Extended Alias Type System using Separating Implication

Toshiyuki Maeda Haruki Sato Akinori Yonezawa
The University of Tokyo
{tosh,haruki,yonezawa}@yl.is.s.u-tokyo.ac.jp

Abstract

Although explicit memory management is necessary to implement low-level software such as operating systems and language runtime systems, it is prohibited by conventional strictly typed programming languages because it violates the type preservation of memory regions, a property that ensures the type safety of programs. The alias type system allows explicit memory management without the loss of type safety by statically tracking pointers and their aliases. However, it suffers from limitations in handling recursive data structures because it requires complete information about the pointer aliases. In this paper, we propose an extension of the alias type system using separating implications, which are derived from separation logic. Separating implications enable us to handle recursive data structures with incomplete aliasing information by assuming aliasing relations in a part of memory. The proposed type system is capable of expressing tail-recursive operations on recursive data structures. For example, we can implement a FIFO queue with constant-time operations; this cannot be achieved using the original alias type system.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Security, Theory, Verification

Keywords Type system, Alias types, Separating implications, Explicit memory management

1. Introduction

Strictly typed programming languages do not allow programmers to explicitly manage memory regions. Therefore, low-level software such as operating systems and language runtime programs have not been written using strictly typed programming languages because explicit memory management is necessary to implement low-level software.

Strictly typed programming languages prohibit explicit memory management because it conflicts with the type preservation property, which ensures that the types of the allocated memory regions never change during program execution.

To investigate the relation between explicit memory management and type preservation, let us consider the following C function:

```
1: void reuse(int *p, int *q) {
2:     *(int**)p = q;
3: }
```

The memory region that stores an integer value pointed by the pointer `p` is reused to store the pointer to an integer, `q` (line 2). More specifically, the type of the pointer `p` must be updated with `int**` when the memory region is reused; thus, the type is not preserved. In general, it is not easy to ensure type safety without type preservation. For example, the function stated is not type-safe if the two pointers `p` and `q` are aliased (that is, if they point to the same memory region) because not only the type of `p` but also the type of `q` is updated accidentally.

1.1 Alias Type System and Its Drawback

The alias type system [13, 18] enables programmers to achieve both explicit memory management and type safety. It keeps track of aliasing relations between pointers by separating the type of the memory region pointed by a pointer from the type of the pointer and by tracking aliasing relations in the memory regions with the separated types. The function `reuse` is rewritten informally using the alias type system as follows:

```
1: {a ↦ int} ⊗ {b ↦ int}
2: void reuse(ptr(a) p, ptr(b) q) {
3:     *p = q;
4:     // {a ↦ ptr(b)} ⊗ {b ↦ int}
5: }
```

Line 1 denotes the store type (that is, the type of memory regions), and it implies that there are two integer values at `a` and `b`. The types of the pointers `p` and `q` are denoted by `ptr(a)` and `ptr(b)`, respectively, as shown in line 2. The type `ptr(a)` implies that it points to the address `a`. By combining the type of `p` and the store type, we know that `p` is a pointer to an integer.

In the alias type system, we also know that the two pointers `p` and `q` never point to the same memory region because the alias type system prevents all the addresses mentioned in the store type from aliasing each other. Thus, after reusing the memory region at `a`, the store type is updated without the loss of type safety, as shown in line 4.

A drawback of the original alias type system is its limited ability to handle recursive data structures, especially in the implementation of tail-recursive operations. To investigate this drawback, we first explain the treatment of recursive data structures by the original alias type system. Then, we describe the drawback.

The alias type system treats recursive data structures using existential types. For example, in the alias type system, a singly-linked list can be expressed as

$$\texttt{List} \equiv \exists[\rho|\{\rho \mapsto \texttt{List}\}].\texttt{ptr}(\rho)$$

The type stated above is an existential type; it implies that the list is merely a pointer to the address ρ (the exact address of ρ is not known), and another singly-linked list exists at the address ρ. The

TLDI'11, January 25, 2011, Austin, Texas, USA.

alias type system prevents all the addresses mentioned in existential types from aliasing with other addresses.

To investigate the drawback of the original alias type system in handling recursive data structures, let us consider a program that simply traverses a singly-linked list as follows:

```
1: {ρ ↦ List}
2: while (p) { // p : ptr(ρ)
3:     unpack ρ with ρ1;
4:     // {ρ ↦ ptr(ρ1)} ⊗ {ρ1 ↦ List}
5:     p = *p;
6:     // p : ptr(ρ1)
7: }
```

Line 1 denotes the loop invariant, and it implies that there is a singly-linked list at some address ρ. In addition, let us assume that the type of the pointer `p` is `ptr(`ρ`)` (line 2). Line 3 performs the unpack operation on the existential type `List`. More specifically, it extracts the packed memory region from the existential type. Thus, after the unpack operation, the store type consists of the extracted memory region and the pointer to it, as shown in line 4. (It is important to note that the packed ρ is renamed as ρ_1 in order to avoid name ambiguity.) Finally, line 5 updates the variable `p`, and its type becomes `ptr(`ρ_1`)` (line 6).

The drawback of the original alias type system is its inability to loop back from line 6 to line 2. More specifically, although the loop invariant is satisfied with the pointer `p` and the type of the memory region at ρ_1 by instantiating ρ with ρ_1, the aliasing relations of the original memory region at ρ must be discarded. Thus, the original singly-linked list is not accessible after the `while` loop is exited.

One possible approach to overcome this drawback is to refine the loop invariant; however, this approach is not applicable to the original alias type system. For example, let us refine the loop invariant as

$$\{\rho \mapsto \texttt{ptr}(\rho_1)\} \otimes \{\rho_1 \mapsto \texttt{List}\}$$

Clearly, this invariant is not satisfied for the second iteration of the loop. Thus, for the nth iteration of the loop, the loop invariant must be of the following form:

$$\{\rho \mapsto \texttt{ptr}(\rho_1)\} \otimes \{\rho_1 \mapsto \texttt{ptr}(\rho_2)\} \otimes \ldots \otimes \{\rho_n \mapsto \texttt{List}\}$$

However, the original alias type system is not able to express the store type stated above because it cannot keep track of the aliasing relations of the n pointers, where n is unknown.

1.2 Proposed Approach: Separating Implication

To overcome the drawback of the original alias type system, we extend it using separating implications. A separating implication is derived from separation logic [10]; it is a novel store type that is written as

$$C_1 \Rightarrow C_2$$

It can be read as "if the memory regions that have the store type C_1 are added, then the combined memory regions have the store type C_2."

For example, using the proposed alias type system, the traversal of a singly-linked list can be written as follows:

```
1: ({ρi ↦ List} ⇒ {ρ ↦ List}) ⊗ {ρi ↦ List}
2: while (p) {
3:     unpack ρi with ρj;
4:     p = *p;
5:     reserve {ρj ↦ List} for {ρi ↦ ptr(ρj)}
6:     pack ρi;
7:     trans ({ρj ↦ List} ⇒ {ρi ↦ List})
8:           with ({ρi ↦ List} ⇒ {ρ ↦ List});
9:     // ({ρj ↦ List} ⇒ {ρ ↦ List})
10:    // ⊗ {ρj ↦ List}
11:    // p : ptr(ρj)
12: }
```

Line 1 denotes the loop invariant. The separating implication in the loop invariant implies that if there is one singly-linked list at ρ_i, there is another singly-linked list at ρ, or $\rho_i = \rho$. It is important to note the extended alias type system allows implicit aliasing relations between the antecedents and consequents of separating implications, unlike the original alias type system.

In addition to the loop invariant, let us assume that the type of the pointer `p` is `ptr(`ρ`)`, and there exists a singly-linked list at ρ before the `while` loop is entered. This assumption satisfies the loop invariant because substituting ρ_i with ρ yields the following store type, which equals $\{\rho \mapsto \texttt{List}\}$.

$$(\{\rho \mapsto \texttt{List}\} \Rightarrow \{\rho \mapsto \texttt{List}\}) \otimes \{\rho \mapsto \texttt{List}\}$$

After the loop is entered, the `unpack` operation at line 3 updates the store type as follows:

$$(\{\rho_i \mapsto \texttt{List}\} \Rightarrow \{\rho \mapsto \texttt{List}\}) \otimes \{\rho_i \mapsto \texttt{ptr}(\rho_j)\} \otimes \{\rho_j \mapsto \texttt{List}\}$$

This operation is similar to that of the original alias type system.

Then, the `reserve` operation at line 5 introduces a new separating implication into the store type as follows:

$$(\{\rho_i \mapsto \texttt{List}\} \Rightarrow \{\rho \mapsto \texttt{List}\}) \otimes (\{\rho_j \mapsto \texttt{List}\} \Rightarrow (\{\rho_i \mapsto \texttt{ptr}(\rho_j)\} \otimes \{\rho_j \mapsto \texttt{List}\})) \otimes \{\rho_j \mapsto \texttt{List}\}$$

More specifically, it reserves a singly-linked list at ρ_j for the memory region at ρ_i. Thus, the store type $\{\rho_i \mapsto \texttt{ptr}(\rho_j)\}$ is coupled with $\{\rho_j \mapsto \texttt{List}\}$ under the supposition of $\{\rho_j \mapsto \texttt{List}\}$.

Next, the `pack` operation at line 6 restores the unpacked pointer at ρ_i to `List` as follows:

$$(\{\rho_i \mapsto \texttt{List}\} \Rightarrow \{\rho \mapsto \texttt{List}\}) \otimes (\{\rho_j \mapsto \texttt{List}\} \Rightarrow \{\rho_i \mapsto \texttt{List}\}) \otimes \{\rho_j \mapsto \texttt{List}\}$$

This can be regarded as the reverse of the `unpack` operation.

Finally, the `trans` operation at line 7 concatenates the two separating implications, as shown in line 9. The final store type and the type of the pointer `p` clearly satisfy the loop invariant by instantiating ρ_i with ρ_j, without the loss of any information about the memory regions. In addition, after the loop is exited, the original singly-linked list at ρ can be accessed by applying the store type $\{\rho_j \mapsto \texttt{List}\}$ to $(\{\rho_j \mapsto \texttt{List}\} \Rightarrow \{\rho \mapsto \texttt{List}\})$.

As shown in the example presented above, the extended alias type system enables us to express recursive data structures and tail-recursive operations that cannot be expressed using the original alias type system. For example, the proposed type system enables us to express a tail-recursive and destructive list append function and FIFO queues with constant-time operations.

The remainder of the paper is organized as follows. The proposed type system and the adopted imperative language are formally described in Section 2. In order to demonstrate the expressiveness of the proposed type system, several example programs are presented in Section 3. Related work is discussed in Section 4, and finally, the paper is concluded in Section 5.

