A Mutable Region System for Equational Reasoning about Pointer Algorithms

by

Zhixuan Yang

Master Dissertation

submitted to the Department of Informatics

SOKENDAI (The Graduate University for Advanced Studies)

June 2019

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.

Contents

List of Figures

1 Introduction

Plotkin and Power's algebraic effects (Plotkin and Power, 2002, 2004) and their handlers (Plotkin and Pretnar, 2013; Pretnar, 2010) 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 (Plotkin and Pretnar, 2008; Pretnar, 2010). The equations of some algebraic effects are also proved to be (Hilbert-Post) complete, including the effects of global and local state (Staton, 2010).

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 (Lucassen and Gifford, 1988) 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. (Benton et al., 2007, 2009, 2006) and Birkedal et al. (Birkedal et al., 2016) gave relational semantics of such region systems of increasing complexity and verified some program equations based on them. Kammar and Plotkin (Kammar and Plotkin, 2012) 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 Chapter 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

$$l : \mathit{ListPtr}\ a \vdash t\ l : \mathbf{1}\ !\,\{\mathit{get}_{\mathbf{rc}\ l}\} \tag{1.1}$$

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 *ListPtr* to the next node of the list. The cells linked from l form a region **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,

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

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 (1.1), then

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

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 4.3, we demonstrate using these transformations, we can equationally prove the correctness of the Schorr-Waite traversal algorithm (on binary trees) (Schorr and Waite, 1967) quite straightforwardly.

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

2 Limitation of Existing Region Systems

This chapter 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 (Schorr and Waite, 1967) 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

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

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.

$$\mathit{foldrl}_{sw}\ f\ e\ v = \mathit{fwd}\ \mathit{Nil}\ v$$

```
$fwd\ f\ \ e\ p\ v =$ case $v$ of
                 $Nil \to bwd\ f\ \ e\ p\ v$
                 $Ptr\ r \to \{(a, n) \leftarrow get\ r; put\ (r, (a, p));$
                          $fwd\ f\ \ e\ v\ n\}$
$bwd\ f\ \ b\ v\ n =$ case $v$ of
                 $Nil \to$ return $b$
                 $Ptr\ r \to \{(a, p) \leftarrow get\ r; put\ (r, (a, n));$
                          $bwd\ f\ \ (f\ \ a\ b)\ p\ v\}$
```

$foldrl_{sw}$ 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:

$$\{b \leftarrow foldrl\ f\ \ e\ v; bwd\ f\ \ b\ p\ v\} = fwd\ f\ \ e\ p\ v \tag{2.1}$$

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

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

Assuming we have some inductive principle allowing us to apply Equation 2.1 to $fwd\ f\ \ e\ v\ \ n$ since n is the tail of list v (We will discuss inductive principles later in

Section 4.1), we proceed:

$$\begin{aligned} \mathit{fwd}\ f\ \ e\ p\ v = & \{(a,n) \leftarrow \mathit{get}\ \ r; \mathit{put}\ \ (r,(a,p)); \\ & \quad b \leftarrow \mathit{foldrl}\ f\ \ e\ \ n; \mathit{bwd}\ f\ \ b\ \ v\ \ n\} \\ = & \text{[- Expanding } \mathit{bwd} \text{ -]} \\ & \{(a,n) \leftarrow \mathit{get}\ \ r; \mathit{put}\ \ (r,(a,p)); \\ & \quad b \leftarrow \mathit{foldrl}\ f\ \ e\ \ n; \\ & \quad (a,p) \leftarrow \mathit{get}\ \ r; \mathit{put}\ \ (r,(a,n)); \\ & \quad \mathit{bwd}\ f\ \ (f\ \ a\ \ b)\ \ p\ \ v\} \end{aligned} \tag{2.2}$$

