Equational Reasoning about Pointer Programs with Separation

May 31, 2019

No Institute Given

Abstract. {Zhixuan: This abstract is not very accurate. You can ignore it.} The equational theories of algebraic effects are natural tools for reasoning about programs using the effects, and some of the theories are proved to be complete, including the one of local state—the effect of mutable memory cells with dynamic allocation. Although being complete, reasoning about large programs with only a small number of equational axioms can sometimes be cumbersome and unscalable, as exposed in a case study of using the theory of local state to equationally reason about the Schorr-Waite traversal algorithm. Motivated by the recurring patterns in the case study, this papers proposes a conservative extension to the theory of local state called *separation guards*, which is used to assert the disjointness of memory cells and allows local equational reasoning as in separation logic.

Keywords: Equational Reasoning · Effect systems · Program transformation · Pointer programs · Algebraic effects

1 Introduction

Plotkin and Power's algebraic effects [10,11] and their handlers [12,13] provide a uniform foundation for a wide range of computational effects by defining an effect as an algebraic theory—a set of operations and equational axioms on them. The approach has proved to be successful because of its composability of effects and clear separation between syntax and semantics. Furthermore, the equations defining an algebraic effect are also natural tools for equational reasoning about programs using the effect, and can be extended to a rich equational logic [13,9]. The equations of some algebraic effects are also proved to be (Hilbert-Post) complete, including the effects of global and local state [16].

However, if one is limited to use only equational axioms on basic operations and must always expand the definition of a program to the level of basic operations, this style of reasoning will not be scalable. A widely-studied solution is to use an *effect system* to track possible operations used by a program and use this information to derive equations (i.e. transformations) of programs. For example, if two programs f and g only invoke operations in sets ϵ_1 and ϵ_2 respectively and every operation in ϵ_1 commutes with every operation in ϵ_2, then f and g commute:

$$x \leftarrow f; y \leftarrow g; k \quad = \quad y \leftarrow g; x \leftarrow f; k$$

The pioneering work by Lucassen and Gifford [7] introduced an effect system to track memory usage in a program by statically partitioning the memory into *regions* and used that information to assist scheduling parallel programs. Benton et al. [3,1,2] and Birkedal et al. [4] gave relational semantics of such region systems of increasing complexity and verified some program equations based on them. Kammar and Plotkin [5] presented a more general account for effect systems based on algebraic effects and studied many effect-dependent program equations. In particular, they also used the Gifford-style region-based approach to manage memory usage.

There is always a balance between expressiveness and complexity. Despite its simplicity and wide applicability, tracking memory usage by *static* regions is not always effective for equational reasoning about some pointer-manipulating programs, especially those manipulating recursive data structures. It is often the case that we want to prove operations on one node of a data structure is irrelevant to operations on the rest of the structure; thus a static region system requires that we annotate the node in a region different from that of the rest of the data structure. If this happens to every node (e.g. in a recursive function), each node of the data structure needs to have its own region, and thus the abstraction provided by regions collapses: regions should abstract disjoint memory cells, not memory cells themselves. In Section 2, we show a concrete example of equational reasoning about a tree traversal program and why a static region system does not work.

The core of the problem is the assumption that every memory cell statically belongs to one region, but when the logical structure of memory is mutable (e.g. when a linked list is split into two lists), we also want regions to be mutable to reflect the structure of the memory (e.g. the region of the list is also split into two regions). To mitigate this problem, we propose a *mutable region system*. In this system, a region is either (i) a single memory cell or (ii) all the cells reachable from a node of a recursive data structure along the points-to relation of cells. For example, the judgement

$$l : \mathit{ListPtr}\ a \vdash t\ l : \mathbf{1}\ !\ \{\mathit{get}_{\mathbf{rc}\ l}\} \tag{1}$$

asserts the program $t\ l$ only reads the linked list starting from l, where the type $\mathit{ListPtr}\ a = \mathit{Nil} \mid \mathit{Ptr}\ (\mathit{Ref}\ (a, \mathit{ListPtr}\ a))$ is either Nil marking the end of the list or a reference to a cell storing a payload of type a and a $\mathit{ListPtr}$ to the next node of the list. The cells linked from l form a region $\mathbf{rc}\ l$ but it is only dynamically determined, and therefore may consist of different cells if the successor field (of type $\mathit{ListPtr}\ a$) stored in l is modified.

