A Dynamic Region System for Transforming Effect-dependent Programs

Zhixuan Yang

SOKENDAI and NII

May 2019

Outline

Assuming you are an optimising compiler, will you transform

$$put\ p\ v; l \leftarrow get\ q; K \qquad \text{(In C syntax } *p = v; l = *q; K\text{)}$$

to

$$l \leftarrow get\ q; put\ p\ v; K \qquad \text{(In C syntax } l = *q; *p = v; K\text{)}$$

($put\ p\ v$ writes v to the cell p; $get\ q$ reads cell q)

Assuming you are an optimising compiler, will you transform

$$put\ p\ v; l \leftarrow get\ q; K \qquad \text{(In C syntax } *p = v; l = *q; K\text{)}$$

to

$$l \leftarrow get\ q; put\ p\ v; K \qquad \text{(In C syntax } l = *q; *p = v; K\text{)}$$

($put\ p\ v$ writes v to the cell p; $get\ q$ reads cell q)

Only when $put\ p$ and $get\ q$ access different cells ($p \neq q$) !

Pointer Analyses

We need a way to statically track which memory cells a program may access.

Two orthogonal problems:

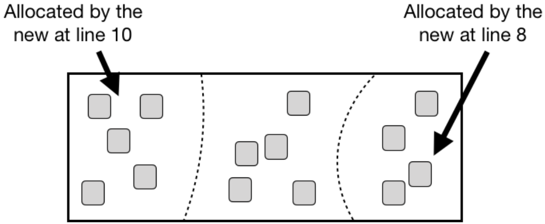
1. How to "talk about" memory cells statically—a static abstraction of memory.
   E.g. "p points to a cell allocated by the new at line 10".

2. How to track which memory cells a pointer variable may point to

Pointer Analyses in Compilers

Core of static program analysis, intensively studied

- Usually with simple model to question 1: partitioning the memory cells by the new call in the code allocating it



- Automated solution to question 2: data-flow analyses, abstract interpretation ...

Pointer Analyses Beyond Compilers

Program transformations are not limited to compilers.

- Equational proof: transforming a program to a simple and obviously correct program
- Done by human: allowing more refined memory models (that are not possible for automated analyses).

Pointer Analyses Beyond Compilers

Region system (Lucassen and Gifford 1988): the memory is partitioned into *regions* (mentally, by the programmer)

- Reference (pointer) type $\mathit{Ref}\ r\ D$ is indexed by a region
- $\mathit{new} : (r : \mathit{Region}) \rightarrow D \rightarrow \mathit{Ref}\ r\ D$ takes a region argument
- Use a type system to track which regions a program access

Compared to the line-number-based memory model, we can distinguish memory cells by passing different region argument to the same new call.

Naive model vs. region model

$$
\begin{array}{ll}
1 & \mathit{makeList}\ [\,] = \mathbf{return}\ \mathit{Nil} \\
2 & \mathit{makeList}\ (a : \mathbf{as}) = \{\, \mathit{ls} \leftarrow \mathit{makeList}\ \mathbf{as}; \\
3 & \qquad\qquad\qquad\qquad\quad p \leftarrow \mathit{new}\ (a, \mathit{ls}); \\
4 & \qquad\qquad\qquad\qquad\quad \mathbf{return}\ (\mathit{Ptr}\ p) \,\} \\
5 & l_1 \leftarrow \mathit{makeList}\ a_1 \\
6 & l_2 \leftarrow \mathit{makeList}\ a_2
\end{array}
$$

In the naive model, both l_1 and l_2 point to cells allocated by the new at line #3.

Naive model vs. region model

(Assuming $r_1 \neq r_2$)

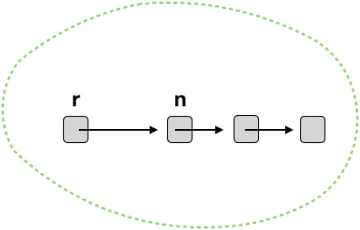
$$
\begin{aligned}
&\mathit{makeList}\ r_1\ [\,] = \mathbf{return}\ \mathit{Nil} \\
&\mathit{makeList}\ r_1\ (a : ax) = \{\, \mathit{ls} \leftarrow \mathit{makeList}\ r\ ax; \\
&\qquad\qquad\qquad\qquad\quad\ p \leftarrow \mathit{new}\ r\ (a, \mathit{ls}) \\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{return}\ (\mathit{Ptr}\ p) \,\} \\
&l_1 \leftarrow \mathit{makeList}\ r_1\ a_1 \\
&l_2 \leftarrow \mathit{makeList}\ r_2\ a_2
\end{aligned}
$$

In the region model, l_1 and l_2 point to cells in region r_1 and r_2 respectively.

