}\ M' = M \cup \overrightarrow{l_1^r} \mapsto \langle r, i+1\rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}}\rangle$$

$$\overrightarrow{\tau_1}; \langle r, i+1\rangle; S \vdash_{end} \langle r, \overrightarrow{w_1}\rangle \qquad \overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle$$

$$K = S(r)(i); j \in 1, \ldots, n-1; n = |\overrightarrow{x : \tau}|$$

```
code                 S                          M

letloc l_r =        {r ↦ ∅ }                   {l_r ↦ (r,0)}
start(r)

letloc la_r = l_r   {r ↦ ∅ }                 {l_r ↦ (r,0), la_r ↦ (r,1)}
+ 1

let x:T@la_r =      {r ↦ {1 ↦ leaf}}         {l_r ↦ (r,0), la_r ↦ (r,1)}
Leaf la_r  1

letloc lb_r =       {r ↦ {1 ↦ leaf}}         {l_r ↦ (r,0), la_r ↦ (r,1)}
after(T@la_r)

let x:T@la_r =      {r ↦ {1 ↦ leaf           {l_r ↦ (r,0), la_r ↦ (r,1),
Leaf lb_r  2              3 ↦ leaf}}          lb_r ↦ (r,2) }


node l_r, x, y    {r ↦ {1 ↦ leaf             {l_r ↦ (r,0), la_r ↦ (r,1),
                   3 ↦ leaf, 0 ↦ node}}          lb_r ↦ (r,2) }
```

$S = r \mapsto \varnothing$ and the location l_r maps to $\langle r, 0 \rangle$ in the location map.

After stepping past the first line, the D-LetLoc-Tag step has allocated a cell for the tag of the interior node and bound the location l_{a_r} to $\langle r, 1 \rangle$. After the next line, the D-DataConstructor transition writes a leaf node to the store at the address represented by l_{1_r}.

$S = r \mapsto 1 \mapsto \text{Leaf}$.

The second letloc obtains the starting address for the second leaf node by using the end witness of the previous leaf node. The write of the second leaf node appears in the store after the next line, leaving the following store:

$S = r \mapsto 1 \mapsto \text{Leaf}, 3 \mapsto \text{Leaf}$.

Finally, after the D-DataConstructor step taken for the last line, the store contains the finalized allocation:

$S = r \mapsto 0 \mapsto \text{Node}, 1 \mapsto \text{Leaf}, 3 \mapsto \text{Leaf}$.

4 Type Satety

4.1 Store well formedness

$\Sigma; C; A; N \vdash_{wf} M; S$
Definition
1. $(l^r \mapsto \tau) \in \Sigma \Rightarrow$

$$((l^r \mapsto \langle r, i_1 \rangle) \in M \wedge$$
$$\tau; \langle r, i_1 \rangle ; S \vdash_{ew} \langle r, i_2 \rangle)$$

2. $C \vdash_{wf_{cfc}} M; S$
3. $A; N \vdash_{wf_{ca}} M; S$
4. $\mathrm{dom}(\Sigma) \cap N = \varnothing$

$\tau; \langle r, i_1 \rangle ; S \vdash_{ew} \langle r, i_2 \rangle)$ mean the store address $\langle$r ,ie$\rangle$ is the position one after the last cell of the tree of type starting at $\langle$r ,is$\rangle$ in store S.

4.1.1 Constructor Application

definition: $C \vdash_{wf_{cfc}} M; S$
1.$(l^r \mapsto (\mathrm{start}\, r)) \in C \Rightarrow$

$$(l^r \mapsto \langle r, 0 \rangle) \in M$$

2. $(l^r \mapsto (l'^r + 1)) \in C \Rightarrow$

$$(l'^r \mapsto \langle r, i_l \rangle) \in M \wedge$$
$$(l^r \mapsto \langle r, i_l + 1 \rangle) \in M$$

3. $(l^r \mapsto (\mathrm{after}\, \tau @ l'^r r)) \in C \Rightarrow$

$$((l'^r \mapsto \langle r, i_1 \rangle) \in M \wedge$$
$$\tau; \langle r, i_1 \rangle ; S \vdash_{ew} \langle r, i_2 \rangle \wedge$$
$$(l^r \mapsto \langle r, i_2 \rangle) \in M)$$

Rule 1 specifies that, if a location corresponding to the first address in a region

is in the constraint environment, then there is a corresponding entry for that location in the location map. Rule 2 specifies that, if a location corresponding to the address one past a constructor tag is in the constraint environment, then there are corresponding locations for the address of the tag and the address after in the location map.