We also introduce a complementary construct called *separation guards*, which are effectful programs checking some pointers or their reachable closures are disjoint, otherwise stopping the execution of the program. For example,

$$l : \mathit{ListPtr}\ a,\ l_2 : \mathit{Ref}\ (a,\ \mathit{ListPtr}\ a) \vdash [\mathbf{rc}\ l * l_2] : \mathbf{1}$$

can be understood as a program checking the cell l_2 is not any node of linked list l. With separation guards and our effect system, we can formulate some program equations beyond the expressiveness of previous region systems. For example, given judgement (1), then

$$[\mathbf{rc}\ l * l_2]; put\ l_2\ v; t\ l \quad = \quad [\mathbf{rc}\ l * l_2]; t\ l; put\ l_2\ v$$

says that if cell l_2 is not a node of linked list l, then modification to l_2 can be swapped with $t\ l$, which only accesses list l. In Section 5, we demonstrate using these transformations, we can equationally prove the correctness of the Schorr-Waite traversal algorithm (on binary trees) [15] quite straightforwardly.

{Zhixuan: Here will be a paragraph summarising contributions and the paper structure.}

2 Limitation of Existing Region Systems

This section we show the limitation of static region systems with a practical example of equational reasoning: proving the straightforward recursive implementation of *foldr* for linked lists is semantically equivalent to an optimised implementation using only constant space. The straightforward implementation is not tail-recursive and thus it uses space linear to the length of the list, whereas the optimised version cleverly eliminate the space cost by reusing the space of the linked list itself to store the information needed to control the recursion and restore the linked list after the process. This optimisation is essentially the Schorr-Waite algorithm [15] adapted to linked lists, whose correctness is far from obvious and has been used as a test for many approaches of reasoning about pointer-manipulating programs {Zhixuan: Citations}.

In the following, we start with an attempt to an algebraic proof of the correctness of this optimisation—transforming the optimised implementation to the straightforward one with equational axioms of the programming language and its effect operations. From this attempt, we can see the limitation of static region systems: we want the region partitioning to match the logical structure of data in memory, but when the structure is mutable, static region systems do not allow region partitioning to be mutable to reflect the change of the underlying structure.

2.1 Motivating Example: Constant-time *foldr* for Linked Lists

The straightforward implementation of folding (from the tail side) a linked list is simply

```
$foldrl : (A \to B \to B) \to B \to ListPtr\ A \to \mathbf{F}B$
$foldrl\ f\ \ e\ \ v = \mathbf{case}\ v\ \mathbf{of}$
                      $Nil \quad\ \to \mathbf{return}\ e$
                      $Ptr\ r \to \{(a, n) \leftarrow get\ r; b \leftarrow foldrl\ f\ \ e\ \ n;$
                                 $\mathbf{return}\ (f\ \ a\ \ b)\}$
```

where $\mathbf{F}B$ is the type of computations of B values. The program is recursively defined but not tail-recursive, therefore a compiler is likely to use a stack to implement the recursion. At runtime, the stack has one frame for each recursive call storing local arguments and variables so that they can be restored later when the recursion returns. If we want to minimise the space cost of the stack, we may notice that most local variables are not necessary to be saved in the stack: arguments f and e are not changed throughout the recursion, and local variables a, n and r can be obtained from v. Hence v is the only variable that a stack frame needs to remember to control the recursion. Somewhat surprisingly, we can even reduce the space cost further: since v is used to restore the state when the recursive call for n is finished and the list node n happens to have a field storing a *ListPtr* (that is used to store the successor of n), we can store v in that field instead of an auxiliary stack. But where does the original value of that field of n go? They can be stored in the corresponding field of its next node too. The following program implements this idea with an extra function argument to juggle with these pointers of successive nodes.

```
foldrl_sw f e v = fwd Nil v
fwd f e p v = case v of
                  Nil → bwd f e p v
                  Ptr r → {(a, n) ← get r; put (r, (a, p));
                           fwd f e v n}
bwd f b v n = case v of
                  Nil → return b
                  Ptr r → {(a, p) ← get r; put (r, (a, n));
                           bwd f (f a b) p v}
```

foldrl$_{sw}$ is in fact a special case of the Schorr-Waite traversal algorithm which traverses a graph whose vertices have at most 2 outgoing edges using only 1 bit for each stack frame to control the recursion. The Schorr-Waite algorithm can be easily generalised to traverse a graph whose out-degree is bounded by k using $\log k$ bits for each stack frame, and the above program is the case when $k = 1$ and the list is assumed to be not cyclic.

2.2 Verifying *foldrl*$_{sw}$: First Attempt

Let us try to prove the optimisation above is correct, in the sense that *foldrl*$_{sw}$ can be transformed to *foldrl* by a series of applications of equational axioms on programs that we postulate, including those characterising properties of the language constructs like **case** and function application, and those characterising the effectful operations *get* and *put*.