Static region is not enough

- Existing region systems assume memory can be *statically* partitioned.
- For some pointer-manipulating programs, this assumption does not hold.

$traverse\ (Ptr\ r)$
$traverse\ (Ptr\ t)$

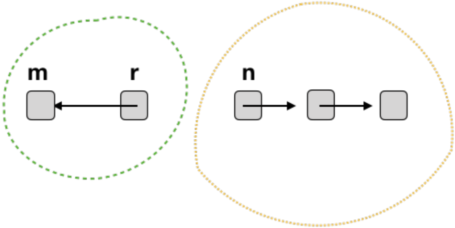


At some place in the code, we may want to put the list r in a region.

Static region is not enough

At some other point in the program, we may want to put the first cell of list r and the rest of the list in different regions to reason about the program.

$$
\begin{aligned}
&p\ \mathit{Nil} = \mathbf{return}\ ()\\
&p\ (\mathit{Ptr}\ r) = \{(a, n) \leftarrow \mathit{get}\ r\\
&\qquad\qquad\qquad \mathit{put}\ r\ (a, m)\\
&\qquad\qquad\qquad \mathit{traverse}\ n\\
&\qquad\qquad\qquad \mathit{put}\ r\ (a, n)\}
\end{aligned}
$$



Thus, a *static* region system is not sufficient.

A dynamic region system

We wish a more *dynamic* region system such that

- the region to which a cell belongs can determined dynamically (by the contents of memory cells)
- we can track the regions a program access

This is what this work wants to do.

A dynamic region system

Currently, my dynamic region system is very simple. A region is

- a single memory cell or
- the *current* reachable closure from a memory cell.

For example, the judgement

$$l : \mathit{ListPtr}\ a \vdash t\ l : \mathbf{1}\ !\, \{get_{r(l)}\}$$

asserts the program $t\ l$ only reads the linked list starting from l.

Separation guard

A complementary construct: *separation guard*

- it checks some pointers or their reachable closures are disjoint, otherwise terminates the program
- resembles the precondition of separation logic

For example,

$$l : ListPtr\ a, l_2 : Ref\ (a,\ ListPtr\ a) \vdash [r(l) * l_2] : \mathbf{F}\, Unit$$

checks the cell l_2 is not any node of linked list l.

Program transformation

Put dynamic region system and separation guard together, we can express and prove some transformations.

Example

$$\frac{l : \mathit{ListPtr}\ a \vdash t\ l : \mathbf{1}\ !\,\{\mathit{get}_{r(l)}\}}{[r(l) * l_2]; \mathit{put}\ l_2\ v; t\ l \quad = \quad [r(l) * l_2]; t\ l; \mathit{put}\ l_2\ v}$$

Outline

Language

For the purpose of discussion, we fix a small programming language based Levy's *call-by-push-value* calculus with algebraic effects (but no handlers)

Base types:	$\sigma ::= \mathit{Bool} \mid \mathit{Unit} \mid \mathit{Void} \mid \ldots$
Value types:	$A ::= \sigma \mid \mathit{ListPtr}\ D \mid \mathit{Ref}\ D \mid A_1 \times A_2$
	$\mid A_1 + A_2 \mid \mathbf{U}\underline{A}$
Storable types:	$D ::= \sigma \mid \mathit{ListPtr}\ D \mid \mathit{Ref}\ D \mid D_1 \times D_2$
	$\mid D_1 + D_2$
Computation types:	$\underline{A} ::= \mathbf{F}A \mid A_1 \to \underline{A_2}$

Syntax

Value terms: $v ::= x \mid c \mid \mathit{Nil} \mid \mathit{Ptr}\ v \mid (v_1,\ v_2)$