4.1.2 Safe Allocation

definition: $A; N \vdash_{wf_{ca}} M; S$
1. $((r \mapsto l^r) \in A \wedge l^r \in N) \Rightarrow$

$$((l^r \mapsto \langle r, i \rangle) \in M \wedge i \rangle \operatorname{MaxIdx}(r, S))$$

2. $((r \mapsto l^r) \in A \wedge (l^r \mapsto \langle r, i_s \rangle) \in M \wedge l^r \notin N \wedge \tau; \langle r, i_s \rangle ; S \vdash_{ew} \langle r, i_e \rangle) \Rightarrow$

$$i_e > \operatorname{Max} Idx(r, S)$$

3. $l^r \in N \Rightarrow$

$$((l^r \mapsto \langle r, i \rangle) \in M \wedge$$
$$(r \mapsto (i \mapsto K)) \notin S)$$

4. $(r \mapsto \varnothing) \in A \Rightarrow$

$$r \notin \operatorname{dom}(S)$$

Rule 1 requires that, if a location is in both the allocation and nursery environments, i.e., that address represents an in-flight constructor application, then there is a corresponding location in the location map and the address of that location is the highest address in the store.

Rule 2 requires that, if there is an address in the allocation environment and that address is fully allocated, then the address of that location is the highest address in the store.

Rule 3 requires that, if there is an address in the nursery, then there is a corresponding location in the location map, but nothing at the corresponding address in the store.

4.1.3 Fields Continuity

definition:

$$\tau; \langle r, i_s \rangle ; S \vdash_{ew} \langle r, i_e \rangle)$$

1. $S(r)\,(i_s) = K'$ such that

$$\operatorname{data} \tau = \overrightarrow{K_1 \vec{\tau}_1} | \ldots | K' \vec{\tau}' | \ldots | \overrightarrow{K_m \vec{\tau}_m}$$

2. $\overrightarrow{w_0} = i_s + 1$
3. $\overrightarrow{\tau_1}; \langle r, \overrightarrow{w_0} \rangle ; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle \wedge$

$$\overrightarrow{\tau'_{j+1}}; \langle r, \overrightarrow{w_j} \rangle ; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$$

where $j \in \{1, \ldots, n-1\}; n = \left| \overrightarrow{\tau'} \right|$
4. $i_e = \overrightarrow{w_n}$

Rule 1 requires that the first cell store a constructor tag of the appropriate type. Rule 3 specifies the address of the cell one past the tag. Rule 4 recursively specifies the positions of the constructor fields. Finally, Rule 2 specifies that the start of a field is the address one past the end of another field.

4.2 Type Safety Theorem

4.2.1 Progress lemma

if $\varnothing; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$ and $\Sigma; C; A; N \vdash_{wf} M; S$ then e value else $S; M; e \Rightarrow S'; M'; e'$

Proof: T-DataConstructor case
Because K l v is not a value, the proof obligation is to show that there is a rule in the dynamic semantics whose left-hand side matches the machine configuration S;M ; e.

The only rule that can match is D-DataConstructor, but to establish the match, there remains one obligation,which is obtained by inversion on D-DataConstructor. The particular obligation is to establish that $(r, i) = M(l_r)$, for some i. To obtain this result, we need to use the well formness of the store, given by the premise of this lemma, and in particular rule Safe Allocation.3.