To prove by induction, it is easy to see that we need to prove a strengthened equality:

$$\{\, b \leftarrow \mathit{foldrl}\ f\ e\ v; \mathit{bwd}\ f\ b\ p\ v \,\} = \mathit{fwd}\ f\ e\ p\ v \tag{2}$$

which specialises to $\mathit{foldrl}_{sw} = \mathit{foldrl}$ when $p = \mathit{Nil}$. When $v = \mathit{Nil}$, the equality can be easily verified. When $v = \mathit{Ptr}\ r$, we have

$$\mathit{fwd}\ f\ e\ p\ v = \{\,(a, n) \leftarrow \mathit{get}\ r; \mathit{put}\ (r, (a, p)); \mathit{fwd}\ f\ e\ v\ n\,\}$$

Assuming we have some inductive principle allowing us to apply Equation 2 to $\mathit{fwd}\ f\ e\ v\ n$ since n is the tail of list v (We will discuss inductive principles later in Section 4.1), we proceed:

$$\begin{aligned}
\mathit{fwd}\ f\ e\ p\ v &= \{\,(a,n) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,p));\\
&\qquad b \leftarrow \mathit{foldrl}\ f\ e\ n; \mathit{bwd}\ f\ b\ v\ n\,\}\\
&= \text{[- Expanding } \mathit{bwd} \text{ -]}\\
&\quad \{\,(a,n) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,p));\\
&\qquad b \leftarrow \mathit{foldrl}\ f\ e\ n;\\
&\qquad (a,p) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,n));\\
&\qquad \mathit{bwd}\ f\ (f\ a\ b)\ p\ v\,\}
\end{aligned} \tag{3}$$

Now we can see why the optimisation works: *fwd* first modifies node v (i.e. $\mathit{Ptr}\ r$) to point to p, and after returning from the recursive call to n, it recovers p from node v and restores v to point to n. Hence we can complete the proof if we show the net effect of those operations leaves node v unchanged.

To show this, if we can prove the two computations in Equation 3 commute with $\mathit{foldr}\ f\ e\ n$:

$$\begin{aligned}
&\{\,b \leftarrow \mathit{foldrl}\ f\ e\ n; \boxed{(a,p) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,n));} K\,\}\\
=\ &\{\,\boxed{(a,p) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,n));}\ b \leftarrow \mathit{foldrl}\ f\ e\ n; K\,\}
\end{aligned} \tag{4}$$

Then

$$\begin{aligned}
\mathit{fwd}\ f\ e\ p\ v &= \{\,(a,n) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,p));\\
&\qquad (a,p) \leftarrow \mathit{get}\ r; \mathit{put}\ (r,(a,n));\\
&\qquad b \leftarrow \mathit{foldrl}\ f\ e\ n;\\
&\qquad \mathit{bwd}\ f\ (f\ a\ b)\ p\ v\,\}\\
&= \text{[- Properties of } \mathit{put} \text{ and } \mathit{get}\text{; See below -]}\\
&\quad \{\,(a,n) \leftarrow \mathit{get}\ r;\\
&\qquad b \leftarrow \mathit{foldrl}\ f\ e\ n;\\
&\qquad \mathit{bwd}\ f\ (f\ a\ b)\ p\ v\,\}\\
&= \text{[- Contracting the definition of } \mathit{foldrl} \text{ -]}\\
&\quad \{\,b' \leftarrow \mathit{foldrl}\ f\ e\ v; \mathit{bwd}\ f\ b'\ p\ v\,\}
\end{aligned}$$

which is exactly what we wanted to show (Equation 2). The properties used in the second step are

$$\begin{aligned}
\{\,\mathit{put}\ (r,v); x \leftarrow \mathit{get}\ r; K\,\} &= \{\,\mathit{put}\ (r,v); K[v/x]\,\}\\
\{\,\mathit{put}\ (r,v); \mathit{put}\ (r,u); K\,\} &= \{\,\mathit{put}\ (r,u); K\,\}\\
\{\,x \leftarrow \mathit{get}\ r; \mathit{put}\ (r,x); K\,\} &= \{\,x \leftarrow \mathit{get}\ r; K\,\}
\end{aligned} \tag{5}$$

Therefore, what remains is to prove the commutativity in Equation 4, which is arguably the most important step of the proof. Intuitively, $\mathit{get}\ r$ and $\mathit{put}\ (r,(a,n))$

access cell r, i.e. the head of the list v, while $\mathit{foldrl}\ f\ e\ n$ accesses the tail of list v. Hence if we want to derive Equation 4 with a region system, we can annotate cell r with some region ϵ_1 and all the cells linked from n with region ϵ_2 so that Equation 4 holds because $\mathit{get}\ r$ and $\mathit{put}\ (r, (a, n))$ access a region different from the one $\mathit{foldrl}\ f\ e\ n$ accesses.