2. Proposed Type System

Before we explain the proposed type system, we explain the adopted base language. The language is designed on the basis of that used by the original alias type system [18]. Careful readers may notice that the language does not contain explicit looping constructs used in Section 1. Instead, the language supports recur-

$$
\begin{array}{rcl}
v & ::= & x \mid i \mid v[c] \mid \\
 & & \texttt{fix}f[\Delta|C,\Theta](x_1:\sigma_1,\cdots,x_n:\sigma_n).I \mid \\
 & & \texttt{ptr}(\ell) \mid \langle v_1,\cdots,v_2\rangle \mid \varsigma(v) \\
\varsigma & ::= & \texttt{pack}_{[c_1,\cdots,c_n|S]\texttt{as}\exists[\Delta|C,\Theta].\tau} \mid \\
 & & \texttt{roll}_{(\texttt{rec}\ \alpha(\Delta).\tau)(c_1,\cdots,c_n)} \\
I & ::= & \iota; I \mid \texttt{ifpeq}\ v = v\ \texttt{then}\ I_1\ \texttt{else}\ I_2 \mid \\
 & & v(v_1,\cdots,v_n) \\
\iota & ::= & \texttt{new}\ \rho, x, i \mid \texttt{free}\ v \mid \texttt{let}\ x = (v).i \mid \\
 & & (v_1).i := v_2 \mid \texttt{coerce}(\gamma) \\
\gamma & ::= & \texttt{roll}_{\texttt{rec}\ \alpha(\Delta).\tau(c_1,\cdots,c_n)}(\eta) \mid \texttt{unroll}(\eta) \mid \\
 & & \texttt{pack}_{[c_1,\cdots,c_n|C]\texttt{as}\exists[\Delta|C].\tau}(\eta) \mid \\
 & & \texttt{unpack}\ \eta\ \texttt{with}\ \Delta \mid \texttt{reserve}\ C\ \texttt{for}\ C \mid \\
 & & \texttt{indicated}\ C\ \texttt{for}\ (C \Rightarrow C) \mid \\
 & & \texttt{trans}\ (C \Rightarrow C)\ \texttt{with}\ (C \Rightarrow C) \\
S & ::= & s \cdots s \\
s & ::= & \{\ell \mapsto v\} \mid \omega \mid \\
 & & \lambda\omega : C.S \quad (\omega \text{ appears exactly once in } S) \\
\mathcal{M} & ::= & (S, I)
\end{array}
$$

Figure 1. Syntax

sive functions. Loops can be represented as tail-recursive functions (several examples are presented in Section 3).

2.1 Language Syntax

Figure 1 shows the syntax of the adopted base language. $\mathcal{M}$ denotes the state of the abstract machine. The state is represented by the store S and the instruction sequence I. The store denotes the state of the allocated memory regions. More specifically, the store is a set consisting of location-value pairs ($\{\ell \mapsto v\}$), store variables (ω), and store abstractions ($\lambda\omega : C.S$).

The store abstraction $\lambda\omega : C.S$ denotes memory regions represented by S, which has a hole represented by ω. It is important to note that ω appears exactly once in S. For example, the store abstraction $\lambda\omega : C.\{\ell \mapsto v\}\omega$ denotes a memory region that stores the value v at the address ℓ. Within the abstraction, ω is treated as a store of the store type C; however, it is not available for memory accesses because it has no concrete mappings from locations to values. Therefore, ℓ is the only accessible location of the store abstraction.

The instruction sequence I is a sequence of instructions (ι) that ends with a function call or pointer comparison. The instructions consist of memory operations and coercions.

The instruction $\texttt{new}\ \rho, x, i$ allocates a memory region of size i, and it binds the variable x to the pointer that points to the allocated memory region. In addition, it assigns the name ρ to the allocated memory location. On the other hand, the instruction $\texttt{free}\ v$ explicitly deallocates the memory region pointed by the pointer v. The instruction $\texttt{let}\ x = (v).i$ loads the ith element of the tuple pointed by v, and it binds the variable x to the loaded value. The instruction $(v_1).i := v_2$ stores the value v_2 in the ith element of the tuple pointed by the pointer v_1.

The function call $v(v_1,\cdots,v_n)$ executes the instruction sequence pointed by v with the arguments $v_1,\cdots,v_n$. The pointer comparison `ifpeq` is described in Section 2.2.

The coercions (γ) are pseudo operations that only manipulate types; they have no runtime effect. They consist of operations for existential types, recursive types, and separating implications. Existential types are manipulated using `pack` and `unpack`. `pack` creates an existential package by abstracting the type constructors $c_1,\cdots,c_n$ and encapsulating the store specified by the store type C in the package. `unpack` destroys an existential package and extracts the store packed in it. Recursive types are manipulated

$$
\begin{array}{rcl}
\kappa & ::= & \texttt{Loc} \mid \texttt{Store} \mid \texttt{Small} \mid \texttt{Type} \mid \\
 & & (\kappa_1,\cdots,\kappa_n) \rightarrow \texttt{Type} \\
\beta & ::= & \rho \mid \alpha \mid \epsilon \\
\Delta & ::= & \cdot \mid \Delta, \beta : \kappa \\
c & ::= & \eta \mid \tau \mid C \\
\eta & ::= & \rho \mid \ell \\
\Theta & ::= & \cdot \mid \Theta, \eta = \eta \mid \Theta, \eta \neq \eta \\
C & ::= & a \otimes \cdots \otimes a \\
a & ::= & \{\eta \mapsto \tau\} \mid C \Rightarrow C \mid \epsilon \\
\tau & ::= & \alpha \mid \sigma \mid \langle \sigma_1,\cdots,\sigma_n\rangle \mid \\
 & & \exists[\Delta|C,\Theta].\tau \mid \texttt{rec}\ \alpha(\Delta).\tau \mid \tau(c_1,\cdots,c_n) \\
\sigma & ::= & \alpha \mid int \mid ptr(\eta) \mid \\
 & & \forall[\Delta|C,\Theta].(\sigma_1,\cdots,\sigma_n) \rightarrow \mathbf{0}
\end{array}
$$

Figure 2. Type Structure

using `roll` and `unroll`. `unroll` expands the recursive type of a memory region, whereas `roll` folds back the expanded recursive type. The operational semantics of the coercions used to manipulate separating implications are explained in Section 2.2.

Figure 2 shows the type structure of the adopted language. C is the store type that describes the shape of memory regions; it is a set consisting of location-type pairs, implications, and store type variables ϵ.

The store type $\{\eta \mapsto \tau\}$ describes the store $\{\ell \mapsto v\}$ where the type of the value v is τ. η ranges over the locations ℓ and location variables ρ. τ denotes the types of the values (the small types σ, tuple types, existential types, and recursive types). The small types σ denotes integers and pointers.

The store type of the form $C_1 \Rightarrow C_2$ corresponds to the store abstraction $\lambda\omega : C_1.S$, where the type of the store S is C_2, and the type of the store ω is C_1.

Θ denotes a set of equality constraints for the locations η. More specifically, Θ consists of equalities $\eta_1 = \eta_2$ or inequalities $\eta_1 \neq \eta_2$. In the original alias type system, the equality constraints are unnecessary because all the locations in a store type are distinguished by the type system. However, in the proposed system, the locations in the antecedent and consequent of the separating implication may be aliased. Therefore, the equality constraints are necessary to access the store within store abstractions. It is important to note that the function types and existential types are augmented with Θ.

2.2 Operational Semantics

The small-step operational semantics of the adopted language are shown in Figures 3, 4, and 5. They are virtually identical to those used in the original alias type system [18], except for the handling of stores. More specifically, we need to handle accesses to the store abstractions. In addition, we need to provide operational semantics for the coercions that manipulate the store abstractions. The notation $A[X/x]$ denotes the capture-avoiding substitution of X for x in A. $X[c_1,\cdots,c_n/\Delta]$ denotes the capture-avoiding substitution of constructors $c_1,\cdots,c_n$ for the corresponding type variables of Δ.

We explain the operational semantics of memory access operations and coercions for three cases because there are three types of stores in the adopted language.

The first and simplest case involves accessing the store of $\{\ell \mapsto v\}$. In this case, the memory access operations and coercions merely access the value v. This is virtually identical to the working of the original alias type system.

The second case is involves accessing the store of the store variable ω. In this case, the access is invalid because the store variables represent holes in the stores.

$\boxed{\mathcal{M} \longmapsto_P \mathcal{M}'}$

$$
\begin{array}{lll}
(S, \texttt{new}\ \rho, x, i; I) & \longmapsto_P & (S\{\ell \mapsto v\}, I') \\
\text{where} \quad \ell \notin S, v = \langle int, \cdots, int \rangle\ , \text{ and } I' = I[\ell/\rho][\texttt{ptr}(\ell)/x] & &
\end{array}
$$

$$
\begin{array}{lll}
(S, \texttt{ifpeq}\ \texttt{ptr}(\ell_1) = \texttt{ptr}(\ell_2)\ \texttt{then}\ I_1\ \texttt{else}\ I_2) & \longmapsto_P & (S, I) \\
\text{where } I = \left\{ \begin{array}{l} I_1 \text{ when } \ell_1 = \ell_2 \\ I_2 \text{ when } \ell_1 \neq \ell_2 \end{array} \right. & &
\end{array}
$$

$$
\begin{array}{lll}
(S, v(v_1, \cdots, v_n)) & \longmapsto_P & (S, \theta(I)) \\
\qquad v = v'[c_1, \cdots, c_m] & & \\
\text{where} \quad v' = \texttt{fix} f[\Delta | C, M](x_1 : \sigma_1, \cdots, x_n : \sigma_n).I & & \\
\qquad \theta = [c_1, \cdots, c_m/\Delta][v'/f][v_1, \cdots, v_n/x_1, \cdots, x_n] & &
\end{array}
$$

$$
\begin{array}{lll}
(S, \iota; I) & \longmapsto_P & (S', \theta(I)) \\
\text{where} \quad \iota(S) \longmapsto_\iota S', \theta & &
\end{array}
$$

Figure 3. Operational Semantics

$\boxed{\iota(S) \longmapsto_\iota S', \theta}$

$$
\begin{array}{lll}
\texttt{free}\ \texttt{ptr}(\ell)(S\{\ell \mapsto v\}) & \longmapsto_\iota & S, [] \\
\texttt{let}\ x = (\texttt{ptr}(\ell)).i(S\{\ell \mapsto \langle v_1, \cdots, v_i, \cdots, v_n \rangle\}) & \longmapsto_\iota & S\{\ell \mapsto \langle v_1, \cdots, v_i, \cdots, v_n \rangle\}, [v_i/x] \\
\text{where} \quad a \leq i \leq n & & \\
(\texttt{ptr}(\ell)).i := v'(S\{\ell \mapsto \langle v_1, \cdots, v_i, \cdots, v_n \rangle\}) & \longmapsto_\iota & S\{\ell \mapsto \langle v_1, \cdots, v', \cdots, v_n \rangle\}, [] \\
\text{where} \quad a \leq i \leq n & & \\
\texttt{coerce}(\gamma)(S) & \longmapsto_\iota & S', \theta \\
\text{where} \quad \gamma(S) \longmapsto_\gamma S', \theta & & \\
\iota(S(\lambda\omega : C.S')) & \longmapsto_\iota & S(\lambda\omega : C.S''), \theta \\
\qquad \cdot; \cdot \vdash \text{target}(\iota) = \mathbf{L} & & \\
\text{where} \quad \mathbf{L} \subseteq Dom(S') & & \\
\qquad \iota(S') \longmapsto_\iota S'', \theta & &
\end{array}
$$