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 2.2 commute with $\mathit{foldr}\ f\ \ e\ \ n$:

$$\begin{aligned} & \{b \leftarrow \mathit{foldrl}\ f\ \ e\ \ n; \boxed{(a,p) \leftarrow \mathit{get}\ \ r; \mathit{put}\ \ (r,(a,n));} K\} \\ = & \{\boxed{(a,p) \leftarrow \mathit{get}\ \ r; \mathit{put}\ \ (r,(a,n));}\ b \leftarrow \mathit{foldrl}\ f\ \ e\ \ n; K\} \end{aligned} \tag{2.3}$$

Then

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

which is exactly what we wanted to show (Equation 2.1). The properties used in the second step are

$$\begin{aligned}\{put\ (r,v); x \leftarrow get\ r; K\} &= \{put\ (r,v); K[v/x]\}\\ \{put\ (r,v); put\ (r,u); K\} &= \{put\ (r,u); K\}\\ \{x \leftarrow get\ r; put\ (r,x); K\} &= \{x \leftarrow get\ r; K\}\end{aligned} \tag{2.4}$$

Therefore, what remains is to prove the commutativity in Equation 2.3, which is arguably the most important step of the proof. Intuitively, $get\ r$ and $put\ (r,(a,n))$ access cell r, i.e. the head of the list v, while $foldrl\ f\ e\ n$ accesses the tail of list v. Hence if we want to derive Equation 2.3 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 2.3 holds because $get\ r$ and $put\ (r,(a,n))$ access a region different from the one $foldrl\ f\ e\ n$ 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 $foldrl\ f\ e\ n$ 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 $Ref\ \epsilon_i\ (a \times ListPtr_{i+1}\ a)$. 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 $(\cdot) \mathrel{!} \epsilon$ 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 $t \mathrel{!} \epsilon$ 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 (Levy, 2012). For a more complete treatment

Base types:	$\sigma ::= \mathit{Bool} \mid \mathit{Unit} \mid \mathit{Void} \mid \dots$
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:	$c ::= \mathit{True} \mid \mathit{False} \mid () \mid \dots$
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:	$t ::= \mathbf{return}\ v \mid \{x \leftarrow t_1; t_2\} \mid \mathbf{match}\ v\ \mathbf{as}\ \{(x_1, x_2) \to t\}$
	$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\mathit{Nil} \to t_1, \mathit{Ptr}\ x \to t_2\}$
	$\mid \mathbf{match}\ v\ \mathbf{as}\ \{\mathbf{inj}_1\ x_1 \to t_1, \mathbf{inj}_2\ x_2 \to t_2\}$
	$\mid \lambda x : A.\ t \mid t\ v \mid \mathbf{force}\ v \mid \mathit{op}\ v \mid \mu x : \underline{A}.\ t$
Operations:	$\mathit{op} ::= \mathit{fail} \mid \Omega \mid \mathit{get} \mid \mathit{put} \mid \mathit{new} \mid \dots$

Figure 3.1: Syntax of the language.

{Zhixuan: To be added}

Figure 3.2: Typing rules of the language.

of such a language, we refer the reader to Plotkin and Pretnar's work (Plotkin and Pretnar, 2008). The syntax and typing rules of this language are listed in Figure (3.1) and Figure (3.2). 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

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

We assume that the language includes the effect of failure (*fail*), non-divergence (Ω) and *local state* (Staton, 2010). 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 (2.4) 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,

$$\begin{array}{lcl} \{l_1 \leftarrow new_D\ v_1 & = & \{l_1 \leftarrow new_D\ v_1; t_1\} \\ \quad \mathbf{match}\ l_1 \equiv l_2\ \mathbf{as} & & \\ \quad\quad \{False \to t_1 & & \\ \quad\quad\ \ True \to t_2\}\} & & \end{array} \tag{Ax-Sep}$$

which is a special case of the axiom schema B3$_n$ in (Staton, 2010) 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 (Plotkin and Pretnar, 2008; Pretnar, 2010) 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:

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

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 (Levy, 2012). For example,

$$\frac{}{v : A,\ t : A \to \underline{A} \mid\ \vdash \{x \leftarrow \textbf{return}\ v; t\} = t\ v} \qquad \frac{\Gamma \vdash (\lambda x.\ t) : A \to \underline{A} \qquad \Gamma \vdash v : A}{\Gamma \mid\ \vdash (\lambda x.\ t)\ v = t[v/x]}$$

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

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

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

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.