Unfortunately, this strategy does not quite work for two reasons: First, since the argument above for r also applies to n and all their successors, what we finally need is one region ϵ_i for every node r_i of a linked list. This is unfavourable because the abstraction of regions collapses—we are forced to say that $\mathit{foldrl}\ f\ e\ n$ only accesses list n's first node, second node, etc, instead of only one region containing all the nodes of n. The second problem happens in the type system: Now that the reference type is indexed by regions, the type of the i-th cell of a linked list may be upgraded to $\mathit{Ref}\ \epsilon_i\ (a,\ \mathit{ListPtr}_{i+1}\ a)$. But this type signature prevents the second field of this cell pointing to anything but its successor, making programs changing the list structure like foldrl_{sw} untypable. This problem cannot be fixed by simply change the type of the second field to be the type of references to arbitrary region, because we will lose track of the region information necessary for our equational reasoning when reading from that field.

The failure of static region systems in this example is due to the fact that a static region system presumes a fixed region partitioning for a program. While as we have seen in the example above, in different steps of our reasoning, we may want to partition regions in different ways: it is not only because we need region partitioning to match the logic structure of memory cells which is mutable—as in the example above, a node of a list is modified to points to something else and thus should no longer be in the region of the list. Even when all data are immutable, we may still want a more flexible notion of regions—in one part of a program, we probably reason at the level of lists and thus we want all the nodes of a list to be in the same region; while in another part of the same program, we may want to reason at the level of nodes, then we want different nodes of a list in different regions.

3 Mutable Region System

Our observations in the last section suggest us to develop a more flexible region system. Our idea is to let the points-to structure of memory cells *determine* regions: a region is either a single memory cell or all the cells reachable from one cell along the points-to structure of cells. We found it is simpler to implement this idea in a logic system rather than a type system: we introduce effect predicates $(\cdot)\ !\ \epsilon$ on programs of computation types where ϵ is a list of effect operations in the language and two 'virtual' operations get_r and put_r where r is a region in the above sense. The semantics of $t\ !\ \epsilon$ is that program t only invokes the operations in ϵ. Inference rules for effect predicates are introduced.

3.1 Preliminaries: the Language and Logic

As the basis of discussion, we fix a small programming language with algebraic effects based on Levy's call-by-push-value calculus [6]. For a more complete treatment of such a language, we refer the reader to Plotkin and Pretnar's work [9]. The syntax and typing rules of this language are listed in Figure (1) and Figure (2). The language has two categories of types: value types, ranged over by A, and computation types, ranged over by $\underline{A}$. Value types excluding thunk types ($\mathbf{U}\underline{A}$) are called storable types, ranged over by D. In this language, only storable types can be stored in memory cells. Furthermore, we omit general recursively-defined types for simplicity and restrict our treatment to only one particular recursive type: type $\mathit{ListPtr}\ A$ is isomorphic to

$$\mathit{Unit} + \mathit{Ref}\ (A \times \mathit{ListPtr}\ A)$$

Base types:	$\sigma ::= \mathit{Bool} \mid \mathit{Unit} \mid \mathit{Void} \mid \ldots$
Value types:	$A,\ B ::= \sigma \mid \mathit{ListPtr}\ D \mid \mathit{Ref}\ D \mid A_1 \times A_2 \mid A_1 + A_2 \mid \mathbf{U}\underline{A}$
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:	$\underline{A},\ \underline{B} ::= \mathbf{F}A \mid A_1 \to \underline{A_2}$
Base values:	$c ::= \mathit{True} \mid \mathit{False} \mid () \mid \ldots$
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:	$t ::= \mathbf{return}\ v \mid \{x \leftarrow t_1; t_2\} \mid \mathbf{match}\ v\ \mathbf{as}\ \{(x_1, x_2) \to t\}$
	$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\mathit{Nil} \to t_1, \mathit{Ptr}\ x \to t_2\}$
	$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\mathbf{inj}_1\ x_1 \to t_1, \mathbf{inj}_2\ x_2 \to t_2\}$
	$\mid \lambda x : A.\ t \mid t\ v \mid \mathbf{force}\ v \mid \mathit{op}\ v \mid \mu x : \mathbf{U}\underline{A}.\ t$
Operations:	$\mathit{op} ::= \mathit{fail} \mid \mathit{get} \mid \mathit{put} \mid \mathit{new} \mid \ldots$

Fig. 1. Syntax of the language.

{Zhixuan: To be added}

Fig. 2. Typing rules of the language.