But a precondition for using Safe Allocation 3 that the location is unwritten, i.e., $l_r N$. This precondition is satisfied by inversion on T-DataConstructor.

4.2.2 Preservation lemma

If $\varnothing; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$
and $\Sigma; C; A; N \vdash_{wf} M; S$
and $S; M; e \Rightarrow S'; M'; e'$
then for some $\Sigma' \supseteq \Sigma, C' \supseteq C$,

$$\varnothing; \Sigma'; C'; A'; N' \vdash A''; N''; e' : \hat{\tau}$$

and $\Sigma'; C'; A'; N' \vdash_{wf} M'; S'$
Proof:

D-LETLOC-TAG case:

$$S; M;\ \text{letloc}\ l^r = l'^r + 1\ \text{in}\ e \Rightarrow S; M'; e$$
$$\texttt{where}\ \ M' = M \cup \{l^r \mapsto \langle r, i+1\rangle\}\,; \langle r, i\rangle = M\,(l'^r)$$

The first of two proof obligations is to show that the result e of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma; C'; A'; N' \vdash A''; N''; e : \hat{\tau}$$

where $\hat{\tau} = \tau@l^r, A' = A \cup \{r \mapsto l^r\}$, and $N' = N \cup \{l^r\}$. This proof obligation follows straightforwardly by inversion on T-LetLoc-Tag. - The second obligation for this proof case is to show that

$$\Sigma; C'; A'; N' \vdash_{wf}\ M'; S$$

The individual requirements, labeled WF 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.
- Case (WF 2.2; 1):
for each $(l^r \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$(l'^r \mapsto \langle r, i_1\rangle) \in M' \wedge$$
$$\tau; \langle r, i_1\rangle\,; S \vdash_{ew} \langle r, i_2\rangle$$

By the well formedness of the store given in the premise of this lemma, the above already holds for the location environment M. The obligation discharges by inspecting the only new location in M', namely l^r, which is fresh and therefore cannot be in the domain of Σ.

- Case (WF 2.2; 2):

$$C' \vdash_{wf_{cfc}}\ M'; S$$

Of the requirements for this judgement, the only one that is not satisfied immediately by the well formedness of the store given in the premise of the lemma is requirement WF 2.2; 2 The specific requirement is to establish that

$$(l'^r \mapsto \langle r, i\rangle) \in M' \wedge$$
$$(l^r \mapsto \langle r, i+1\rangle) \in M',$$

which follows immediately by inversion on D-LetLoc-Tag. - Case (WF 2.2; 3):

$$A'; N' \vdash_{wf_{\text{ca}}}\ M'; S$$

* Case (WF 2.2.2; 1):

$$(l^r \mapsto \langle r, i+1\rangle) \in M' \wedge i + 1 > \text{MaxIdx}(r, S)$$

The first conjunct follows immediately from inversion on D-LetLoc-Tag. To establish the second, however, we first need to establish that the address corresponding to location l'^r is the highest index in the store S. To do so, we need to

appeal to the well formedness of the store given by the premise of this lemma. In particular, we need to use the same requirement we are trying to prove, namely WF 2.2.2;1, but in this case, instantiating for l'^r in the original location environment M. By inversion on T-LetLoc-Tag, we have that $A(r) = l'^r$ and $l'^r \in N$, and as a consequence of WF 2.2.2; 1,

$$(l'^r \mapsto \langle r, i\rangle) \in M \wedge i > \operatorname{Max}\operatorname{Idx}(r, S).$$

Using the second conjunct above, this case discharges immediately.
* Case (WF 2.2.2; 2):
This obligation discharges immediately because, by inversion on T-LetLoc-Tag, $l^r \in N'$.
* Case (WF 2.2.2; 3):
The proof obligation is to establish that, for any constructor tag K,

$$\begin{aligned}&((l^r \mapsto \langle r, i+1\rangle) \in M' \wedge\\ &(r \mapsto (i+1 \mapsto K)) \notin S)\end{aligned}$$