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:

$$\begin{aligned}
&\phi ::= \ldots \mid t \ ! \ \epsilon \\
&\epsilon ::= \emptyset \mid \epsilon, op \mid \epsilon, get_{\mathbf{rc}\ v} \mid \epsilon, put_{\mathbf{rc}\ v} \mid \epsilon, get_v \mid \epsilon, put_v
\end{aligned}$$

The new term is well-formed when

$$\frac{\Gamma \vdash t : \mathbf{F}A}{\Gamma \vdash t \ ! \ \cdot : \mathbf{form}} \qquad \frac{\Gamma \vdash t \ ! \ \epsilon : \mathbf{form}}{\Gamma \vdash t \ ! \ \epsilon, op : \mathbf{form}} \ (op \notin \{get,\ put\})$$

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

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

Although ϵ is formally a list of comma-separated operations, we will regard it as a set and thus use set operations like inclusion and minus on it.

Example 3.2.1. *Let* Γ *be* $\{l : Ref\ D,\ r : Ref\ (D \times ListPtr\ D)\}$,

$$\Gamma \vdash \{(a, n) \leftarrow get\ r; put\ (l, a)\} \ ! \{put_l,\ get_{\mathbf{rc}\ (Ptr\ r)}\} : \mathbf{form}$$

is derivable.

Although our effect predicate is defined only on first-order computations, we can work with higher order functions by using quantification in the logic. For example, if $\Gamma \vdash t : ListPtr\ D \to \mathbf{F}A$

$$\Gamma \mid \ \vdash \forall (l : ListPtr\ D).\ t\ l \ ! \{get_{\mathbf{rc}\ l}\}$$

is well-formed and it expresses that function t only reads l when it is applied to list l.

The intended meaning of effect predicate $t \ ! \ \epsilon$ is: provided that the regions mentioned in ϵ are dijoint, the computation t only applies a finite number of operations in ϵ. Before giving a formal definition of this semantics, we present the inference rules first in the rest of this section, which may provide more intuition, and then in Section 3.4 we give the formal semantics of effect predicates.

3.3 Inference Rules

An advantage of tracking effects in the equational logic is that we only need to design inference rules for effects-related language constructs—**return**, sequencing and operation application. Other language constructs like case-analysis are handled by the equational logic as we will see in the example below. Our inference rules are:

- two structural rules

$$\frac{\Gamma \mid \Psi \vdash t \ ! \ \epsilon \qquad \epsilon \subseteq \epsilon'}{\Gamma \mid \Psi \vdash t \ ! \ \epsilon'} \ \text{R-Sub} \qquad\qquad \frac{\Gamma \mid \Psi \vdash t = t' \ \wedge \ t' \ ! \ \epsilon}{\Gamma \mid \Psi \vdash t \ ! \ \epsilon} \ \text{R-Eq}$$

- rules for **return** and sequencing

$$\frac{}{x : A \mid \ \vdash \mathbf{return}\ x \ ! \ \emptyset} \ \text{R-Pure} \qquad \frac{\Gamma \mid \Psi \vdash t_1 \ ! \ \epsilon \qquad \Gamma, x : A \mid \Psi \vdash t_2 \ ! \ \epsilon}{\Gamma \mid \Psi \vdash \{x \leftarrow t_1; t_2\} \ ! \ \epsilon} \ \text{R-Seq}$$

- rules for effect operations, for any operation of type $A \to \underline{B}$ in the language,

$$\frac{}{v : A \mid \ \vdash op\ v \ ! \{op\}} \ \text{R-Op}$$

and specially for get_l and put_l (Formally, they are not operation of the language so these rules are needed)

$$\frac{}{l : \mathit{Ref}\ D \mid\ \vdash \mathit{get}\ l\ !\ \{\mathit{get}_l\}} \qquad \frac{}{l : \mathit{Ref}\ D,\ a : D \mid\ \vdash \mathit{put}\ (l, a)\ !\ \{\mathit{put}_l\}}$$