We assume that the language includes the effect of failure and *local state* [16]. Failure has one nullary operation *fail* and no equations. Local state has the following three operations:

$$\begin{aligned} get_D &: Ref\ D \to \underline{D} \\ put_D &: (Ref\ D) \times D \to \underline{Unit} \\ new_D &: D \to \underline{Ref\ D} \end{aligned}$$

and they satisfy (a) the three equations in (5) and

$$\{x \leftarrow get\ r; y \leftarrow get\ r; K\} = \{x \leftarrow get\ r; K[x/y]\}$$

(b) commutativity of *get* and *put* on different cells, for example,

$$\{put\ (l_1, u); put\ (l_2, v); K\} = \{put\ (l_2, v); put\ (l_1, u); K\} \quad (l_1 \neq l_2)$$

(c) commutativity laws for *new*, that is,

$$\begin{aligned} \{l \leftarrow new\ v; put\ (r, u); K\} &= \{put\ (r, u); l \leftarrow new\ v; K\} \\ \{l \leftarrow new\ v; x \leftarrow get\ r; K\} &= \{x \leftarrow get\ r; l \leftarrow new\ v; K\} \\ \{l_1 \leftarrow new\ v; l_2 \leftarrow new\ u; K\} &= \{l_2 \leftarrow new\ u; l_1 \leftarrow new\ v; K\} \end{aligned}$$

and (d) the following *separation law*: for any D,

$$\begin{array}{lcl} \{l_1 \leftarrow new_D\ v_1 & = & \{l_1 \leftarrow new_D\ v_1; t_1\} \\ \quad \mathbf{match}\ l_1 \equiv l_2\ \mathbf{as} & & \\ \quad\quad \{False \to t_1 & & \\ \quad\quad\ \ True \to t_2\}\} & & \end{array} \qquad \text{(NEW-DISJ)}$$

which is a special case of the axiom schema $B3_n$ in [16] but is sufficient for our purposes. {Zhixuan: In the standard treatment, these equations come with a context of free variables like l_2 in the last one. Do you find it strange if I omit them?}

Semantics ...

We also need an equational logic for reasoning about programs of this language. We refer the reader to the papers [13,9] for a complete treatment of its semantics and inference rules. Here we only record what is needed in this paper. The formulas of this logic are:

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

The judgement of this logic has form $\Gamma \mid \Psi \vdash \phi$ where Γ is a context of the types of free variables and Φ is a list of formulas that are the premises of ϕ. The inference rules of this logic include:

1. Standard rules for connectives in classical first order logic and structural rules for judgements (e.g. weakening and contraction on premises),

2. standard equivalences for language constructs including sequencing, thunking, function application, case analysis, etc, as in call-by-push-value [6]. For example,

$$\frac{}{v : A,\ t : A \to \underline{A} \mid\ \vdash \{x \leftarrow \mathbf{return}\ v; t\} =_{\mathtt{CBPV}} t\ v}$$

$$\frac{\Gamma \vdash (\lambda x.\ t) : A \to \underline{A} \qquad \Gamma \vdash v : A}{\Gamma \mid\ \vdash (\lambda x.\ t)\ v =_{\mathtt{CBPV}} t[v/x]}$$

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

3. rules inherited from the effect theories, for example,

$$\frac{}{l_{1,2} : \mathit{Ref}\ D,\ v_{1,2} : D,\ t : \underline{A} \mid\ \vdash \{\mathit{put}\ (l_1, v_1); \mathit{put}\ (l_2, v_2); t\} =_{\mathtt{Alg}} \{\mathit{put}\ (l_2, v_2); t\}}$$

4. algebraicity of effect operations, the inductive principle over computations and the universal property of computation types.

In this paper, we will only use the first three kinds of rules.

A difference from the logic defined in [9,13] is that we distinguish equivalences derived only from CBPV rules (written as $=_{\mathtt{CBPV}}$) and those derived from both CBPV rules and effect theories (written as $=_{\mathtt{Alg}}$). This is preferable for our purpose because we do not want to regard $\{v \leftarrow \mathit{get}\ l; \mathit{put}\ l\ v\}$ and $\mathbf{return}\ ()$ as the same because they invoke different operations.

3.2 Effect System as Logic Predicates

Unlike existing type-and-effect systems, our mutable region system is 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:

$$\phi ::= \ldots \mid t\ !\ \epsilon$$
$$\epsilon ::= \emptyset \mid \epsilon, \mathit{op} \mid \epsilon, \mathit{get}_{\mathbf{rc}\ v} \mid \epsilon, \mathit{put}_{\mathbf{rc}\ v} \mid \epsilon, \mathit{get}_v \mid \epsilon, \mathit{put}_v$$

The new term is well-formed when

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

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

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

The intended meaning of the formula $t \mathrel{!} \epsilon$ is that the computation t only invokes operations in ϵ. Specially, get_v and put_v in ϵ mean reading and writing the reference $v : \mathit{Ref}\ D$, and $get_{\mathbf{rc}\ v}$ and $put_{\mathbf{rc}\ v}$ mean reading and writing all the cells linked from v of type $\mathit{ListPtr}\ D$.