$\mid \mathbf{inj}_1^{A_1+A_2}\ v \mid \mathbf{inj}_2^{A_1+A_2}\ v \mid \mathbf{thunk}\ t$

Computation terms: $t ::= \mathbf{return}\ v \mid \{\, x : A \leftarrow t_1; t_2 \,\}$

$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\, (x_1, x_2) \rightarrow t \,\}$

$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\, \mathit{Nil} \rightarrow t_1, \mathit{Ptr}\ x \rightarrow t_2 \,\}$

$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\mathbf{inj}_1\ x_1 \rightarrow t_1, \mathbf{inj}_2\ x_2 \rightarrow t_2\}$

$\mid \lambda x : A.\ t \mid t\ v \mid \mathbf{force}\ v \mid \mathit{op}\ v \mid$

$\mu x : \mathbf{U}\underline{A}.\ t$

Operations: $\mathit{op} ::= \mathit{fail} \mid \mathit{get} \mid \mathit{put} \mid \mathit{new} \mid \ldots$

Effect theories

For the operations in the language, we have equations on them:

- $l \leftarrow get\ p; put\ p\ l = \mathbf{return}\ ()$
- $put\ p\ v_1; put\ p\ v_2 = put\ p\ v_2$
- ...
- And this *separation axiom*

$$
\begin{array}{lcl}
\{l_1 \leftarrow new_D\ v_1 & = & \{l_1 \leftarrow new_D\ v_1 \\
\ \ l_2 \leftarrow new_D\ v_2 & & \ \ l_2 \leftarrow new_D\ v_2 \\
\ \ \mathbf{match}\ l_1 \equiv l_2\ \mathbf{as} & & \ \ t_1\} \\
\qquad \{False \rightarrow t_1 & & \\
\qquad \ \ True \rightarrow t_2\}\} & &
\end{array}
$$

Type system and semantics

Type system

Please refer to Plotkin and Pretnar's *A Logic for Algebraic Effects* paper.

Semantics

$[\![A]\!]$ maps to some set.

$[\![\underline{A}]\!]$ maps to the set of tree whose internal nodes of operations and leaves are A-values.

$[\![\Gamma \vdash t : \tau]\!]$ maps to a function $[\![\Gamma]\!] \to [\![\tau]\!]$.

Logic

We have a first-order equational logic for reasoning about programs of this language.

Judgements

$$\Gamma \mid \Psi \vdash \phi$$

Γ is a context of the types of free variables and Φ is a list of formulas that are the premises of ϕ.

Terms

$$\begin{aligned} \phi ::= {} & t_1 =_{\texttt{CBPV}} t_2 \mid t_1 =_{\texttt{Alg}} t_2 \mid \forall x : A.\ \phi \mid \forall x : \underline{A}.\ \phi \\ & \mid \exists x : A.\ \phi \mid \exists x : \underline{A}.\ \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ & \mid \neg\phi \mid \phi_1 \to \phi_2 \mid \top \mid \bot \end{aligned}$$

Inference rules

The logic system's inference rules are:

- Standard rules for classical first order connectives and structural rules for judgements,

Inference rules

The logic system's inference rules are:

- standard β- and η- equivalence for CBPV language constructs, for example,

$$\frac{}{\{\, x \leftarrow \mathbf{return}\ v; t\,\} =_{\mathtt{CBPV}} t[v/x]} \qquad \frac{}{(\lambda x.\ t)\ v =_{\mathtt{CBPV}} t[v/x]}$$

$$\frac{}{\mathbf{case}\ \mathit{True}\ \mathbf{of}\ \{\, \mathit{True} \to t_1; \mathit{False} \to t_2\,\} =_{\mathtt{CBPV}} t_1}$$

Inference rules

The logic system's inference rules are:

- rules inherited from the effect theories, for example,

$$\frac{}{\{\, put\ (l_1, v_1); put\ (l_2, v_2); t\,\} =_{\texttt{Alg}} \{\, put\ (l_2, v_2); t\,\}}$$

Inference rules

The logic system's inference rules are:

- algebraicity of effect operations, an inductive principle over computations and a universal property of computation types.