- rules for $\mathit{get}_{\mathbf{rc}\ l}$ and $\mathit{put}_{\mathbf{rc}\ l}$

$$\frac{\Gamma \mid \Psi \vdash k\ !\ \epsilon \setminus \{\mathit{get}_{\mathbf{rc}\ \mathit{Nil}}, \mathit{put}_{\mathbf{rc}\ \mathit{Nil}}\}}{\Gamma \mid \Psi \vdash t\ !\ \epsilon}\ \text{R-NIL}$$

$$\frac{\Gamma, a, n \mid \Psi \vdash k\ !\ \epsilon[l/x] \cup \epsilon[\mathbf{rc}\ n/x]}{\Gamma \mid \Psi \vdash \{(a, n) \leftarrow \mathit{get}\ l; k\}\ !\ \epsilon[\mathbf{rc}\ (\mathit{Ptr}\ l)/x]}\ (\mathit{get}_x \in \epsilon)\ \text{R-GETRC}$$

$$\frac{\Gamma \mid \Psi \vdash k\ !\ \epsilon[l/x]}{\Gamma \mid \Psi \vdash k\ !\ \epsilon[\mathbf{rc}\ (\mathit{Ptr}\ l)/x]}\ (\mathit{put}_x \in \epsilon)\ \text{R-PUTRC}$$

These rules deserve some explanation: The rule R-NIL means that $\mathit{get}_{\mathbf{rc}\ \mathit{Nil}}$ and $\mathit{put}_{\mathbf{rc}\ \mathit{Nil}}$ cannot be used by the program; the rule R-GETRC means that if a program has the permission to read the list from l, it can read the cell l and its permission on $\mathbf{rc}\ (\mathit{Ptr}\ l)$ is split into the same permission on cell l and the rest of the list (i.e. $\mathbf{rc}\ n$);

Example 3.3.1. *By R-GETRC, if* $\Gamma, a, n \mid\ \vdash k\ !\ \{\mathit{get}_l, \mathit{put}_l, \mathit{get}_{\mathbf{rc}\ n}, \mathit{put}_{\mathbf{rc}\ n}\}$ *is derivable, then*

$$\Gamma \mid\ \vdash \{(a, n) \leftarrow \mathit{get}\ l; k\}\ !\ \{\mathit{get}_{\mathbf{rc}\ (\mathit{Ptr}\ l)}, \mathit{put}_{\mathbf{rc}\ (\mathit{Ptr}\ l)}\}$$

is derivable.

the rule R-PUTRC says that if a program has the permission to write the list from l, then it has the permission to write the cell l itself. This rule seems not very useful, but it reflects the fact that even if a program can write $\mathbf{rc}\ \mathit{Ptr}\ l$, if it cannot read l, its accessible cells are restricted to l only.

- Finally, we introduce a rule that may not be valid in more general settings but is safe in our context because it assumes all lists in memory are finite. The rule is, under side condition $get_{\mathbf{rc}\ v} \in \epsilon$,

$$\frac{\begin{array}{c}\Gamma,\ v \mid \Psi, v = \mathit{Nil} \vdash t\ !\ \epsilon \qquad \Gamma,\ v,\ r \mid \Psi,\ v = \mathit{Ptr}\ r \vdash t = \{(a, n) \leftarrow \mathit{get}\ r; k\} \\ \Gamma, v, r, a, n \mid \Psi,\ v = \mathit{Ptr}\ r,\ t[n/v]\ !\ \epsilon[n/v] \vdash k\ !\ \epsilon[n/v] \cup \epsilon[r/\mathbf{rc}\ v]\end{array}}{\Gamma,\ v : \mathit{ListPtr}\ D \mid \Psi \vdash t\ !\ \epsilon}$$

Without this rule, a recursive program defined with μ can only satisfy $\cdot\ !\ \epsilon$ if $\Omega \in \epsilon$ (See Scott-induction in Chapter 9 of paper (Pretnar, 2010)). This rule allows a program that is a structural recursion along some linked list to satisfy predicate $\cdot\ !\ \epsilon$ without including Ω in ϵ, as if it is not a recursive program.