The semantics of $[\![\mathit{\Gamma} \vdash t \mathrel{!} \epsilon : \mathbf{form}]\!]$ (abbreviated as $[\![t \mathrel{!} \epsilon]\!]$ below) is a subset of $[\![\mathit{\Gamma}]\!]$ that is the *least*-fixed-point solution of the following mutual-recursive equations. {Zhixuan: No, this definition is wrong. A plausible definition: for all memory configuration (in which regions in ϵ are fintie), running t (with get/put handled and other operations un-handled) only operates corresponding cells (and terminates).}

$$
\begin{aligned}
[\![t \mathrel{!} \epsilon]\!] \;=\; & \{\gamma \in [\![\mathit{\Gamma}]\!] \mid \exists \mathit{\Gamma} \vdash v : A.\ [\![t]\!](\gamma) = [\![\mathbf{return}\ v]\!](\gamma)\} \\
& \cup \{\gamma \in [\![\mathit{\Gamma}]\!] \mid \exists op : B \to \underline{C} \in \epsilon,\ \mathit{\Gamma} \vdash v : B,\ (\mathit{\Gamma}, c : C) \vdash k : \mathbf{F}A. \\
& \qquad [\![t]\!](\gamma) = [\![c \leftarrow op\ v; k]\!](\gamma) \wedge \forall v_c \in [\![C]\!].\ (\gamma, v_c) \in [\![k \mathrel{!} \epsilon]\!]\} \\
& \cup \{\gamma \in [\![\mathit{\Gamma}]\!] \mid \exists get_v \in \epsilon,\ (\mathit{\Gamma}, c : D) \vdash k : \mathbf{F}A. \\
& \qquad [\![t]\!](\gamma) = [\![c \leftarrow get\ v; k]\!](\gamma) \wedge \forall v_c \in [\![D]\!].\ (\gamma, v_c) \in [\![k \mathrel{!} \epsilon]\!]\} \\
& \cup \{\gamma \in [\![\mathit{\Gamma}]\!] \mid \exists put_v \in \epsilon,\ \mathit{\Gamma} \vdash d : D,\ \mathit{\Gamma} \vdash k : \mathbf{F}A. \\
& \qquad [\![t]\!](\gamma) = [\![put\ (v, d); k]\!](\gamma) \wedge \gamma \in [\![k \mathrel{!} \epsilon]\!]\} \\
& \cup \{\gamma \in [\![\mathit{\Gamma}]\!] \mid \exists get_{\mathbf{rc}\ v} \in \epsilon, \text{if } [\![v]\!](\gamma) = [\![\mathit{Nil}]\!] \text{ then } \gamma \in [\![t \mathrel{!} \epsilon \setminus \{get_{\mathbf{rc}\ v}, put_{\mathbf{rc}\ v}\}]\!] \\
& \qquad \text{otherwise } \exists\, l'.\ [\![t]\!](\gamma) = [\![(d, n) \leftarrow get\ l'; k]\!](\gamma) \\
& \qquad \wedge \forall v_d, v_n.\ (\gamma, v_d, v_n) \in [\![k \mathrel{!} \epsilon[\mathbf{rc}\ n/\mathbf{rc}\ v] \cup \epsilon[l'/\mathbf{rc}\ v]]\!]\} \\
& \cup \{\gamma \in [\![\mathit{\Gamma}]\!] \mid \exists put_{\mathbf{rc}\ v} \in \epsilon, \text{if } [\![v]\!](\gamma) = [\![\mathit{Nil}]\!] \text{ then } \gamma \in [\![t \mathrel{!} \epsilon \setminus \{get_{\mathbf{rc}\ v}, put_{\mathbf{rc}\ v}\}]\!] \\
& \qquad \text{otherwise } \exists\, l',\ d,\ k.\ [\![t]\!](\gamma) = [\![put\ (l', d); k]\!](\gamma) \\
& \qquad \wedge \gamma \in [\![k \mathrel{!} \epsilon[l'/\mathbf{rc}\ v]]\!]\}
\end{aligned}
$$

3.3 Inference Rules

{Zhixuan: It sounds worthwhile to to de-couple the inferences of terminance and possible operations } We have the following rules to infer the possible operations used by a program.

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

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

and for any $op : A \to \underline{B} \in \epsilon$ that is not $get/put_{\mathbf{rc}\ v}$,

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

R-Sub says if t only operates on ϵ then it also operates on any larger ϵ'. R-Eq says the predicate $\cdot \mathrel{!} \epsilon$ is compatible with CBPV-equivalence. For example, since programs (**if** *True* **then** t_1 **else** t_2) $=_{\texttt{CBPV}}$ t_1, if $t_1 \mathbin{!} \epsilon$, we also have **if** *True* **then** t_1 **else** $t_2 \mathbin{!} \epsilon$.