The first conjunct discharges by inversion on D-LetLoc-Tag, and the second as a consequence of having already established just above that $i+1 > \operatorname{MaxIdx}(r, S)$.
* Case (WF 2.2.2; 4): The proof obligation is to establish that, for each $(r \mapsto \varnothing) \in A'$, it is the case that $r \notin \operatorname{dom}(S)$. This case discharges because, from the premise of the lemma, this property holds for the original environment A and store S, and, by inversion on T-LetLoc-Tag, continues to hold for A' and S'.
- Case (WF 2.2; 4):

$$\operatorname{dom}(\Sigma) \cap N' = \varnothing$$

Because it is a bound location, $l \notin \operatorname{dom}(\Sigma)$, and by inversion on T-LetLoc-Tag, $l \in N'$, which discharges the obligation.

5 Implementation Guidance

5.1 Set Up

Set Up

- OS: Ubuntu 20.04

```
 $ sudo apt-get install libgc-dev libgmp-dev gcc-7 uthash-dev software-properties-common
 $ sudo add-apt-repository ppa:hvr/ghc && sudo apt update && sudo apt install ghc-9.0.1
cabal-install-3.4
 $export PATH="/opt/ghc/bin:$PATH"
```

- Install haskell stack

```
donzho@LAPTOP-7IE4DBQH ~ % stack --version
Version 2.9.3, Git revision 6cf638947a863f49857f9cfbf72a38a48b183e7e x86_64 hpack-0.35.1
```

- BUild steps
    1. run "stack build" from root directory(.\ddpl-final) % stack build

```
gibbon-0.2: unregistering (local file changes: README.md)
gibbon> configure (lib + exe)
Configuring gibbon-0.2...
gibbon> build (lib + exe)
Preprocessing library for gibbon-0.2..
Building library for gibbon-0.2..
[ 1 of 13] Compiling Gibbon.DynFlags
[ 2 of 13] Compiling Gibbon.Common
[ 3 of 13] Compiling Gibbon.Language.Syntax
```

2. run stack test from root directory

5.2 implementation

My implementation is divided into three parts, all of which are located in the ddpl-final/src/Gibbon/MyLocTypeLang directory. These parts include the syntax definition of LocTypeLang, the implementation of static semantics (i.e. typechecker), and the implementation of dynamic semantics (i.e. interpreter).


s > LocTypeCheck > Typecheck.hs > LocTypeCheck.Typecheck
LitE 1
expected = LocationTC "Expected after constant relationship" (DataConE (Var "l") "Node" [VarE (V
-- Pattern Matching case
-- letregion r in
-- letloc l = startof r in
-- letloc l1 = l + 1 in
-- let x = Leaf l1 1 in
-- letloc l2 = l1 + 1 in
-- let y = Leaf l2 2 in
-- let z = Node x y 1 in
-- case z of
-- [Leaf num:Int@lnum , node x:Tree@lnodex y:Tree@lnodey 0]
case_test6 :: Assertion
EasyCode: Explain
case_test6 = assertValue exp (IntTy,LocationTypeState {tsmap = M.fromList []})
where exp = Ext $ LetRegionE (VarR "r") Undefined Nothing $
Ext $ LetLocE "l" (StartOfLE (VarR "r")) $
Ext $ LetLocE "l1" (AfterConstantLE 1 "l") $
LetE ("x", [], PackedTy "Tree" "l1",
DataConE "l1" "Leaf" [LitE 1]) $
Ext $ LetLocE "l2" (AfterVariableLE "x" "l1" False) $
LetE ("y", [], PackedTy "Tree" "l2",
DataConE "l2" "Leaf" [LitE 2]) $
LetE ("z", [], PackedTy "Tree" "l",
DataConE "l" "Node" [VarE "x",
VarE "y"]) $
CaseE (VarE "z")
[("Leaf",[("num","lnum")], VarE "num")
, ("Node",[("x","lnodex"),("y","lnodey")],
LitE 0)]

5.3 examples

You can check the examples in tests/LocTypeCheck , which shows how these samples pass the Typechecker.