Example 3.3.2. *Without this rule, we can only derive* $\mathit{foldrl}\ f\ e\ l\ !\ \{get_{\mathbf{rc}\ l}, \Omega\}$ *using Scott-induction. With this, we can derive*

$$\frac{\dfrac{\mathit{foldrl}\ f\ e\ \mathit{Nil} = \mathbf{return}\ () \qquad \mathbf{return}\ ()\ !\ \{get_{\mathbf{rc}\ l}\}}{f, e, l \mid l = \mathit{Nil} \vdash \mathit{foldrl}\ f\ e\ \mathit{Nil}\ !\ \{get_{\mathbf{rc}\ l}\}} \qquad ① \qquad ②}{f, e, l \mid\ \vdash \mathit{foldrl}\ f\ e\ l\ !\ \{get_{\mathbf{rc}\ l}\}}$$

where ① is

$$f, e, l, r \mid l = \mathit{Ptr}\ r \vdash \mathit{foldrl}\ f\ e\ l = \{(a, n) \leftarrow \mathit{get}\ r; K\}$$
$$K =_{\mathrm{def}} \{b \leftarrow \mathit{foldrl}\ f\ e\ n; \mathbf{return}\ f\ a\ b\}$$

and ② is

$$\frac{\cdots}{f, e, l, r, a, n \mid l = \mathit{Ptr}\ r, \mathit{foldrl}\ f\ e\ n\ !\ \{get_{\mathbf{rc}\ n}\} \vdash K\ !\ \{get_r, get_{\mathbf{rc}\ n}\}}$$

An obvious difference of our effect predicates from existing type-and-effect system is that we only have inference rules for effect related language constructs, because other language constructs can be handled by R-Eq and corresponding elimination rules for the construct of the logic.

Example 3.3.3. *Letting P denote* **case** b **of** $\{\mathit{True} \to op_1; \mathit{False} \to op_2\}$, *we can derive*

$$\dfrac{\dfrac{\text{①}\qquad\text{②}}{b : \mathit{Bool} \mid b = \mathit{False} \lor b = \mathit{True} \vdash P \,!\, \{op_1, op_2\}}}{b : \mathit{Bool} \mid \ \vdash P \,!\, \{op_1, op_2\}}\,(\textsc{ExM-Bool})$$

where ExM-Bool *is the rule in the logic saying that $b : \mathit{Bool}$ is either True or False,* ① *is*

$$\dfrac{b : \mathit{Bool} \mid b = \mathit{True} \vdash P = op_1 \qquad \dfrac{\mid\ \vdash op_1 \,!\, \{op_1\}}{b : \mathit{Bool} \mid b = \mathit{True} \vdash op_1 \,!\, \{op_1, op_2\}}}{b : \mathit{Bool} \mid b = \mathit{False} \vdash P \,!\, \{op_1, op_2\}}$$

and similarly ② *is*

$$\dfrac{b : \mathit{Bool} \mid b = \mathit{False} \vdash P = op_2 \qquad \dfrac{\mid\ \vdash op_2 \,!\, \{op_2\}}{b : \mathit{Bool} \mid b = \mathit{False} \vdash op_2 \,!\, \{op_1, op_2\}}}{b : \mathit{Bool} \mid b = \mathit{False} \vdash P \,!\, \{op_1, op_2\}}$$

{Zhixuan: An example demonstrating the assumption of disjointness of regions by these rules.}