Figure 4. Operational Semantics: Instruction

$\boxed{\gamma(S) \longmapsto_\gamma S', \theta}$

$$
\begin{array}{lll}
\texttt{roll}_\tau(\ell)(S\{\ell \mapsto v\}) & \longmapsto_\gamma & S\{\ell \mapsto \texttt{roll}_\tau(v)\}, [] \\
\texttt{unroll}(\ell)(S\{\ell \mapsto \texttt{roll}_\tau(v)\}) & \longmapsto_\gamma & S'\{\ell \mapsto v\}, [] \\
\texttt{pack}_{[c_1, \cdots, c_n | C, \Theta] \texttt{as} \tau}(\ell)(S\{\ell \mapsto v\}S') & \longmapsto_\gamma & S\{\ell \mapsto \texttt{pack}_{[c_1, \cdots, c_n | S'] \texttt{as} \tau}(v)\}, [] \\
\text{where} \quad Dom(S') = Dom(C) & & \\
\texttt{unpack}\ \ell\ \texttt{with}\ \Delta(S\{\ell \mapsto \texttt{pack}_{[c_1, \cdots, c_n | S'] \texttt{as} \exists [\Delta | C, \Theta].\tau}(v)\}) & \longmapsto_\gamma & S\{\ell \mapsto v\}S', [c_1, \cdots, c_n/\Delta] \\
\texttt{reserve}\ C_1\ \texttt{for}\ C_2(SS_2) & \longmapsto_\gamma & S(\lambda\omega : C_1.S_2\omega), [] \\
\text{where} \quad Dom(S_2) = Dom(C_2) & & \\
\texttt{indicated}\ C_1\ \texttt{for}\ C_1 \Rightarrow C_2(S(\lambda\omega : C_1.S_2')S_1) & \longmapsto_\gamma & S(S_2'[S_1/\omega]), [] \\
\text{where} \quad \begin{array}{l} Dom(\lambda\omega : C_1.S_2') = Dom(C_1 \Rightarrow C_2) \\ Dom(S_1) = Dom(C_1) \end{array} & &
\end{array}
$$

Figure 5. Operational Semantics: Coercions

The third and final case involves accessing the store abstraction $\lambda\omega : C.S$. The abstraction may include other stores besides ω; hence, the store S is accessed recursively. The rule at the bottom of Figure 4 represents recursive access to the store abstractions. In this rule, $Dom(S)$ denotes the accessible locations of the store S. $Dom(S)$ is defined as follows:

$$\begin{array}{lcl} Dom(\cdot) & = & \emptyset \\ Dom(s_1 \cdots s_n) & = & Dom(s_1) \oplus \cdots \oplus Dom(s_n) \\ Dom(\{\ell \mapsto v\}) & = & \{\ell\} \\ Dom(\lambda\omega : C.S') & = & Dom(S') \\ Dom(\omega) & = & \emptyset \end{array}$$

Here, $\oplus$ denotes the disjoint union of sets of locations. Additionally, target(ι) denotes the locations to be accessed by ι. Figure 6 shows the definition of target(ι).

Except for the memory access conditions described above, the operational semantics of instructions and coercions are fairly straightforward, as in the case of the original alias type system [18]. Therefore, we only explain the newly introduced instructions: `reserve`, `indicated`, `trans` and `ifpeq`.

The coercion `reserve` introduces store abstractions. For example, let us assume that the current store is

$$\{\ell_1 \mapsto \langle \mathtt{ptr}(\ell_2)\rangle\}$$

The coercion `reserve` $\{\ell_2 \mapsto \tau\}$ `for` $\{\ell_1 \mapsto \langle ptr(\ell_2)\rangle\}$ translates it as

$$(\lambda\omega : \{\ell_2 \mapsto \tau\}.\{\ell_1 \mapsto \langle \mathtt{ptr}(\ell_2)\rangle\}\omega)$$

In the store type stated above, a hole ω of the store type $\{\ell_2 \mapsto \tau\}$ is introduced by the `reserve` operation.

On the other hand, the coercion `indicated` eliminates store abstractions. For example, let us assume that the current store is

$$(\lambda\omega : \{\ell_2 \mapsto \tau\}.\{\ell_1 \mapsto \mathtt{pack}_{[|\omega|]\mathtt{as}\tau'}(v')\})\{\ell_2 \mapsto v\}$$

where the type of v is τ. The coercion `indicated` $\{\ell_2 \mapsto \tau\}$ `for` $\{\ell_2 \mapsto \tau\} \Rightarrow \{\ell_1 \mapsto \tau'\}$ translates it as

$$\{\ell_1 \mapsto \mathtt{pack}_{[|\{\ell_2 \mapsto v\}|]\mathtt{as}\tau'}(v')\})$$

It is important to note that the store variable can be instantiated, even though it may be packed with the coercion `pack`, as shown above.

The coercion operation `trans` is not included in Figure 5 because it can be represented by combining `reserve` and `indicated`. More specifically, `trans` $C_1 \Rightarrow C_2$ `with` $C_2 \Rightarrow C_3$ can be rewritten as follows:

`reserve` C_1 `for` $(C_1 \Rightarrow C_2) \otimes (C_2 \Rightarrow C_3)$;
`indicated` C_1 `for` $C_1 \Rightarrow C_2$;
`indicated` C_2 `for` $C_2 \Rightarrow C_3$

For example, if we apply the coercions stated above to the store $(\lambda\omega_1 : C_1.S_2)\ (\lambda\omega_2 : C_2.S_3)$, they translate it as

$$(\lambda\omega_1' : C_1.S_3[(S_2[\omega_1'/\omega_1])/\omega_2])$$

The instruction sequence `ifpeq` $v_1 = v_2$ `then` I_1 `else` I_2 compares the two given pointers (v_1 and v_2). If $v_1 = v_2$, `ifpeq` executes I_1; otherwise, it executes I_2. `ifpeq` is essential for accessing locations in store abstractions because it can recover the aliasing relations in the store abstractions that may be disregarded by the proposed type system. More specifically, it introduces the equality constraints Θ, as describe in Section 2.2.1.

2.2.1 Typing Rules

The important typing rules of the proposed type system are shown in Figures 7, 8, and 9; they are based on those of the original alias type system [18]. The proposed type system provides type safety by ensuring that the following theorem holds:

$\boxed{\Delta; \Gamma \vdash \text{target}(\iota) = \eta}$

$$\frac{\iota = \mathtt{free}\ v \mid \mathtt{let}\ \_ = (v).\_ \mid (v).\_ := \_ \qquad \Delta; \Gamma \vdash v : ptr(\eta)}{\Delta; \Gamma \vdash \text{target}(\iota) = \{\eta\}}$$

$$\frac{\gamma = \mathtt{roll}\_(\eta) \mid \mathtt{unroll}(\eta) \mid \mathtt{pack}\_\mathtt{as}\_(\eta) \mid \mathtt{unpack}\ \eta\ \mathtt{with}\ \_}{\Delta; \Gamma \vdash \text{target}(\mathtt{coerce}(\gamma)) = \{\eta\}}$$

$$\frac{\gamma = \mathtt{reserve}\ \_\ \mathtt{for}\ \_ \mid \mathtt{indicated}\ \_\ \mathtt{for}\ \_}{\Delta; \Gamma \vdash \text{target}(\mathtt{coerce}(\gamma)) = \{\}}$$

Figure 6. Target Location of Instruction

THEOREM 1 (Type Soundness). *If* $\vdash \mathcal{M}$, *there exists* $\mathcal{M}'$, *where* $\mathcal{M} \longmapsto_P^* \mathcal{M}'$ *and* $\vdash \mathcal{M}'$.

Well-formedness of the states of the abstract machine is defined as follows:

DEFINITION 1. $\vdash (S, I)$ *iff*

1. *There are no duplicate locations in* S, *including the packed stores.*
2. *There exists a store type* C *such that* $\cdot \vdash S : C$
3. $\cdot; C; \cdot \vdash I$

Well-formedness of the stores is defined as follows:

$$\frac{\Sigma \vdash s_1 : a_1 \quad \cdots \quad \Sigma \vdash s_n : a_n}{\Sigma \vdash s_1 \cdots s_n : a_1 \otimes \cdots \otimes a_n}$$

$$\frac{\Sigma \vdash v : \tau}{\Sigma \vdash \{\ell \mapsto v\} : \{\ell \mapsto \tau\}} \qquad \frac{}{\Sigma \vdash \omega : \Sigma(\omega)}$$

$$\frac{\Sigma, \omega : \{\eta \mapsto \tau\} \vdash S_2 : C_2}{\Sigma \vdash \lambda\omega : \{\eta \mapsto \tau\}.S_2 : \{\eta \mapsto \tau\} \Rightarrow C_2} \qquad \frac{\cdot; \cdot \vdash v : \tau}{\Sigma \vdash v : \tau}$$

$$\frac{\cdot; \cdot \vdash \tau = (\mathtt{rec}\ \alpha(\Delta).\tau')(c_1, \cdots .c_n) : \mathtt{Type} \qquad \Sigma \vdash v : \tau'[\mathtt{rec}\ \alpha(\Delta).\tau'/\alpha][c_1, \cdots, c_n/\Delta]}{\Sigma \vdash \mathtt{roll}_\tau(v) : \tau}$$

$$\frac{\begin{array}{c}\Delta = \beta_1 : \kappa_1, \cdots, \beta_n : \kappa_n \qquad \cdot \vdash c_i : \kappa_i \\ \cdot; \cdot \vdash \Theta[c_1, \cdots, c_n/\Delta] \\ \Sigma \vdash S : C[c_1, \cdots, c_n/\Delta] \qquad \cdot; \cdot \vdash v : \tau[c_1, \cdots, c_n/\Delta]\end{array}}{\Sigma \vdash \mathtt{pack}_{[c_1, \cdots, c_n | S]\mathtt{as}\exists[\Delta|C,\Theta].\tau}(v) : \tau}$$

Here, Σ denotes an environment for store variables. It is important to note here that store variables (ω) do not affect disjointness of locations in S because store variables denote holes in S, not locations. In addition, instantiation of store variables preserves the disjointness because each store variable appears exactly once in S.

As shown above, the separating implication $C_1 \Rightarrow C_2$ corresponds to the store abstraction $\lambda\omega : C_1.S_2$. It is important to note that the store type of the store variable within the store abstraction must be of the form $\{\eta \mapsto \tau\}$. This limitation can be relaxed; however, it is sufficient to describe tail-recursive operations on recursive data structures. The reason why separating implications ($C_1 \Rightarrow C_2$) are not allowed is that, in the proposed language, store applications cannot be represented as the store constructs, unlike store abstractions. The reason why store type variables (ϵ) are not allowed is that, although the proposed type system keeps track of equality constraints for locations (Θ), it does not consider inclusion relations between locations and store type variables.

The typing rules for memory accesses are shown in Figure 8. Let us consider the typing rule for the instruction `let` $x = (v).i$. Two possible rules can be applied on the basis of its target location.