R-Op deals with the case where the program invokes an operation in ϵ. It is worth mentioning that the rule R-Op requires $k \mathbin{!} \epsilon$ for *arbitrary* $a : B$ as the premise, even the effect theory for *op* may constrain the possible values for a returned by *op*.

If ϵ contains $get_{\mathbf{rc}\ v}$ or $put_{\mathbf{rc}\ v}$, the program can read or write the cells linked from $v : \mathit{ListPtr}\ D$. When $v = \mathit{Nil}$, the program get no cells to access from $\mathbf{rc}\ v$. When $v = \mathit{Ptr}\ v'$, the program can read or write the cell v', and if it reads it by $(a, n) \leftarrow get\ v'$, its allowed operation on $\mathbf{rc}\ v$ is inherited by $\mathbf{rc}\ n$ and v', which is achieved by substituting $\mathbf{rc}\ n$ for $\mathbf{rc}\ v$ and v' for $\mathbf{rc}\ v$ in ϵ in the inference rules below.

If $get_{\mathbf{rc}\ v} \in \epsilon$

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

If $put_{\mathbf{rc}\ v} \in \epsilon$

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

The inference rule for $get_{\mathbf{rc}\ v}$ also encodes the inductive principle for (finite) linked list by adding $t\ n \mathbin{!} \epsilon[\mathbf{rc}\ n/\mathbf{rc}\ l]$ to the assumption for k. {Zhixuan: It is obviously a very restrictive way because it restricts the recursive structure of t to be aligned with one list $\mathbf{rc}\ v$. As a consequence, this rule cannot deal with $merge\ p\ q \mathbin{!} \{get_{\mathbf{rc}\ p}, put_{\mathbf{rc}\ p}, get_{\mathbf{rc}\ q}, put_{\mathbf{rc}\ q}\}$, the program merging two lists.

But I have no good idea how to work around. Perhaps we need to upgrade predicates $(\cdot \mathbin{!} \epsilon)$—currently only on $\mathbf{F}A$—to higher order functions?}

Theorem 1 (Soundness). *If* $\Gamma \mid \Psi \vdash t \mathbin{!} \epsilon$, *then* $[\![\Psi]\!] \subseteq [\![t \mathbin{!} \epsilon]\!]$.

Proof. {Zhixuan: To add. Hopefully it will not be very difficult.}

4 Separation Guards

{Zhixuan: Given definition and semantics of separation guards.}

Our region predicates defined above can be used to show a program only operates on certain memory cells determined by some variables, but this information is useful only when we know the cells that two programs respectively operates on are disjoint. Ultimately, disjointness comes from the New-Disj axiom of *new* saying that references returned by distinct *new* invocations are different. But this axiom is too primitive for practical use. In this section, we introduce *separation guards* for tracking disjointness more easily at a higher level.

Following the notation of separation logic [14], we write $\phi = l_1 * l_2 * \cdots * l_n$ to denote that cells described by l_i are dijoint. Here l_i can be either a value of type *Ref D* or **rc** v for a value v of type *ListPtr D*. A separation guard $[\phi]$ is a computation of type $\mathbf{F}\,\mathit{Unit}$:

$$
\begin{aligned}
&[\phi] = \mathit{sepChk}\ \phi\ \emptyset \\
&\mathit{sepChk}\ [\,]\ s = \mathbf{return}\ () \\
&\mathit{sepChk}\ (v * \phi)\ x = \mathbf{if}\ l \in x\ \mathbf{then}\ \mathit{fail}\ \mathbf{else}\ \mathit{sepChk}\ \phi\ (x\ \cup\ l) \\
&\mathit{sepChk}\ (\mathbf{rc}\ v * \phi)\ x = \{x' \leftarrow \mathit{chkList}\ v\ x; \mathit{sepChk}\ \phi\ x'\} \\
&\mathit{chkList}\ \mathit{Nil}\ x = \mathbf{return}\ x \\
&\mathit{chkList}\ (\mathit{Ptr}\ p)\ x = \mathbf{if}\ p \in x \\
&\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \{\mathit{fail}; \mathbf{return}\ ()\} \\
&\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \{(\_, n) \leftarrow \mathit{get}\ p; \mathit{chkList}\ n\ (x\ \cup\ p)\}
\end{aligned}
$$

{Zhixuan: This definition may not be formal enough because we didn't assumed the language has a type *Set* to implement x, but we don't necessarily need to implement separation guards in the language, we can treat it as a new language construct and interpret it freely.} $[\phi]$ checks cells described by ϕ are distinct. For a *ListPtr D* element in ϕ, the terminance of $[\phi]$ also implies this list in memory is finite.