3.4 Semantics

{Zhixuan: And by Thursday, I will write the equivalence section!}

{Zhixuan: And on Friday, I will clean the paper from the beginning}

{Zhixuan: And on Sat and Sun, I will write the related work and conclusion}

Now let us formalise our intuitive semantics of effect predicate $t \,!\, \epsilon$: when regions mentioned in ϵ are disjoint and have finite cells, t is a computation only using the operations contained in ϵ and t is also finite. Recall that the semantics of $t : \mathsf{F}A$ is an equivalence class of trees whose internal nodes are labeled with operation symbols and leaves are labeled with **return** v for some $v \in [\![A]\!]$. Trees in $[\![t]\!]$ are equal in the sense that anyone of them can be rewritten to another by the equations of the effect theory (Bauer, 2018). Therefore if we can define a denotational semantics $[\![\epsilon]\!]$, presumably to be the set of operations available to t, then $[\![t \; ! \; \epsilon]\!]$ can be defined to mean that $[\![t]\!]$ has some element T whose operations is a subset of $[\![\epsilon]\!]$ and T is a well-founded tree.

However, how to interpret ϵ in the framework of algebraic effects is not straightforward. For op, get_l and put_l in ϵ, they can be easily interpreted by corresponding operations op, $get_{[\![l]\!]}$ and $put_{[\![l]\!]}$. For $get_{\mathbf{rc}\ l}$ (and $put_{\mathbf{rc}\ l}$), we want to interpret it as a set of operations $\{get_{[\![l]\!]}, get_{r_1}, get_{r_2}, \ldots\}$ where $[\![l]\!]$ points to r_1, r_1 points to r_2 in the memory, etc. However, in the semantics of algebraic effects, there is no explicit representation for the memory so that we do not immediately know what $r_1, r_2, \ldots$ are. (For comparison, the semantics of t in other approaches is usually a function $Mem \to ([\![A]\!], Mem)$ which has an explicit Mem.)

This problem may be tackled by the coalgebraic treatment of effects (Plotkin and Power, 2008), but here we adopt a simple workaround: Although we do not have an explicit representation of memory to work with, we do have an operation get to probe the memory—if $get\ r$ returns v, we know memory cell r currently stores value v. Hence if $[\mathbf{rc}\ v]$ is a program traversing linked list v and returns the set of references to the nodes of the list, then we can interpret $t \; ! \, \{get_{\mathbf{rc}\ v}\}$ in this way: in program $\{r \leftarrow [\mathbf{rc}\ v]; t\}$, t only reads the references in r. And as we mentioned in Section 3.3, predicate $t \; ! \, \{get_{r_1}, put_{r_2}\}$ implicitly assumes that r_1 and r_2 are disjoint, so to interpret this predicate, we want $[r_1 * r_2]$ not only returns the references in r_1 and r_2 but also checks they are disjoint.

Now let us define such programs more formally, which collect references to cells of memory regions and check disjointness. Following the notation of separation logic (Reynolds, 2002), we write $\phi = l_1 * l_2 * \cdots * l_n$ to denote a list of separate regions. Here l_i is either an expression of type $Ref\ \ D$ or expression $\mathbf{rc}\ v$ for some v of type $ListPtr\ \ D$. We add a new kind of term $[\phi]$ in the language, which we call *separation guards*. It has type $\mathbf{F}MemSnap$, where type $MemSnap$ abbreviates

$$FinMap\ (Ref\ \ (a \times ListPtr\ \ a))\ (Set\ (Ref\ \ (a \times ListPtr\ \ a)))$$

Thus $x : MemSnap$ is finite map from references to sets of references. The semantics of $[\phi]$ is the computation denoted by the following program