$\boxed{\Delta; C; \Theta; \Gamma \vdash \iota \Longrightarrow \Delta'; C'; \Theta'; \Gamma'}$

$$\frac{\Delta; \Gamma \vdash v : ptr(\eta) \qquad \Delta; \Theta \vdash C = C' \otimes \{\eta \mapsto \langle \sigma_1, \cdots, \sigma_i, \cdots, \sigma_n \rangle\} : \texttt{Store}}{\Delta; C; \Theta; \Gamma \vdash \texttt{let}\ x = (v).i \Longrightarrow \Delta; C; \Theta; \Gamma, x : \sigma_i} \left(\begin{array}{l} 1 \leq i \leq n \\ x \notin Dom(\Gamma) \end{array} \right)$$

$$\frac{\Delta; \Gamma \vdash v_1 : ptr(\eta) \qquad \Delta; \Gamma \vdash v_2 : \sigma \qquad \Delta; \Theta \vdash C = C' \otimes \{\eta \mapsto \langle \sigma_1, \cdots, \sigma_i, \cdots, \sigma_n \rangle\} : \texttt{Store}}{\Delta; C; \Theta; \Gamma \vdash (v_1).i := v_2 \Longrightarrow \Delta; C' \otimes \{\eta \mapsto \langle \sigma_1, \cdots, \sigma, \cdots, \sigma_n \rangle\}; \Theta; \Gamma} \left(\ 1 \leq i \leq n \ \right)$$

$$\frac{\Delta; \Theta \vdash \tau = (\texttt{rec}\ \alpha(\Delta').\tau')(c_1, \cdots, c_n) : \texttt{Type} \qquad \Delta; \Theta \vdash C = C' \otimes \{\eta \mapsto \tau'[\texttt{rec}\ \alpha(\Delta').\tau'/\alpha][c_1, \cdots, c_n/\Delta']\} : \texttt{Store}}{\Delta; C; \Theta; \Gamma; \vdash \texttt{coerce}(\texttt{roll}_\tau(\eta)) \Longrightarrow \Delta; C' \otimes \{\eta \mapsto \tau\}; \Theta; \Gamma}$$

$$\frac{\Delta; \Theta \vdash C = C' \otimes \{\eta \mapsto \tau\} : \texttt{Store} \qquad \Delta; \Theta \vdash \tau = (\texttt{rec}\ \alpha(\Delta').\tau')(c_1, \cdots, c_n) : \texttt{Type}}{\Delta; C; \Theta; \Gamma; \vdash \texttt{coerce}(\texttt{unroll}(\eta)) \Longrightarrow \Delta; C' \otimes \{\eta \mapsto \tau'[\texttt{rec}\ \alpha(\Delta').\tau'/\alpha][c_1, \cdots, c_n/\Delta']\}; \Theta; \Gamma}$$

$$\frac{\begin{array}{c}\Delta' = \beta_1 : \kappa_1, \cdots, \beta_n : \kappa_n \qquad \cdot \vdash c_i : \kappa_i \text{ (for } 1 \leq i \leq n) \qquad \Delta; \Theta \vdash \Theta'[c_1, \cdots, c_n/\Delta'] \\ \Delta; \Theta \vdash C = C'' \otimes \{\eta \mapsto \tau[c_1, \cdots, c_n/\Delta']\} \otimes C'[c_1, \cdots, c_n/\Delta'] : \texttt{Store}\end{array}}{\Delta; C; \Theta; \Gamma \vdash \texttt{coerce}(\texttt{pack}_{[c_1, \cdots, c_n | C'[c_1, \cdots, c_n/\Delta'], \Theta'[c_1, \cdots, c_n/\Delta']]\texttt{as}\exists[\Delta' | C', \Theta'].\tau}(\eta)) \Longrightarrow \Delta; C'' \otimes \{\eta \mapsto \exists[\Delta' | C', \Theta'].\tau\}; \Theta; \Gamma}$$

$$\frac{\Delta; \Theta \vdash C = C'' \otimes \{\eta \mapsto \exists[\Delta' | C', \Theta'].\tau\} : \texttt{Store}}{\Delta; C; \Theta; \Gamma \vdash \texttt{coerce}(\texttt{unpack}\ \eta\ \texttt{with}\ \Delta') \Longrightarrow \Delta, \Delta'; C'' \otimes \{\eta \mapsto \tau\} \otimes C'; \Theta\Theta'; \Gamma}$$

$$\frac{\Delta; \Theta \vdash C = C' \otimes C_2 : \texttt{Store}}{\Delta; C; \Theta; \Gamma \vdash \texttt{coerce}(\texttt{reserve}\ \{\eta \mapsto \tau\}\ \texttt{for}\ C_2) \Longrightarrow \Delta; C' \otimes (\{\eta \mapsto \tau\} \Rightarrow C_2 \otimes \{\eta \mapsto \tau\}); \Theta; \Gamma}$$

$$\frac{\Delta; \Theta \vdash C = C' \otimes C_1 \otimes (C_1 \Rightarrow C_2) : \texttt{Store}}{\Delta; C; \Theta; \Gamma \vdash \texttt{coerce}(\texttt{indicated}\ C_1\ \texttt{for}\ (C_1 \Rightarrow C_2)) \Longrightarrow \Delta; C' \otimes C_2; \Theta; \Gamma}$$

$$\frac{\Delta' \equiv \Delta, \rho : \texttt{Loc} \quad C' \equiv C \otimes \{\rho \mapsto \overbrace{\langle int, \cdots, int \rangle}^{i}\} \quad \Gamma' \equiv \Gamma, x : ptr(\rho)}{\Delta; C; \Theta; \Gamma \vdash \texttt{new}\ x, \rho, i \Longrightarrow \Delta'; C'; \Theta; \Gamma'} \left(\begin{array}{l} \rho \notin Dom(\Delta) \\ x \notin Dom(\Gamma) \end{array} \right)$$

$$\frac{\Delta; \Gamma \vdash v : ptr(\eta) \qquad \Delta; \Theta \vdash C = C' \otimes \{\eta \mapsto \tau\} : \texttt{Store}}{\Delta; C; \Theta; \Gamma; \vdash \texttt{free}\ v \Longrightarrow \Delta; C'; \Theta; \Gamma}$$

$$\frac{\begin{array}{c}\Delta; \Theta \vdash C = C' \otimes (C_1 \Rightarrow C_2) \qquad \Delta; \Gamma \vdash \text{target}(\iota) = \mathbf{L} \qquad \forall \eta \in \mathbf{L}. \exists \eta' \in Dom_\Theta(C_1 \Rightarrow C_2). \Theta \vdash \eta = \eta' \\ \Delta; C_2; \Theta; \Gamma \vdash \iota \Longrightarrow \Delta'; C_2'; \Theta'; \Gamma'\end{array}}{\Delta; C; \Theta; \Gamma \vdash \iota \Longrightarrow \Delta'; C' \otimes (C_1 \Rightarrow C_2'); \Theta'; \Gamma'}$$

Figure 8. Typing Rules for Instructions

$\boxed{\Delta; C; \Theta; \Gamma \vdash I}$

$$\frac{\Delta; C; \Theta; \Gamma \vdash \iota \Longrightarrow \Delta'; C'; \Theta'; \Gamma' \qquad \Delta'; C'; \Theta'; \Gamma' \vdash I}{\Delta; C; \Theta; \Gamma \vdash \iota; I}$$

$$\frac{\Delta; \Gamma \vdash v_1 : ptr(\eta_1) \qquad \Delta; \Gamma \vdash v_2 : ptr(\eta_2) \qquad \Delta; C; \Theta, (\eta_1 = \eta_2); \Gamma \vdash I_1 \qquad \Delta; C; \Theta, (\eta_1 \neq \eta_2); \Gamma \vdash I_2}{\Delta; C; \Theta; \Gamma \vdash \texttt{ifpeq}\ v_1 = v_2\ \texttt{then}\ I_1\ \texttt{else}\ I_2}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall[\cdot | C', \Theta'].(\sigma_1, \cdots, \sigma_n) \to \mathbf{0} \qquad \Delta; \Theta \vdash C = C' : \texttt{Store} \\ \Delta; \Theta \vdash \Theta' \qquad \Delta; \Gamma \vdash v_1 : \sigma_1 \quad \cdots \quad \Delta; \Gamma \vdash v_n : \sigma_n\end{array}}{\Delta; C; \Theta; \Gamma \vdash v(v_1, \cdots, v_n)}$$

Figure 9. Typing Rules for Instruction Sequences

$\boxed{\Delta; \Gamma \vdash v : \tau}$

$$\frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Delta; \Gamma \vdash i : int}$$

$$\frac{\begin{array}{c}\Delta, \Delta'; C'; \Theta'; f : \sigma_f, x_1 : \sigma_1, \cdots, x_n : \sigma_n \vdash I \\ \text{(where store type variables do not appear in} \\ \text{antecedents of separating implications in } C')\end{array}}{\begin{array}{c}\Delta; \Gamma \vdash \texttt{fix} f[\Delta'|C', \Theta'](x_1 : \sigma_1, \cdots, x_n : \sigma_n).I : \sigma_f \\ (\sigma_f = \forall[\Delta'|C', \Theta'].(\sigma_1, \cdots, \sigma_n) \rightarrow \mathbf{0})\end{array}}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall[\beta : \kappa, \Delta'|C', \Theta'].(\sigma_1, \cdots, \sigma_n) \rightarrow \mathbf{0} \\ \Delta \vdash c : \kappa\end{array}}{\Delta; \Gamma \vdash v[c] : (\forall[\Delta'|C', \Theta'].(\sigma_1, \cdots, \sigma_n) \rightarrow \mathbf{0})[c/\beta]}$$

Figure 7. Typing Rules for Values

If the target location is contained in a separating implication, the last rule of Figure 8 is applied. This rule states that we can temporarily discard the antecedent of the separating implication if the target location is accessible according to the separating implication. More specifically, this rule checks whether the target location is contained in the domain of the store type $C_1 \Rightarrow C_2$ under the equality constraints Θ. Then, it checks the instruction with the store type C_2. Finally, it checks the rest of the instruction sequence with the updated store type $C_1 \Rightarrow C_2'$. $Dom_\Theta(C)$ is defined as follows:

$$\begin{array}{lcl} Dom_\Theta(\cdot) & = & \emptyset \\ Dom_\Theta(a_1 \otimes \cdots) & = & Dom_\Theta(a_1) \oplus \cdots \\ Dom_\Theta(\{\eta \mapsto \tau\}) & = & \{\eta\} \\ Dom_\Theta(C_1 \Rightarrow C_2) & = & \{\eta \in Dom_\Theta(C_2) | \\ & & \quad \forall \eta' \in Dom_\Theta(C_1).\Theta \vdash \eta \neq \eta'\} \\ Dom_\Theta(\epsilon) & = & \emptyset \end{array}$$

If the target location is not contained in a separating implication, the second rule of Figure 8 is applied. It checks whether the pointer v points to a tuple by examining the store type. Then, it checks whether the size of the tuple satisfies the access offset i. Finally, it binds the variable x to the type of the ith element of the tuple, and it checks the rest of the instruction sequence.

The typing rule for the instruction $(v_1).i := v_2$ governs the strong update, that is, it changes the types of the memory regions. It first checks whether there exists a tuple at the location η by examining the current store type; it also checks whether the size of the tuple satisfies the specified access offset i. Then, it updates the type of the ith element of the tuple with the type of v_2, and it checks the rest of the instruction sequence with the updated tuple type. This strong update is safe because the proposed type system prevents the location η from aliasing with other locations in the store type. Strictly speaking, the proposed type system allows locations in the consequent of a separating implication to be aliased with locations in its antecedent. The aliasing relations in separating implications are handled using the last rule of Figure 8, as described above.

The typing rules for `roll` and `unroll` govern recursive types. The rule for `roll` first unrolls the given recursive type ($\texttt{rec}\ \alpha(\Delta').\tau'$) with the given type constructors $(c_1, \cdots, c_n)$. Then, it checks whether the location η holds the unrolled type. Finally, it updates the type at the location η with the given type τ, and it checks the rest of the instruction sequence with the updated store type. The rule for `unroll` first checks whether the location η holds a recursive type. Then, it unrolls the recursive type and updates the type of the location η with the unrolled type. Finally, it checks the rest of the instruction sequence with the updated store type. It is important to note that the last rule of Figure 8 can be used before applying these rules, as in the case of memory accesses,

The typing rules for `pack` and `unpack` govern existential types. As in the case of the typing rules for recursive types, the last rule of Figure 8 can be used before applying these rules. The rule for `pack` first instantiates the packed type τ with the given type constructors $(c_1, \cdots, c_n)$, and it checks whether the location η holds the instantiated type. Next, it checks whether the current store type contains the store type to be packed. In addition, it checks whether the equality constraints specified in the given existential type are satisfied by the current equality constraints ($\Delta; \Theta \vdash \Theta'[c_1, \cdots, c_n/\Delta']$). Then, it updates the type of the location η with the existential type, and it removes the packed store from the current store type. Finally, it checks the rest of the instruction sequence with the updated store type. The rule for `unpack` first checks whether the location η holds an existential type. Next, it extracts the packed store type (C') by unpacking the existential type, and it adds the store type to the current store type. Finally, it checks the rest of the instruction sequence with the extended store type. It is important to note that the equality constraints are also extended with the packed equality constraints (Θ').