Sum up of preliminaries

Plotkin's algebraic effects have these layers of concepts:

1. Basic types (e.g. Nat, $Bool$, $\dots$)
2. Effect theories (operations and equations)
3. A programming language with effect operations, sequential composition, and (possibly) handlers
4. An equational logic for reasoning about programs of this language.

Sum up of preliminaries

Plotkin's algebraic effects have these layers of concepts:

1. Basic types (e.g. Nat, $Bool$, $\ldots$)
2. Effect theories (operations and equations)
3. A programming language with effect operations, sequential composition, and (possibly) handlers
4. An equational logic for reasoning about programs of this language.

The logic is rich and powerful—we will express our region system in this logic.

Outline

Background

Preliminaries: effect theories, a language and a logic

Dynamic region system

Separation guards

Region system as logic predicates

Our dynamic region systems are defined as logic predicates on computation terms in the logic.

Let op range over possible effect operations in the language. We extend the term of the logic:

$$\phi ::= \ldots \mid t \,!\, \epsilon$$

$$\epsilon ::= \emptyset \mid \epsilon, op \mid \epsilon, get_{r(v)} \mid \epsilon, put_{r(v)} \mid \epsilon, get_v \mid \epsilon, put_v$$

Well-formedness

The new term is well-formed when

$$\frac{\Gamma \vdash t : \mathbf{F} A}{\Gamma \vdash t \ ! \cdot : \mathbf{form}} \qquad \frac{\Gamma \vdash t \ ! \ \epsilon : \mathbf{form}}{\Gamma \vdash t \ ! \ \epsilon, op : \mathbf{form}}$$

$$\frac{\Gamma \vdash t \ ! \ \epsilon : \mathbf{form} \qquad \Gamma \vdash v : \mathit{Ref}\ D}{\Gamma \vdash t \ ! \ \epsilon, o_v : \mathbf{form}} \ (o \in \{get,\ put\})$$

$$\frac{\Gamma \vdash t \ ! \ \epsilon : \mathbf{form} \qquad \Gamma \vdash v : \mathit{ListPtr}\ D}{\Gamma \vdash t \ ! \ \epsilon, o_{r(v)} : \mathbf{form}} \ (o \in \{get,\ put\})$$

Semantics

We intend the semantics of $[\![\Gamma \vdash t \,!\, \epsilon : \mathbf{form}]\!]$ to be

$$\{\gamma \in [\![\Gamma]\!] \mid [\![t]\!](\gamma) \text{ is a computation only invokes operations in } \epsilon\}$$

I am hesitating about the definition—should we interpret get/put in t?

Inference rules

$$\frac{}{\Gamma \mid \Psi \vdash \textbf{return}\ x \ !\ \emptyset}\ \textsc{R-Pure} \qquad \frac{\Gamma \mid \Psi \vdash t \ !\ \epsilon \qquad \epsilon \subseteq \epsilon'}{\Gamma \mid \Psi \vdash t \ !\ \epsilon'}\ \textsc{R-Sub}$$

$$\frac{\Gamma \mid \Psi \vdash t =_{\texttt{CBPV}} t' \ \wedge\ t' \ !\ \epsilon}{\Gamma \mid \Psi \vdash t \ !\ \epsilon}\ \textsc{R-Eq}$$

And for any $op : A \to \underline{B} \in \epsilon$ that is not $get/put_{r(v)}$,

$$\frac{\Gamma, a : B \mid \Psi \vdash k \ !\ \epsilon}{\Gamma \mid \Psi \vdash (a \leftarrow op\ v; k) \ !\ \epsilon}\ \textsc{R-Op}$$

Inference rules

Intuitively,
if ϵ contains $get_{r(v)}$ or $put_{r(v)}$, the program can read or write the cells linked from $v : ListPtr\ D$.