$$
\begin{aligned}
&[\![[\phi]]\!] = \mathit{sepChk}\ \phi\ \emptyset\ \emptyset\\
&\mathit{sepChk}\ [\,]\ x\ \mathit{rcs} = \textbf{return}\ \mathit{rcs}\\
&\mathit{sepChk}\ (v * \phi)\ x\ \mathit{rcs} = \textbf{if}\ l \in x\ \textbf{then}\ \mathit{fail}\ \textbf{else}\ \mathit{sepChk}\ \phi\ (x \cup l)\ \mathit{rcs}\\
&\mathit{sepChk}\ (\textbf{rc}\ v * \phi)\ x\ \mathit{rcs} = \{x' \leftarrow \mathit{tvsList}\ v;\\
&\qquad\qquad\qquad\qquad\qquad \textbf{if}\ x' \cap x \not\equiv \emptyset\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{fail}\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{sepChk}\ \phi\ (x' \cup x)\ (\mathit{rcs} \cup \{v \mapsto x'\})\}\\
&\mathit{tvsList}\ \mathit{Nil} = \textbf{return}\ \emptyset\\
&\mathit{tvsList}\ (\mathit{Ptr}\ r) = \{(a, n) \leftarrow \mathit{get}\ p; \mathit{rs} \leftarrow \mathit{tvsList}\ n; \textbf{return}\ (r \cup \mathit{rs})\}
\end{aligned}
$$

Thus $[\phi]$ traverses each region $r \in \phi$ one by one and checks their cells are disjoint, otherwise it calls *fail*. When it terminates, it returns a finite map *rcs* mapping every region r in ϕ to the set of its cells, which can be thought as a snapshot of the current memory.

By probing the memory with separation guards, we can define the semantics of effect predicates now. For any effect set ϵ, let $R_\epsilon = \{l \mid \mathit{put}_l \in \epsilon\} \cup \{\textbf{rc}\ p \mid \mathit{get}_{\textbf{rc}\ p} \in \epsilon\}$. Then take ϕ_ϵ to be an arbitrary $*$-sequence of all the elements of R. We define the semantics of judgement $\Gamma \vdash t\ !\ \epsilon$ to be the set of $\gamma \in [\![\Gamma]\!]$ such that $[\![\{[\phi_\epsilon]; t\}]\!]_\gamma$ has an element T, which is a tree satisfying:

- there is some $T_1 \in [\![[\phi_\epsilon]]\!]_\gamma$ and for any leaf node of T_1 label with **return** x, there is a computation tree T_x, such that T is equal to the tree obtained by replacing every leaf node **return** x of T_1 with corresponding T_x;
- every T_x is well-founded and every operation in T_x is either: (i) $\mathit{op}\ v$ for some $\mathit{op} \in \epsilon$, (ii) $\mathit{get}\ v$ for some $\mathit{get}_l \in \epsilon$ and $v = [\![l]\!]_\gamma$, (iii) $\mathit{put}\ (v, d)$ for some $\mathit{put}_l \in \phi$ and $v = [\![l]\!]_\gamma$, (iv) $\mathit{get}\ v$ for some $\mathit{get}_{\textbf{rc}\ r}$ and $v \in x([\![r]\!]_\gamma)$, and (v) $\mathit{put}\ (v, d)$ for some $\mathit{put}_{\textbf{rc}\ r}$ and $v \in x([\![r]\!]_\gamma)$.

Theorem 3.4.1 (Soundness). *If $\Gamma \mid \psi_1, \ldots, \psi_n \vdash \phi$ is derivable from the rules in Section 3.3, then*

$$\bigcap_{1 \leqslant i \leqslant n} [\![\Gamma \vdash \psi_i]\!] \subseteq [\![\Gamma \vdash \phi]\!]$$

4 Program Equivalences

{Zhixuan: Revise the introductory paragraph because separation guards are also necessary for defining the semantics of effect predicates.} Our effect 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 Ax-Sep axiom of *new* saying that references returned by distinct *new* invocations are different. But this axiom is too primitive for practical use. In this chapter, we introduce *separation guards* for tracking disjointness more easily at a higher level.

Following the notation of separation logic (Reynolds, 2002), we write $\phi = l_1 * l_2 * \cdots * l_n$ to denote that cells described by l_i are disjoint. Here l_i can be either a value of type $\mathit{Ref}\ D$ or $\mathbf{rc}\ v$ for a value v of type $\mathit{ListPtr}\ D$. A separation guard $[\phi]$ is a computation of type $\mathbf{F}\mathit{Unit}$:

$$[\phi] = \mathit{sepChk}\ \phi\ \emptyset$$

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

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

{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 $l : ListPtr\ D$, it is (algebraically) equivalent to **return** () when l is a finite list:

$$[\textbf{rc}\ l]; t \quad = \quad [\textbf{rc}\ l]; \textbf{return}\ ()$$

The equality holds whenever l is finite or not: when l is infinite, $[\textbf{rc}\ l]$ 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 $c\langle\ a = b\ \rangle$ to be

$(c; a) = (c; b)$.