The typing rules for `reserve` and `indicated` govern separating implications. The rule for `reserve` is the introduction rule of the separating implication. It basically checks nothing; however, it states that the antecedent of the implication to be introduced must be of the form of $\{\eta \mapsto \tau\}$. The reason for this limitation is the same as explained in Section 2.2.1. Then, it checks the subsequent instruction sequence with the introduced separating implication. The rule for `indicated` is the elimination rule of the separating implication. It checks whether the antecedent of the given implication equals the given store type. Then, it extracts the consequent of the implication, and it checks the subsequent instruction sequence with the extracted store type.

The typing rules for `new` and `free` govern explicit memory allocation and deallocation. The rule for `new` first extends the current store type with the store type that corresponds to the allocated memory region. Then, it checks the rest of the instruction sequence with the extended store type. The rule for `free` first checks whether the given pointer points to the valid memory region by examining the current store type. Then, it removes the store type that corresponds to the memory region to be freed, and it checks the rest of the instruction sequence with the updated store type.

The equality constraints Θ are introduced by the typing rule for `ifpeq`, as shown in Figure 9. The rule first checks whether both the variables v_1 and v_2 are pointers. Then, it checks the instruction sequences I_1 and I_2 under the condition that $v_1 = v_2$ and $v_1 \neq v_2$, respectively. For example, let us consider a store of the type $\{\rho_1 \mapsto \tau\} \Rightarrow \{\rho_2 \mapsto \tau\}$. The proposed type system does not keep track of aliasing relations between ρ_1 and ρ_2; therefore, we need to explicitly check the aliasing relations by using `ifpeq`. At first glance, this approach may seem superfluous; however, it is essential and well suited to typical programming styles, as described in Section 3.

The typing rule for function calls is the last rule of Figure 9. It checks whether the type of the value v is the instruction type and whether the types of the values $v_1, \cdots, v_n$ are the types specified in the instruction type. It also checks whether the store type and equality constraints specified in the instruction type are satisfied by the current store type and equality constraints.

3. Examples

In this section, we present several examples in order to demonstrate the expressiveness of the proposed type system. The obvious coercions are omitted for brevity.

```
 1 : fix append[ρh, ρp, ρx, ρy, ϵ|
 2 :   ({ℓ0 ↦ List} ⇒
 3 :     ({ρp ↦ List} ⇒ {ρh ↦ List})
 4 :     ⊗{ρp ↦ ⟨ptr(ρx)⟩} ⊗ {ρx ↦ List})
 5 :   ⊗({ℓ0 ↦ List} ⇒ {ρy ↦ List}) ⊗ ϵ,
 6 :   ℓ0 ≠ ρp]
 7 :   (h : ptr(ρh), p : ptr(ρp), y : ptr(ρy), cont : τc[ρh, ϵ]).
 8 :   let x = (p).1;
 9 :   ifpeq x = ptr(ℓ0) then
10 :     (p).1 := y;
11 :     last_coercions;
12 :     cont(h);
13 :   else
14 :     unpack ρx with ρxs;
15 :     reserve {ρx ↦ List} for {ρp ↦ ⟨ptr(ρx)⟩};
16 :     pack[ρx|{ρx↦List}]asList(ρp);
17 :     trans {ρx ↦ List} ⇒ {ρp ↦ List} with
18 :       {ρp ↦ List} ⇒ {ρh ↦ List};
19 :     append[ρh, ρx, ρxs, ρy, ϵ](h, x, y, cont);
where
20 : List ≡ ∃[ρ|{ρ ↦ List}]. ⟨ptr(ρ)⟩
21 : τc[ρ, ϵ] ≡ ∀[|{ℓ0 ↦ List} ⇒
22 :                 {ρ ↦ List} ⊗ ϵ].(a : ptr(ρ)) → 0
23 : last_coercions ≡
24 :   reserve {ℓ0 ↦ List} for Call;
25 :   indicated {ℓ0 ↦ List} for
26 :     {ℓ0 ↦ List} ⇒
27 :       ({ρp ↦ List} ⇒ {ρh ↦ List})
28 :       ⊗{ρp ↦ ⟨ptr(ρy)⟩} ⊗ {ℓ0 ↦ List};
29 :   indicated {ℓ0 ↦ List} for
30 :     {ℓ0 ↦ List} ⇒ {ρy ↦ List};
31 :   pack[ρy|{ρy↦List}]asList(ρp);
32 :   indicated {ρp ↦ List} for
33 :     {ρp ↦ List} ⇒ {ρh ↦ List};
```

Figure 10. Tail-Recursive Append

3.1 List Append

Figure 10 shows a destructive and tail-recursive append function for null-terminated lists. It takes four arguments (shown in line 7). The argument h is a pointer to one list, and the argument y is a pointer to the other list that is destructively appended to the end of the first list. The argument p is a pointer to an element of the first list, and it is incremented each time the function is recursively called. The variable *cont* is a continuation.

As shown in line 5, the type of the lists is basically denoted by $\{\ell_0 \mapsto List\} \Rightarrow \{\rho_y \mapsto List\}$. The implication implies that the list at the location ρ_y is null-terminated. More specifically, the list can be accessed if the location ρ_y is not equal to ℓ_0, as described in the typing rules of Section 2.2.1. Here, we assume that the location ℓ_0 denotes the null pointer, that is, ℓ_0 is never available during program execution. It is important to note that this representation of null-terminated lists is more intuitive than that of the original alias type system, which uses the variant types [18].

The function first loads the pointer to the next element in the first list by using p (line 8), and it binds the variable x to it. If the pointer x is null, that is, if the end of the first list is reached, the second list is appended to the end of the first list by storing the pointer y in the last element (line 10). Then, the store type is adjusted via several coercion operations (line 11). After the coercion `reserve` (line 24), the entire store type is translated as follows (C_{all} denotes the entire store type here):

$$\begin{array}{l}\{\ell_0 \mapsto List\} \Rightarrow \\ \quad \{\ell_0 \mapsto List\} \otimes (\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_p \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes\{\rho_p \mapsto \langle ptr(\rho_y)\rangle\} \otimes \{\ell_0 \mapsto List\}) \\ \quad \otimes(\{\ell_0 \mapsto List\} \Rightarrow \{\rho_y \mapsto List\}) \otimes \epsilon\end{array}$$

Then, the coercion `indicated` is performed (line 25), and the store type is translated as follows:

$$\begin{array}{l}\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_p \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes\{\rho_p \mapsto \langle ptr(\rho_y)\rangle\} \otimes \{\ell_0 \mapsto List\} \\ \quad \otimes(\{\ell_0 \mapsto List\} \Rightarrow \{\rho_y \mapsto List\}) \otimes \epsilon\end{array}$$

Next, the coercion `indicated` is performed again (line 29), and the store type is translated as follows:

$$\begin{array}{l}\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_p \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes\{\rho_p \mapsto \langle ptr(\rho_y)\rangle\} \otimes \{\rho_y \mapsto List\} \otimes \epsilon\end{array}$$

Now, the location ρ_p is packed (line 31), and the store type is translated as follows:

$$\begin{array}{l}\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_p \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes\{\rho_p \mapsto List\} \otimes \epsilon\end{array}$$

Next, the coercion `indicated` is performed (line 32), and the store type is translated as follows:

$$\{\ell_0 \mapsto List\} \Rightarrow \{\rho_h \mapsto List\} \otimes \epsilon$$

Finally, the function returns the appended list (line 12).

If the pointer x is not null, the sublist of the first list pointed by x is unpacked (line 14). Then, the entire store type is translated as follows:

$$\begin{array}{l}(\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_p \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes\{\rho_p \mapsto \langle ptr(\rho_x)\rangle\} \otimes \{\rho_x \mapsto \langle ptr(\rho_{xs})\rangle\} \otimes \{\rho_{xs} \mapsto List\}) \\ \quad \otimes(\{\ell_0 \mapsto List\} \Rightarrow \{\rho_y \mapsto List\}) \otimes \epsilon\end{array}$$

Next, the store type $\{\rho_x \mapsto List\}$ is reserved for $\{\rho_p \mapsto \langle ptr(\rho_x)\rangle\}$, and the location ρ_p is packed to *List* (lines 15 and 16, respectively). Then, the store type is translated as follows:

$$\begin{array}{l}(\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_p \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes(\{\rho_x \mapsto List\} \Rightarrow \{\rho_p \mapsto List\}) \\ \quad \otimes\{\rho_x \mapsto \langle ptr(\rho_{xs})\rangle\} \otimes \{\rho_{xs} \mapsto List\}) \\ \quad \otimes(\{\ell_0 \mapsto List\} \Rightarrow \{\rho_y \mapsto List\}) \otimes \epsilon\end{array}$$

Next, the `trans` operation (lines 17 and 18) is performed, and the entire store type is finally translated as follows:

$$\begin{array}{l}(\{\ell_0 \mapsto List\} \Rightarrow \\ \quad (\{\rho_x \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \\ \quad \otimes\{\rho_x \mapsto \langle ptr(\rho_{xs})\rangle\} \otimes \{\rho_{xs} \mapsto List\}) \\ \quad \otimes(\{\ell_0 \mapsto List\} \Rightarrow \{\rho_y \mapsto List\}) \otimes \epsilon\end{array}$$

Finally, the function calls itself tail-recursively (line 19). The tail-recursive call passes the type check because the entire store type satisfies the precondition of the append function by instantiating ρ_p with ρ_x and ρ_x with ρ_{xs}. In addition, its equality constraint is also satisfied because $\rho_x \neq \ell_0$, as derived from the `ifpeq` operation (line 9).

It is important to note here that the destructive list append function presented in this section is fully tail-recursive, unlike the one presented in [18]. As stated in [18], the list append function of [18] is not fully tail-recursive because each time the function calls itself recursively, it creates a new continuation by wrapping

the passed continuation with several type coercions. Although the coercions can be erased at compilation time (as indicated in [18]), they cannot be omitted in the original alias type system.

At first glance, it seems to be possible to make the list append function of [18] fully tail-recursive, for example, by introducing an additional pointer that always points to the head of the original list. However, this is not the case in the original alias type system because it prohibits aliasing of locations in store types entirely. More specifically, a recursive data structure with two (or more) pointers that may point to the different elements cannot be implemented in a straightforward way.

For example, let us consider a list and a pair of pointers. In addition, let us also consider that the first pointer of the pair always points to the head of the list, and the second pointer points to one of the elements in the list. First, assume that both the pointers point to the head of the list as follows:

$$\{\rho_p \mapsto \langle ptr(\rho_h), ptr(\rho_h)\rangle\} \otimes \{\rho_h \mapsto \mathit{List}\}$$

Next, let us consider to make the second pointer point to the second element of the list. In order to obtain the location of the second element of the list, it is necessary to unpack the location ρ_h. Then, the store type is updated as follows:

$$\{\rho_p \mapsto \langle ptr(\rho_h), ptr(\rho'_h)\rangle\} \\ \otimes\{\rho_h \mapsto \langle ptr(\rho_{h'})\rangle\} \otimes \{\rho_{h'} \mapsto \mathit{List}\}$$

At this point, although the pointer to the first element of the list is not modified, the store type forgets that the pointer points to the list. In order to revert the type of the location ρ_h, it is necessary to pack the location ρ_h. Then, the store type is translated as follows:

$$\{\rho_p \mapsto \langle ptr(\rho_h), ptr(\rho'_h)\rangle\} \otimes \{\rho_h \mapsto \mathit{List}\}$$

Now, the store type remembers that the first pointer of the pair points to the first element of the list. However, the store type forgets that the second pointer points to the second element of the list.