- When $v = Nil$, the program get no cells to access from $r(v)$.
- When $v = Ptr\ v'$, the program can read or write the cell v',
  - and if it reads it by $(a, n) \leftarrow get\ v'$, its allowed operation on $r(v)$ is inherited by $r(n)$ and v'

Inference rules

If $get_{r(v)} \in \epsilon$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash t\ Nil\ !\ \epsilon \setminus \{get_{r(v)},\ put_{r(v)}\} \\ \Gamma, v' \mid \Psi \vdash t\ (Ptr\ v') =_{\texttt{CBPV}} \{(a,n) \leftarrow get\ v'; k\} \\ \Gamma, v', a, n \mid \Psi,\ t\ n\ !\ \epsilon[r(n)/r(v)] \vdash k\ !\ \epsilon[r(n)/r(v)] \cup \epsilon[v'/r(v)]\end{array}}{\Gamma \mid \Psi \vdash t\ v\ !\ \epsilon}$$

If $put_{r(v)} \in \epsilon$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash t\ Nil\ !\ \epsilon \setminus \{get_{r(v)},\ put_{r(v)}\} \\ \Gamma \mid \Psi \vdash t\ (Ptr\ v') =_{\texttt{CBPV}} \{put\ v'\ c; k\} \qquad \Gamma \mid \Psi \vdash k\ !\ \epsilon[v'/r(v)]\end{array}}{\Gamma \mid \Psi \vdash t\ v\ !\ \epsilon}$$

Theorem (Soundness)

If $\Gamma \mid \Psi \vdash t \,!\, \epsilon$, *then* $[\![\Psi]\!] \subseteq [\![t \,!\, \epsilon]\!]$.

Proof.

To do $\square$

Outline

Separation guards

- Our dynamic region systems proves a program only operates on certain memory cells

$$t_1 \; ! \{get_{r(l_1)}, put_{r(l_1)}\} \quad \text{and } t_2 \; ! \{get_{r(l_2)}, put_{r(l_2)}\}$$

- This information is useful only when we can also shows the cells that two programs respectively operates on are disjoint

$$r(l_1) \cap r(l_2) = \emptyset \implies t_1; t_2 = t_2; t_1$$

- Ultimately, disjointness comes from the separation axiom of new, but it is too primitive for practical use.

Separation guards

- Our dynamic region systems proves a program only operates on certain memory cells

$$t_1 \; ! \, \{get_{r(l_1)}, put_{r(l_1)}\} \quad \text{and } t_2 \; ! \, \{get_{r(l_2)}, put_{r(l_2)}\}$$

- This information is useful only when we can also shows the cells that two programs respectively operates on are disjoint

$$r(l_1) \cap r(l_2) = \emptyset \implies t_1; t_2 = t_2; t_1$$

- Ultimately, disjointness comes from the separation axiom of new, but it is too primitive for practical use.

We introduce *separation guards* for tracking disjointness more easily at a higher level.

Following separation logic, we write $\phi = l_1 * l_2 * \cdots * l_n$ to denote that cells described by l_i are dijoint.

Each l_i can be either a value v of type $Ref\ D$, or $r(v)$ for a value v of type $ListPtr\ D$.

A separation guard $[\phi]$ is a computation of type $\mathbf{F}\,Unit$.

Definition

$[\phi]$ is defined as

$$[\phi] = sepChk\ \phi\ \emptyset$$

$$sepChk\ [\,]\ s = \mathbf{return}\ ()$$
$$sepChk\ (v * \phi)\ x = \mathbf{if}\ l \in x\ \mathbf{then}\ fail\ \mathbf{else}\ sepChk\ \phi\ (x\ \cup\ l)$$
$$sepChk\ (r(v) * \phi)\ x = \{x' \leftarrow chkList\ v\ x; sepChk\ \phi\ x'\}$$

$$chkList\ Nil\ x = \mathbf{return}\ x$$
$$chkList\ (Ptr\ p)\ x = \mathbf{if}\ p \in x$$
$$\mathbf{then}\ \{fail; \mathbf{return}\ ()\}$$
$$\mathbf{else}\ \{(\_, n) \leftarrow get\ p; chkList\ n\ (x\ \cup\ p)\}$$

$[\phi]$ checks regions l_i of ϕ are distinct.

Inference rules

Define $c\langle\ a =_{\texttt{Alg}} b\ \rangle$ to be $(c; a) =_{\texttt{Alg}} (c; b)$.