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

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

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

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

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

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

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

$$[\phi_1 * \phi_2] = [\phi_2 * \phi_1] \qquad [(\phi_1 * \phi_2) * \phi_3] = [\phi_1 * (\phi_2 * \phi_3)] \qquad [\top] = \mathbf{return}\ ()$$

$$[\phi_1 * \phi_2] = ([\phi_1 * \phi_2]; [\phi_1]) \qquad ([\phi_1]; [\phi_2]) = ([\phi_2]; [\phi_1])$$

Proposition 4.1.1. *The inference rules above are sound.*

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

4.2 Effect-dependent Transformations

A frame rule:

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

Commutativity lemma

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

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

4.3 Verifying $foldrl_{sw}$, resumed

5 Related Work

Algebraic effects: (Plotkin and Power, 2002)

Effect systems: (Lucassen and Gifford, 1988; Marino and Millstein, 2009; Talpin and Jouvelot, 1992)

6
Conclusion

Bibliography

Andrej Bauer. 2018. What is algebraic about algebraic effects and handlers? *CoRR* abs/1807.05923 (2018). arXiv:1807.05923 http://arxiv.org/abs/1807.05923

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. 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 (PPDP '07)*. ACM, New York, NY, USA, 87–96. https://doi.org/10.1145/1273920.1273932

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. 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 (PPDP '09)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/1599410.1599447

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–130.

Lars Birkedal, Guilhem Jaber, Filip Sieczkowski, and Jacob Thamsborg. 2016. A Kripke Logical Relation for Effect-based Program Transformations. *Inf. Comput.* 249, C (Aug. 2016), 160–189. https://doi.org/10.1016/j.ic.2016.04.003

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic Foundations for Effect-dependent Optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages (POPL '12). ACM, New York, NY, USA, 349–360. https://doi.org/10.1145/2103656.2103698

Paul Blain Levy. 2012. *Call-by-push-value: A Functional/imperative Synthesis*. Vol. 2. Springer Science & Business Media.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Daniel Marino and Todd Millstein. 2009. A Generic Type-and-effect System. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1481861.1481868

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures*, Mogens Nielsen and Uffe Engberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. *Electron. Notes Theor. Comput. Sci.* 73 (Oct. 2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

Gordon Plotkin and John Power. 2008. Tensors of Comodels and Models for Operational Semantics. *Electron. Notes Theor. Comput. Sci.* 218 (Oct. 2008), 295–311. https://doi.org/10.1016/j.entcs.2008.10.018

G. Plotkin and M. Pretnar. 2008. A Logic for Algebraic Effects. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 118–129. https://doi.org/10.1109/LICS.2008.45

Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec 2013). https://doi.org/10.2168/lmcs-9(4:23)2013

Matija Pretnar. 2010. *Logic and handling of algebraic effects*. Ph.D. Dissertation.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, Washington, DC, USA, 55–74. http://dl.acm.org/citation.cfm?id=645683.664578

H. Schorr and W. M. Waite. 1967. An Efficient Machine-independent Procedure for Garbage Collection in Various List Structures. *Commun. ACM* 10, 8 (Aug. 1967), 501–506. https://doi.org/10.1145/363534.363554

Sam Staton. 2010. Completeness for Algebraic Theories of Local State. In *Foundations of Software Science and Computational Structures*, Luke Ong (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 48–63.

Jean-Pierre Talpin and Pierre Jouvelot. 1992. Polymorphic type, region and effect inference. *Journal of Functional Programming* 2, 3 (1992), 245–271. https://doi.org/10.1017/S0956796800000393