On the other hand, in the proposed type system, the list and the pair of the pointers can be concisely represented using a separating implication as follows:

$$\{\rho_p \mapsto \langle ptr(\rho_h), ptr(\rho'_h)\rangle\} \\ \otimes(\{\rho_{h'} \mapsto \mathit{List}\} \Rightarrow \{\rho_h \mapsto \mathit{List}\}) \otimes \{\rho'_h \mapsto \mathit{List}\}$$

3.2 FIFO Queue

An implementation of a FIFO queue is shown in Figures 11 and 12. Both the functions $enqueue$ and $dequeue$ are constant-time functions. In these functions, we use the following type abbreviations:

$$\begin{array}{lcl} \mathit{List} & \equiv & \exists[\rho|\{\rho \mapsto \mathit{List}\}].\, \langle int, ptr(\rho)\rangle \\ \tau_c[\epsilon] & \equiv & \forall[\rho_q, \rho_h|(\{\rho_q \mapsto \mathit{List}\} \Rightarrow \{\rho_h \mapsto \mathit{List}\})\otimes \\ & & \quad \{\rho_q \mapsto \langle \mathcal{S}(0), ptr(\rho_h)\rangle\} \otimes \epsilon]. \\ & & \quad\quad (r : int, q : ptr(\rho_q)) \rightarrow \mathbf{0} \end{array}$$

The type *List* denotes singly-linked lists of integers, and the type τ_c denotes the type of the continuations of the functions.

The queue is represented by the store type $(\{\rho_q \mapsto List\} \Rightarrow \{\rho_h \mapsto List\}) \otimes \{\rho_q \mapsto \langle \mathcal{S}(0), ptr(\rho_h)\rangle\}$. In the store type, ρ_q denotes the tail of the queue, and it contains the pointer to ρ_h, the head of the queue. If $\rho_q = \rho_h$, the queue is empty. It is important to note that if $\rho_q = \rho_h$, the implication in the store type is equivalent to $\{\rho_q \mapsto List\} \Rightarrow \{\rho_q \mapsto List\}$, which denotes an empty store.

The $enqueue$ function is shown in Figure 11. It takes three arguments (line 4). The argument q is a queue to be manipulated, and the argument x is an integer value to be enqueued. The argument $cont$ is a continuation of the function. The function first binds the variable h to the head of the queue (line 5). Next, it allocates a new tuple, and it binds the variable n to the pointer to the tuple (line 6). It also initializes the second field of the tuple with the pointer to the head of the queue (line 8). Then, it concatenates the allocated tuple to the tail of the queue (lines 9 and 10). Next, it reserves $\{\rho_n \mapsto \mathit{List}\}$ for $\{\rho_q \mapsto \langle int, \rho_n\rangle\}$ (line 11). Then, the entire store type is updated as follows:

$$(\{\rho_q \mapsto \mathit{List}\} \Rightarrow \{\rho_h \mapsto \mathit{List}\})\otimes \\ (\{\rho_n \mapsto \mathit{List}\} \Rightarrow \{\rho_q \mapsto \langle int, ptr(\rho_n)\rangle\} \otimes \{\rho_n \mapsto \mathit{List}\})\otimes \\ \{\rho_n \mapsto \langle \mathcal{S}(0), ptr(\rho_h)\rangle\} \otimes \epsilon$$

Next, it packs the tuple at the location ρ_q (line 12), and the store type is translated as follows:

$$(\{\rho_q \mapsto \mathit{List}\} \Rightarrow \{\rho_h \mapsto \mathit{List}\})\otimes \\ (\{\rho_n \mapsto \mathit{List}\} \Rightarrow \{\rho_q \mapsto \mathit{List}\})\otimes \\ \{\rho_n \mapsto \langle \mathcal{S}(0), ptr(\rho_h)\rangle\} \otimes \epsilon$$

Then, it concatenates the two separating implications via `trans` (lines 13 and 14) as follows:

$$(\{\rho_n \mapsto \mathit{List}\} \Rightarrow \{\rho_h \mapsto \mathit{List}\})\otimes \\ \{\rho_n \mapsto \langle \mathcal{S}(0), ptr(\rho_h)\rangle\} \otimes \epsilon$$

Finally, the function returns the updated queue (line 14) by instantiating ρ_q with ρ_n.

The $dequeue$ function is shown in Figure 12. It takes two arguments (line 4). The argument q is a queue to be manipulated, and the argument $cont$ is a continuation of the function. The function first binds the variable h to the head of the queue (line 5). Next, it checks whether the queue is empty (line 6). If the queue is empty, it returns the integer value -1 (line 7). Otherwise, it unpacks the first element of the queue (line 9). Then, the entire store type is translated as follows:

$$(\{\rho_q \mapsto \mathit{List}\} \Rightarrow \{\rho_h \mapsto \langle int, ptr(\rho_n)\rangle\} \otimes \{\rho_n \mapsto \mathit{List}\})\otimes \\ \{\rho_q \mapsto \langle \mathcal{S}(0), ptr(\rho_h)\rangle\} \otimes \epsilon$$

It is important to note that `unpacking` is possible because of the last rule of Figure 8, described in Section 2.2.1. Next, it binds the variable r to the stored integer (line 10) and the variable n to the pointer to the next element (line 11). Then, it unlinks and deallocates the first element (lines 12 and 13), and the entire store type is translated as follows:

$$(\{\rho_q \mapsto \mathit{List}\} \Rightarrow \{\rho_n \mapsto \mathit{List}\})\otimes \\ \{\rho_q \mapsto \langle \mathcal{S}(0), ptr(\rho_n)\rangle\} \otimes \epsilon$$

Finally, the function returns the stored integer and the updated queue (line 14) by instantiating ρ_h with ρ_n.

```
 1 : fix enqueue[ρh, ρq, ϵ|
 2 :   ({ρq ↦ List} ⇒ {ρh ↦ List})⊗
 3 :   {ρq ↦ ⟨S(0), ptr(ρh)⟩} ⊗ ϵ]
 4 :   (q : ptr(ρq), x : int, cont : τc[ϵ]).
 5 :   let h = (q).2;
 6 :   new ρn, n, 2;
 7 :   (n).1 := 0;
 8 :   (n).2 := h;
 9 :   (q).1 := x;
10 :   (q).2 := n;
11 :   reserve {ρn ↦ List} for {ρq ↦ ⟨int, ρn⟩};
12 :   pack[ρn|{ρn↦List}]asList(ρq);
13 :   trans {ρn ↦ List} ⇒ {ρq ↦ List} with
14 :         {ρq ↦ List} ⇒ {ρh ↦ List};
15 :   cont[ρn, ρh](0, n)
```

Figure 11. Enqueue

3.3 Deletion from a Binary Search Tree

An implementation of deletion from a binary search tree is shown in Figures 13, 14, 15, 16, and 17. The main function ($delete\_bst$)

```
 1 : fix dequeue[ρ_h, ρ_q, ϵ|
 2 :   ({ρ_q ↦ List} ⇒ {ρ_h ↦ List})⊗
 3 :   {ρ_q ↦ ⟨S(0), ptr(ρ_h)⟩} ⊗ ϵ]
 4 :   (q : ptr(ρ_q), cont : τ_c[ϵ]).
 5 :   let h = (q).2;
 6 :   ifpeq q = h then
 7 :     cont[ρ_q, ρ_h](−1, q);
 8 :   else
 9 :     unpack ρ_h with ρ_n;
10 :     let r = (h).1;
11 :     let n = (h).2;
12 :     (q).2 := n;
13 :     free h;
14 :     cont[ρ_q, ρ_n](r, q)
```

Figure 12. Dequeue

is shown in Figure 13. The auxiliary functions (*delete_aux*, *delete_aux_left*, and *delete_aux_right*) used in *delete_bst* are shown in Figures 14, 15, and 16, respectively. The *search_max* function that finds the maximum value from a binary search tree is shown in Figure 17.

The *delete_bst* function takes five arguments (shown in lines 6 and 7). The argument h is a pointer to the root node of a binary search tree, and the argument p is a pointer to a node that is to be examined in the binary search tree. The argument p' is a pointer to the parent node of the node pointed by p, and the argument i is an integer value to be deleted from the binary search tree. The argument *cont* is a continuation.

As shown in lines 33 and 34, the type of the binary search trees is defined using a variant type. The first element of the variant type denotes leaf nodes, and the second element denotes intermediate nodes. An intermediate node consists of three elements. The first element denotes an integer value stored in the node. The second and third elements denote left and right subtrees, respectively. It is important to note that, although variant types are not handled in the proposed language presented in Section 2, it should be straightforward to extend it with variant types, in the same way as in [18].

The store type of lines 2 to 5 specifies the precondition of the function in the same way as in Sections 3.1 and 3.2, except for the location $\rho_{p'}$. As shown in lines 3 and 4, the type of the value at the location $\rho_{p'}$ is denoted as a variant type. This is because there are two cases how the node pointed by p is reached from its parent node pointed by p': its left child and its right subtree. The variant type is utilized in order to handle the two cases uniformly by the single function *delete_bst*.

The function *delete_bst* first examines the node pointed by p whether it is a leaf node or an intermediate node (line 8). If the node is a leaf, the location $\rho_{p'}$ is packed (line 10), and the store type is translated as follows:

$$(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\}) \otimes \{\rho_{p'} \mapsto \mathit{Tree}\} \otimes \epsilon$$

Next, the coercion `indicated` is performed (lines 11 and 12), and the store type is translated as follows:

$$\{\rho_h \mapsto \mathit{Tree}\} \otimes \epsilon$$

Finally, the function returns the binary search tree pointed by h (line 13).