We have the following inference rules for separation guards

$$\frac{}{[\phi]\langle\ \mathit{new}\ t =_{\texttt{Alg}} (l \leftarrow \mathit{new}\ t; [\phi * l]; \mathbf{return}\ l)\ \rangle}$$

$$\frac{\Gamma \mid \Psi, l_1 \neq l_2 \vdash t_1 =_{\texttt{Alg}} t_2}{\Gamma \mid \Psi \vdash [l_1 * l_2]\langle\ t_1 =_{\texttt{Alg}} t_2\ \rangle}$$

$$\frac{}{\mathbf{return}\ \mathit{Nil} =_{\texttt{Alg}} (l \leftarrow \mathbf{return}\ \mathit{Nil}; [r(l)]; \mathbf{return}\ l)}$$

$$\frac{}{[r(l)]\langle\ \mathit{new}\ (\mathit{Cell}\ a\ l) =_{\texttt{Alg}} (l' \leftarrow \mathit{new}\ (\mathit{Cell}\ a\ l); [r(l')]; \mathbf{return}\ l')\ \rangle}$$

Inference rules

The rule for list induction

$$\frac{\texttt{base case} \qquad \texttt{inductive case}}{\Gamma \mid \Psi \vdash [r(l) * \phi]\langle\, t_1 =_{\texttt{Alg}} t_2 \,\rangle} \quad (\textsc{ListInd})$$

- $\texttt{base case}$ is $\Gamma \mid \Psi, l =_{\texttt{Alg}} \mathit{Nil} \vdash [\phi]\langle\, t_1 =_{\texttt{Alg}} t_2 \,\rangle$
- $\texttt{inductive case}$ is

$$\Gamma \mid l = \mathit{Ptr}\ l', \texttt{hyp} \vdash \\ \big((\mathit{Cell}\ \_, n) \leftarrow \mathit{get}\ l'; [l' * r(n) * \phi]\big)\,\langle\, t_1 =_{\texttt{Alg}} t_2 \,\rangle$$

and $\texttt{hyp} =_{\texttt{def}} [r(n) * \phi]\langle\, t_1 =_{\texttt{Alg}} t_2 \,\rangle$

Soundness

Theorem

The inference rules for separation guards are sound.

Proof.

To do. It'll be a large verifying proof. $\square$

Program transformations

Now we have enough ingredients to express the program transformations I wanted:

A frame rule:

$$\frac{\Gamma \mid \Psi \vdash t_1 \ ! \ \overline{\phi_1} \qquad \Gamma \mid \Psi \vdash [\phi_1]\langle\, t_1 =_{\texttt{Alg}} t_2 \,\rangle}{\Gamma \mid \Psi \vdash [\phi_1 * \phi_2]\langle\, t_1 =_{\texttt{Alg}} (t_2; [\phi_2]) \,\rangle}$$

Commutativity lemma

$$\frac{\Gamma \mid \Psi \vdash t_i \ ! \ \overline{\phi_i} \ (i = 1, 2)}{\Gamma \mid \Psi \vdash [\phi_1 * \phi_2]\langle\, (t_1; t_2) =_{\texttt{Alg}} (t_2; t_1) \,\rangle}$$

Program transformations

Now we have enough ingredients to express the program transformations I wanted:

A frame rule:

$$\frac{\Gamma \mid \Psi \vdash t_1 \; ! \; \overline{\phi_1} \qquad \Gamma \mid \Psi \vdash [\phi_1]\langle \, t_1 =_{\texttt{Alg}} t_2 \, \rangle}{\Gamma \mid \Psi \vdash [\phi_1 * \phi_2]\langle \, t_1 =_{\texttt{Alg}} (t_2; [\phi_2]) \, \rangle}$$

Commutativity lemma

$$\frac{\Gamma \mid \Psi \vdash t_i \; ! \; \overline{\phi_i} \; (i = 1, 2)}{\Gamma \mid \Psi \vdash [\phi_1 * \phi_2]\langle \, (t_1; t_2) =_{\texttt{Alg}} (t_2; t_1) \, \rangle}$$

Proof: induction on the derivation of region predicates and using the inference rules of separation guards.

Conclusion

- We proposed static methods to track
  - which memory cells a program access, and
  - if these cells are disjoint.
- Technical parts are remained to be completed and polished.