Separation guards can be used to assert preconditions of some program equivalences. For example, if t is a program traversing list $l : \mathit{ListPtr}\ D$, it is (algebraically) equivalent to **return** () when l is a finite list:

$$[\mathbf{rc}\ l]; t \quad =_{\mathtt{Alg}} \quad [\mathbf{rc}\ l]; \mathbf{return}\ ()$$

The equality holds whenever l is finite or not: when l is infinite, $[\mathbf{rc}\ l]$ diverges or fails. In both cases, it is a left-zero of the sequencing operator " ; " and thus the equality holds.

4.1 Inference Rules

Although separation guards can be defined as a concrete program as above, we intend them to be used abstractly with the following inference rules. Define $c\langle\, a =_{\mathtt{Alg}} b\,\rangle$ to be $(c; a) =_{\mathtt{Alg}} (c; b)$.

$$\frac{}{[\phi]\langle\ \mathit{new}\ t =_{\mathtt{Alg}} (l \leftarrow \mathit{new}\ t; [\phi * l]; \mathbf{return}\ l)\ \rangle} \qquad \frac{\Gamma \mid \Psi, l_1 \neq l_2 \vdash t_1 =_{\mathtt{Alg}} t_2}{\Gamma \mid \Psi \vdash [l_1 * l_2]\langle\ t_1 =_{\mathtt{Alg}} t_2\ \rangle}$$

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

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

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

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

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

$[\cdot]$ has the following structural properties:

$$[\phi_1 * \phi_2] =_{\texttt{Alg}} [\phi_2 * \phi_1] \qquad [(\phi_1 * \phi_2) * \phi_3] =_{\texttt{Alg}} [\phi_1 * (\phi_2 * \phi_3)]$$

$$[\top] =_{\texttt{Alg}} \mathbf{return}\ () \qquad [\phi_1 * \phi_2] =_{\texttt{Alg}} ([\phi_1 * \phi_2]; [\phi_1])$$

$$([\phi_1]; [\phi_2]) =_{\texttt{Alg}} ([\phi_2]; [\phi_1])$$

Proposition 1. *The inference rules above are sound.*

Proof. {Zhixuan: It'll be a large verifying proof.}

4.2 Effect-dependent Transformations

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. {Zhixuan: Proving the above two rules using the inference rules of separation guards and effect predicate.}

5 Case Study: Equational Reasoning about Schorr-Waite Traversal

6 Related Work

Algebraic effects: [10]
Effect systems: [7,17,8]

7 Conclusion

References

1. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 87–96. PPDP '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1273920.1273932

2. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations: Higher-order store. In: Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. pp. 301–312. PPDP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1599410.1599447
3. Benton, N., Kennedy, A., Hofmann, M., Beringer, L.: Reading, writing and relations. In: Kobayashi, N. (ed.) Programming Languages and Systems. pp. 114–130. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
4. Birkedal, L., Jaber, G., Sieczkowski, F., Thamsborg, J.: A kripke logical relation for effect-based program transformations. Inf. Comput. **249**(C), 160–189 (Aug 2016). https://doi.org/10.1016/j.ic.2016.04.003
5. Kammar, O., Plotkin, G.D.: Algebraic foundations for effect-dependent optimisations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 349–360. POPL '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103698
6. Levy, P.B.: Call-by-push-value: A Functional/imperative Synthesis, vol. 2. Springer Science & Business Media (2012)
7. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 47–57. POPL '88, ACM, New York, NY, USA (1988). https://doi.org/10.1145/73560.73564
8. Marino, D., Millstein, T.: A generic type-and-effect system. In: Proceedings of the 4th International Workshop on Types in Language Design and Implementation. pp. 39–50. TLDI '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1481861.1481868
9. Plotkin, G., Pretnar, M.: A logic for algebraic effects. In: 2008 23rd Annual IEEE Symposium on Logic in Computer Science. pp. 118–129 (June 2008). https://doi.org/10.1109/LICS.2008.45
10. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) Foundations of Software Science and Computation Structures. pp. 342–356. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_24
11. Plotkin, G., Power, J.: Computational effects and operations: An overview. Electron. Notes Theor. Comput. Sci. **73**, 149–163 (Oct 2004). https://doi.org/10.1016/j.entcs.2004.08.008
12. Plotkin, G., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science **9**(4) (Dec 2013). https://doi.org/10.2168/lmcs-9(4:23)2013
13. Pretnar, M.: Logic and handling of algebraic effects. Ph.D. thesis (2010)
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002), http://dl.acm.org/citation.cfm?id=645683.664578
15. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM **10**(8), 501–506 (Aug 1967). https://doi.org/10.1145/363534.363554
16. Staton, S.: Completeness for algebraic theories of local state. In: Ong, L. (ed.) Foundations of Software Science and Computational Structures. pp. 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
17. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming **2**(3), 245–271 (1992). https://doi.org/10.1017/S0956796800000393