Otherwise, if the node is an intermediate node, the location pointed by p is unpacked (line 15). Then, the entire store type is

```
 1 : fix delete_bst[ρ_h, ρ_p', ρ_p, ρ_q, ϵ|
 2 :   ({ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree})
 3 :   ⊗{ρ_p' ↦ ⟨int, ptr(ρ_p), ptr(ρ_q)⟩
 4 :             ∪ ⟨int, ptr(ρ_q), ptr(ρ_p)⟩}
 5 :   ⊗{ρ_p ↦ Tree} ⊗ {ρ_q ↦ Tree} ⊗ ϵ]
 6 :   (h : ptr(ρ_h), p : ptr(ρ_p), p' : ptr(ρ_p'),
 7 :     i : int, cont : τ_c[ρ_h, ϵ]).
 8 : case p of
 9 : ( inl ⟶
10 :   pack_[ρ_p,ρ_q|{ρ_p↦Tree}⊗{ρ_q↦Tree}] as Tree (ρ_p');
11 :   indicated {ρ_p' ↦ Tree} for
12 :     {ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree};
13 :   cont(h)
14 : | inr ⟶
15 :   unpack ρ_p with ρ_pl, ρ_pr;
16 :   let x = (p).1;
17 :   if i = x then
18 :     delete_aux[ρ_h, ρ_p', ρ_p, ρ_q, ρ_pl, ρ_pr, ϵ](h, p, p', cont)
19 :   else
20 :     reserve {ρ_p ↦ Tree} for {ρ_q ↦ Tree}
21 :       ⊗{ρ_p' ↦ ⟨int, ptr(ρ_p), ptr(ρ_q)⟩
22 :           ∪ ⟨int, ptr(ρ_q), ptr(ρ_p)⟩};
23 :     pack_[ρ_p,ρ_q|{ρ_p↦Tree}⊗{ρ_q↦Tree}] as Tree (ρ_p');
24 :     trans {ρ_p ↦ Tree} ⇒ {ρ_p' ↦ Tree} with
25 :       {ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree};
26 :     if i < x then
27 :       let pl = (p).2;
28 :       delete_bst[ρ_h, ρ_p, ρ_pl, ρ_pr, ϵ](h, pl, p, i, cont)
29 :     else
30 :       let pr = (p).3;
31 :       delete_bst[ρ_h, ρ_p, ρ_pr, ρ_pl, ϵ](h, pr, p, i, cont)
32 : )
where
33 : Tree ≡ ⟨⟩ ∪ ∃[ρ_pl, ρ_pr|{ρ_pl ↦ Tree} ⊗ {ρ_pr ↦ Tree}].
34 :             ⟨int, ptr(ρ_pl), ptr(ρ_pr)⟩
35 : τ_c[ρ, ϵ] ≡ ∀[|{ρ ↦ Tree} ⊗ ϵ].(a : ptr(ρ)) → 0
```

Figure 13. Deletion from a Binary Search Tree

translated as follows:

$$\begin{aligned}&(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ &\otimes\{\rho_{p'} \mapsto \langle int, ptr(\rho_p), ptr(\rho_q)\rangle \cup \langle int, ptr(\rho_q), ptr(\rho_p)\rangle\}\\ &\otimes\{\rho_p \mapsto \langle int, ptr(\rho_{pl}), ptr(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ &\otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \{\rho_q \mapsto \mathit{Tree}\} \otimes \epsilon\end{aligned}$$

Next, the integer value stored in the node pointed by p is examined (lines 16 and 17). If the value is equal to the value to be deleted (i), the auxiliary function *delete_aux* (shown in Figure 14) is called (line 18).

Otherwise, if the value is not equal to the value to be deleted (line 19), the coercion `reserve` first translates the store type as follows (lines 20 to 22):

$$\begin{aligned}&(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ &\otimes(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_p \mapsto \mathit{Tree}\} \otimes \{\rho_q \mapsto \mathit{Tree}\}\\ &\quad\otimes\{\rho_{p'} \mapsto \langle int, ptr(\rho_p), ptr(\rho_q)\rangle \cup \langle int, ptr(\rho_q), ptr(\rho_p)\rangle\})\\ &\otimes\{\rho_p \mapsto \langle int, ptr(\rho_{pl}), ptr(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ &\otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{aligned}$$

Next, the location $\rho_{p'}$ is packed (line 23), and the entire store type is translated as follows:

$$\begin{aligned}&(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ &\otimes(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_{p'} \mapsto \mathit{Tree}\})\\ &\otimes\{\rho_p \mapsto \langle int, ptr(\rho_{pl}), ptr(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ &\otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{aligned}$$

```
 1 : fix delete_aux[ρ_h, ρ_p', ρ_p, ρ_q, ρ_pl, ρ_pr, ϵ|
 2 :   ({ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree})
 3 :   ⊗{ρ_p' ↦ ⟨int, ptr(ρ_p), ptr(ρ_q)⟩
 4 :           ∪ ⟨int, ptr(ρ_q), ptr(ρ_p)⟩}
 5 :   ⊗{ρ_p ↦ ⟨int, ptr(ρ_pl), ptr(ρ_pr)⟩} ⊗ {ρ_pl ↦ Tree}
 6 :   ⊗{ρ_pr ↦ Tree} ⊗ {ρ_q ↦ Tree} ⊗ ϵ]
 7 :   (h : ptr(ρ_h), p : ptr(ρ_p), p' : ptr(ρ_p'),
 8 :     cont : τ_c[ρ_h, ϵ]).
 9 : case p' of
10 : ( inl ⟶
11 :     delete_aux_left[ρ_h, ρ_p', ρ_p, ρ_q, ρ_pl, ρ_pr, ϵ]
12 :       (h, p, p', cont)
13 : | inr ⟶
14 :     delete_aux_right[ρ_h, ρ_p', ρ_p, ρ_q, ρ_pl, ρ_pr, ϵ]
15 :       (h, p, p', cont)
16 : )
```

Figure 14. Auxiliary Function for Deletion from a Binary Search Tree (1 of 3)

Then, the two separating implications are concatenated via `trans` (lines 24 and 25) as follows:

$$\begin{array}{l}(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes\{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ \otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

Finally, the function determines which subtree should be examined next (line 26), and calls itself tail-recursively (lines 28 and 31).

The *delete_aux* function (shown in Figure 14) does almost nothing but call one of the two auxiliary functions (*delete_aux_left* or *delete_aux_right*), depending on how the node pointed by p is reached from its parent node (line 9). If it is in the left subtree of the parent node, *delete_aux_left* is called (lines 11 and 12). Otherwise, *delete_aux_right* is called (lines 14 and 15).

The *delete_aux_left* function (shown in Figure 15) first examines whether the left subtree of the node pointed by p is a leaf or not (lines 8 and 10). If it is a leaf, the function stores the pointer to the right subtree in the parent node pointed by p' (line 12), and frees the node pointed by p and the leaf (lines 13 and 14). Next, the location $\rho_{p'}$ is packed (line 15), and the store type is translated as follows:

$$(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\}) \otimes \{\rho_{p'} \mapsto \mathit{Tree}\} \otimes \epsilon$$

Then, the coercion `indicated` is performed (lines 16 and 17), and the entire store type is translated as follows:

$$\{\rho_h \mapsto \mathit{Tree}\} \otimes \epsilon$$

Finally, the function returns the binary search tree pointed by h (line 18). It is important to note that the continuation *cont* is the one passed to *delete_bst*, that is, it directly returns to the caller of *delete_bst*.

If the left subtree is an intermediate node, the function examines whether the right subtree is a leaf or not (lines 9 and 20). If the right subtree is a leaf, the function does the same as lines 12 to 18, except that the function stores the pointer to the left subtree in the parent node (line 22).

Otherwise, if both of the subtrees are intermediate nodes, the coercion `reserve` (line 30 and 31) first translates the entire store type as follows:

$$\begin{array}{l}(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_p \mapsto \mathit{Tree}\} \otimes \{\rho_q \mapsto \mathit{Tree}\}\\ \quad \otimes\{\rho_{p'} \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_p), \mathit{ptr}(\rho_q)\rangle\})\\ \otimes\{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ \otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

```
 1 : fix delete_aux_left[ρ_h, ρ_p', ρ_p, ρ_q, ρ_pl, ρ_pr, ϵ|
 2 :   ({ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree})
 3 :   ⊗{ρ_p' ↦ ⟨int, ptr(ρ_p), ptr(ρ_q)⟩}
 4 :   ⊗{ρ_p ↦ ⟨int, ptr(ρ_pl), ptr(ρ_pr)⟩} ⊗ {ρ_pl ↦ Tree}
 5 :   ⊗{ρ_pr ↦ Tree} ⊗ {ρ_q ↦ Tree} ⊗ ϵ]
 6 :   (h : ptr(ρ_h), p : ptr(ρ_p), p' : ptr(ρ_p'),
 7 :     cont : τ_c[ρ_h, ϵ]).
 8 : let pl = (p).2;
 9 : let pr = (p).3;
10 : case pl of
11 : ( inl ⟶
12 :   (p').2 := pr;
13 :   free p;
14 :   free pl;
15 :   pack_[ρ_pr,ρ_q|{ρ_pr↦Tree}⊗{ρ_q↦Tree}] as Tree (p');
16 :   indicated {ρ_p' ↦ Tree} for
17 :     {ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree};
18 :   cont(h)
19 : | inr ⟶
20 :   case pr of
21 :   ( inl ⟶
22 :     (p').2 := pl;
23 :     free p;
24 :     free pr;
25 :     pack_[ρ_pl,ρ_q|{ρ_pl↦Tree}⊗{ρ_q↦Tree}] as Tree (p');
26 :     indicated {ρ_p' ↦ Tree} for
27 :       {ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree};
28 :     cont(h)
29 :   | inr ⟶
30 :     reserve {ρ_p ↦ Tree} for {ρ_q ↦ Tree}
31 :       ⊗{ρ_p' ↦ ⟨int, ptr(ρ_p), ptr(ρ_q)⟩};
32 :     pack_[ρ_p,ρ_q|{ρ_p↦Tree}⊗{ρ_q↦Tree}] as Tree (ρ_p');
33 :     trans {ρ_p ↦ Tree} ⇒ {ρ_p' ↦ Tree} with
34 :         {ρ_p' ↦ Tree} ⇒ {ρ_h ↦ Tree};
35 :     reserve {ρ_pl ↦ Tree} for ·;
36 :     search_max[ρ_pl, ρ_pl, {ρ_pr ↦ Tree} ⊗ ϵ
37 :         ⊗({ρ_p ↦ Tree} ⇒ {ρ_h ↦ Tree})
38 :         ⊗{ρ_p ↦ ⟨int, ptr(ρ_pl), ptr(ρ_pr)⟩}]
39 :       (pl, min_int,
40 :         fix cont'[|{ρ_pr ↦ Tree} ⊗ ϵ
41 :           ({ρ_p ↦ Tree} ⇒ {ρ_h ↦ Tree})
42 :           ⊗{ρ_p ↦ Tree} ⊗ {ρ_pl ↦ Tree}](m : int).
43 :         (p).1 := m;
44 :         delete_bst[ρ_h, ρ_p, ρ_pl, ρ_pr, ϵ](h, pl, p, m, cont)
45 :       )
46 :   )
47 : )
```

Figure 15. Auxiliary Function for Deletion from a Binary Search Tree (2 of 3)

Next, the location $\rho_{p'}$ is packed (line 32) as follows:

$$\begin{array}{l}(\{\rho_{p'} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_{p'} \mapsto \mathit{Tree}\})\\ \otimes\{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ \otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

Then, the coercion `trans` (lines 33 and 34) translates the entire store type as follows:

$$\begin{array}{l}(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes\{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ \otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

Next, the coercion `reserve` (line 35) introduces a seemingly useless separating implication that is necessary to call $search\_max$ (shown in Figure 17) as follows:

$$\begin{array}{l}(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes\{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\} \otimes \{\rho_{pl} \mapsto \mathit{Tree}\}\\ \otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\\ \otimes(\{\rho_{pl} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_{pl} \mapsto \mathit{Tree}\})\end{array}$$

Then, it calls $search\_max$ in order to get the maximum value in the left subtree (lines 36 to 45), and the value is stored in the node pointed by p (line 43). Finally, it calls $delete\_bst$ tail-recursively in order to delete the maximum value from the left subtree (line 44).

The $delete\_aux\_right$ function (shown in Figure 16) is almost the same as $delete\_aux\_left$ except for that it is used when the node pointed by p is in the right subtree of the parent node pointed by p'. More specifically, if one of the subtrees of the node pointed by p is a leaf, the pointer to the other subtree is stored to the third element of the parent node pointed by p' (lines 12 and 22), instead of the second element.

$search\_max$ is a tail-recursive function that returns the maximum value stored in a binary search tree (shown in Figure 17). It first examines the tree pointed by p whether the tree is a leaf or not (line 5). If the tree is a leaf, the coercion `indicated` translates the store type as follows (lines 7 and 8):

$$\{\rho_h \mapsto \mathit{Tree}\} \otimes \epsilon$$

Finally, the function returns the integer value m (line 9). It is important to note here that the second argument of the function holds the maximum value in the tree pointed by h, excluding the subtree pointed by p.

Otherwise, the location ρ_p is unpacked (line 11), and the entire store type is translated as follows:

$$\begin{array}{l}(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes\{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\}\\ \otimes\{\rho_{pl} \mapsto \mathit{Tree}\} \otimes \{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

Next, the coercion `reserve` translates the store type as follows (lines 14 and 15):

$$\begin{array}{l}(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes(\{\rho_{pr} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_p \mapsto \langle \mathit{int}, \mathit{ptr}(\rho_{pl}), \mathit{ptr}(\rho_{pr})\rangle\}\\ \quad\otimes\{\rho_{pl} \mapsto \mathit{Tree}\} \otimes \{\rho_{pr} \mapsto \mathit{Tree}\})\\ \otimes\{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

Then, the location ρ_p is packed as follows (line 16):

$$\begin{array}{l}(\{\rho_p \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\})\\ \otimes(\{\rho_{pr} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_p \mapsto \mathit{Tree}\}) \otimes \{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon\end{array}$$

Now, the `trans` operation (lines 17 and 18) is performed, and the entire store type is finally translated as follows:

$$(\{\rho_{pr} \mapsto \mathit{Tree}\} \Rightarrow \{\rho_h \mapsto \mathit{Tree}\}) \otimes \{\rho_{pr} \mapsto \mathit{Tree}\} \otimes \epsilon$$

Finally, the function calls itself tail-recursively (line 19) by instantiating ρ_p with ρ_{pr}.

```
 1 : fix delete_aux_right[ρh, ρp', ρp, ρq, ρpl, ρpr, ϵ|
 2 :   ({ρp' ↦ Tree} ⇒ {ρh ↦ Tree})
 3 :   ⊗{ρp' ↦ ⟨int, ptr(ρq), ptr(ρp)⟩}
 4 :   ⊗{ρp ↦ ⟨int, ptr(ρpl), ptr(ρpr)⟩} ⊗ {ρpl ↦ Tree}
 5 :   ⊗{ρpr ↦ Tree} ⊗ {ρq ↦ Tree} ⊗ ϵ]
 6 :   (h : ptr(ρh), p : ptr(ρp), p' : ptr(ρp'),
 7 :     cont : τc[ρh, ϵ]).
 8 : let pl = (p).2;
 9 : let pr = (p).3;
10 : case pl of
11 : ( inl ⟶
12 :     (p').3 := pr;
13 :     free p;
14 :     free pl;
15 :     pack[ρpr,ρq|{ρpr↦Tree}⊗{ρq↦Tree}]asTree(p');
16 :     indicated {ρp' ↦ Tree} for
17 :       {ρp' ↦ Tree} ⇒ {ρh ↦ Tree};
18 :     cont(h)
19 : | inr ⟶
20 :     case pr of
21 :     ( inl ⟶
22 :         (p').3 := pl;
23 :         free p;
24 :         free pr;
25 :         pack[ρpl,ρq|{ρpl↦Tree}⊗{ρq↦Tree}]asTree(p');
26 :         indicated {ρp' ↦ Tree} for
27 :           {ρp' ↦ Tree} ⇒ {ρh ↦ Tree};
28 :         cont(h)
29 :     | inr ⟶
30 :         reserve {ρp ↦ Tree} for {ρq ↦ Tree}
31 :           ⊗{ρp' ↦ ⟨int, ptr(ρp), ptr(ρq)⟩};
32 :         pack[ρp,ρq|{ρp↦Tree}⊗{ρq↦Tree}]asTree(ρp');
33 :         trans {ρp ↦ Tree} ⇒ {ρp' ↦ Tree} with
34 :             {ρp' ↦ Tree} ⇒ {ρh ↦ Tree};
35 :         reserve {ρpl ↦ Tree} for ·;
36 :         search_max[ρpl, ρpl, {ρpr ↦ Tree} ⊗ ϵ
37 :             ⊗({ρp ↦ Tree} ⇒ {ρh ↦ Tree})
38 :             ⊗{ρp ↦ ⟨int, ptr(ρpl), ptr(ρpr)⟩}]
39 :         (pl, min_int,
40 :           fix cont'[|{ρpr ↦ Tree} ⊗ ϵ
41 :             ({ρp ↦ Tree} ⇒ {ρh ↦ Tree})
42 :             ⊗{ρp ↦ Tree} ⊗ {ρpl ↦ Tree}](m : int).
43 :           (p).1 := m;
44 :           delete_bst[ρh, ρp, ρpl, ρpr, ϵ](h, pl, p, m, cont)
45 :         )
46 :     )
47 : )
```

Figure 16. Auxiliary Function for Deletion from a Binary Search Tree (3 of 3)

```
 1 :  fix search_max[ρ_h, ρ_p, ϵ|
 2 :    ({ρ_p ↦ Tree} ⇒ {ρ_h ↦ Tree})
 3 :    ⊗{ρ_p ↦ Tree} ⊗ ϵ]
 4 :    (p : ptr(ρ_p), m : int, cont : τ'_c[ρ_h, ϵ]).
 5 :  case p of
 6 :  ( inl ⟶
 7 :    indicated {ρ_p ↦ Tree} for
 8 :      {ρ_p ↦ Tree} ⇒ {ρ_h ↦ Tree};
 9 :    cont(m)
10 :  | inr ⟶
11 :    unpack ρ_p with ρ_pl, ρ_pr;
12 :    let x = (p).1;
13 :    let pr = (p).3;
14 :    reserve {ρ_pr ↦ Tree} for {ρ_pl ↦ Tree}
15 :      ⊗{ρ_p ↦ ⟨int, ptr(ρ_pl), ptr(ρ_pr)⟩};
16 :    pack_[ρ_pl,ρ_pr|{ρ_pl↦Tree}⊗{ρ_pr↦Tree}]asTree(ρ_p);
17 :    trans {ρ_pr ↦ Tree} ⇒ {ρ_p ↦ Tree} with
18 :      {ρ_p ↦ Tree} ⇒ {ρ_h ↦ Tree};
19 :    search_max[ρ_h, ρ_pr, ϵ](pr, x, cont)
20 :  )
where
21 :  τ'_c[ρ, ϵ] ≡ ∀[|{ρ ↦ Tree} ⊗ ϵ].(m : int) → 0
```

Figure 17. Search Maximum Value from a Binary Search Tree

It is important to note that, in a strict sense, the implementation of deletion from a binary search tree shown in this section is not fully tail-recursive because it creates a continuation as an argument for $search\_max$ in $delete\_aux\_left$ and $delete\_aux\_right$. However, the implementation only takes constant space regardless of the size of a binary search tree because $search\_max$ is tail-recursive and does not call any other function except for the continuation.

4. Related Work

Linear types [16, 17] are type systems based on linear logic. They statically detect values that are used exactly once, and they deallocate their memory regions immediately after they are used. Linear types have two drawbacks from the view point of explicit memory management. First, many common data structures using pointer aliasing cannot be expressed using linear type systems because they effectively prohibit pointer aliasing of linear values, unlike alias type systems. Quasi-linear types [6] relax this limitation by taking the order of evaluation into consideration. Second, although they achieve explicit memory deallocation, explicit memory reuse is not allowed because they do not support the strong updating of memory regions, unlike alias type systems. One advantage of linear type systems is their ability to infer types. On the other hand, alias type systems currently require type annotations by programmers.

Region-based memory management [5, 14, 15] is another approach to memory management, which does not rely on garbage collection. Region-based memory management involves the division of memory allocations into groups called regions and the static management of pointers among them. By adopting the region-based memory management technique described in [5], programmers can reclaim regions explicitly; however, such a technique does not provide type inferencing mechanisms, as in the case of alias type systems. Region-base memory management has two drawbacks. First, the memory management unit is coarse, as compared to that of linear and alias type systems. Therefore, the ability to explicitly manage memory is limited. Second, it does not support explicit memory reuse, as in the case of linear type systems.

Shape analysis [11, 12] involves the extraction of shape invariants, that is, approximated structures built with memory regions and pointers among them. Although shape analysis can be used to explicitly deallocate memory regions [4], it is sometimes inadequate for explicit memory management because it does not determine whether memory regions can be explicitly reused. In addition, effective shape analysis approaches [3, 7] suffer from limitations in handling generic recursive data structures. This is because the detection of aliasing relations in programs is undecidable [9]. On the other hand, the proposed type system handles recursive data structures efficiently and precisely at the cost of coercions and type annotations by programmers. The workload of the programmers can be reduced by incorporating shape analysis.

Separation logic [10] is an extension of Hoare logic, which permits reasoning for low-level imperative programs that use shared mutable data structures. It is a substructural logic that can describe heap-allocated memory regions and pointers among them. It is known that the concept of the separating conjunction in separation logic is closely related to the original alias type system [10, 18]. The present study is the first elucidation of the application of separating implications to the alias type system; moreover, we have demonstrated its effectiveness. Separation logic is more expressive than the proposed type system; however, it requires manual proofs that are relatively complex, as opposed to the insertion of coercions in the proposed type system. Although decision procedures for restricted separation logic have been investigated [1, 2, 8], they do not handle separating implications.

5. Conclusion

In this paper, we proposed an extension of the alias type system [18] using separating implications, which are derived from separation logic [10]. The original alias type system allows programmers to write explicit memory management code by tracking aliasing relations in memory regions. However, it suffers from limitations in expressing tail-recursive operations on recursive data structures because it requires complete information about the aliasing relations. In the proposed type system, the separating implications relax the limitations by allowing implicit pointer aliases between the antecedents and the consequents of the implications. Further, we presented examples to demonstrate the expressiveness of the proposed type system. The proposed type system enables us to describe a tail-recursive and destructive list append function, a FIFO queue with constant-time operations, and an implementation of deletion from a binary search tree that only takes constant space.

Acknowledgments

This work has been partially supported by CREST of JST (Japan Science and Technology Agency).

References

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *Proc. of FSTTCS 2004*, pages 97–109, 2004.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proc. of APLAS 2005*, pages 52–68, 2005.

[3] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Proc. of CAV 2007*, pages 178–192, 2007.

[4] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *Proc. of ISMM 2006*, pages 138–149, 2006.

[5] K. Crary, D. Walker, and J. G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. of POPL 1999*, pages 262–275, 1999.

[6] N. Kobayashi. Quasi-linear types. In *Proc. of POPL 1999*, pages 29–42, 1999.

[7] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *Proc. of ESOP 2005*, pages 124–140, 2005.

[8] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *Proc. of VMCAI 2007*, pages 251–266, 2007.

[9] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[10] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. of LICS 2002*, pages 55–74, 2002.

[11] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3–valued logic. In *Proc. of POPL 1999*, pages 105–118, 1999.

[13] F. Smith, D. Walker, and G. Morrisett. Alias types. *Lecture Notes in Computer Science*, 1782:366–381, 2000.

[14] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proc. of POPL 1994*, pages 188–201, 1994.

[15] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132:109–176, 1997.

[16] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA 1995*, pages 1–11, 1995.

[17] P. Wadler. Linear types can change the world! In *Proc. of PROCOMET 1990*, pages 347–359, 1990.

[18] D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177–206, 2001.