Reasoning about Data-Parallel Pointer Programs in a Modal Extension of Separation Logic[*]

Susumu Nishimura
Department of Mathematics, Faculty of Science, Kyoto University
Sakyo-ku, Kyoto 606-8502, Japan
E-mail: `susumu@math.kyoto-u.ac.jp`

April 17, 2006

Abstract

This paper proposes a modal extension of Separation Logic [8, 11] for reasoning about data-parallel programs that manipulate heap allocated linked data structures. Separation Logic provides a formal means for expressing allocation of disjoint substructures, which are to be processed in parallel. A modal operator is also introduced to relate the global property of a parallel operation with the local property of each sequential execution running in parallel. The effectiveness of the logic is demonstrated through a formal reasoning on the parallel list scan algorithm featuring the pointer jumping technique.

1 Introduction

Parallel prefix or *scan* on arrays is a fundamental collective operation in parallel computing [7, 2]. For example, the parallel prefix sum algorithm computes the sums of prefix subsequences of an integer array. The prefix computation can be efficiently implemented on parallel computers, where the basic technique is simultaneous addition of array elements at indices of exponentially increasing intervals. The same technique applies to implement a range of parallel algorithms including parallel sorting, maximum segment sum, etc. A *data-parallel* programming paradigm is best suited for the implementation, where the same sequential program processes every different array element simultaneously, as attributed by the SPMD (single program, multiple data-stream) execution scheme [6].

A similar but more sophisticated programming technique can even implement parallel collective operations on linked data structures, *e.g.*, lists and trees. The program in Figure 1 implements a data-parallel scan operation on integer list that computes the sum

[*]This preprint is an extended version of the article which will appear in the Proc. of 11th International Conference on Algebraic Methodology and Software Technology (AMAST '06) under the same title. A preliminary short summary was presented at the 3rd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2006), Charleston, SC, January 2006.

```
(* $p$ is a pointer to the initial cell of the list *)
$q := [p+1]$;
**while** $q \neq \mathsf{nil}$ **do**
  **begin**
    **forall** $x \mapsto n, t$ **in** $\langle$*allocation addresses of list cells*$\rangle$ **do**
      **begin var** $m$;
        **if** $t \neq \mathsf{nil}$ **then** $m := [t]$; $t := [t+1]$; $[x] := n+m$; $[x+1] := t$
      **end**;
    $q := [p+1]$
  **end**
```

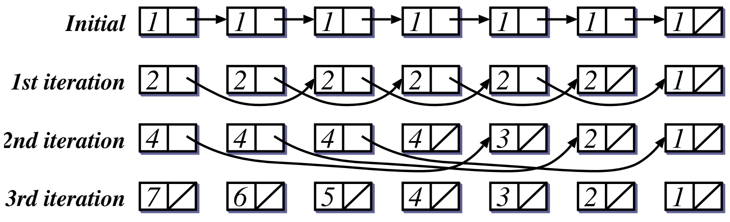

Figure 1: Data-parallel list scan algorithm

of every sublist, where the integer list is expressed by a linked structure. Each cons cell allocates an integer value in the head position and a pointer to the successor cell (or a special value nil when there is no successor cell) in the tail position.

The programming technique employed in the program is called *pointer jumping* [7, 5]. As depicted in Figure 1, in each step of parallel execution, the tail position in every cons cell is updated with the value contained in the tail position of the successor cell and also the integer value in the head position is added with that in the successor cell. Every single step of parallel execution is expressed in the program by a data-parallel primitive, the **forall** command. In the program, the **forall** command executes the command body for every different cons cell in parallel, with x bound to the allocation address of the cons cell and n and t bound to values stored in the head part and the tail part of the cell, respectively. In the command body, $[p]$ stands for dereferencing of a pointer p, $[p+1]$ for dereferencing with displacement 1, and the command $[e] := e'$ updates the address e with the value of e'. The iteration is terminated as soon as the tail position of the initial cell has been set to nil. It is easy to see that the iteration requires only a logarithmic number of steps, since the length of pointers is doubled per each iteration.

This paper proposes a program logic, in the style of Hoare, for verifying such data-parallel programs implementing a collective operation on linked data structure, on top of a formal semantics of a suitable data-parallel programming language.

So far several formal semantics have been proposed for data-parallel programming languages (*e.g.*, [4, 3, 9]). The most prospective for our purpose would be the assertional approach by Bougé et al. [4]. However, they only deal with arrays as the primary data structure for parallel processing. Thus pointer jumping is expressed with indirection of interpreting pointers as array indices. This is not merely a notational issue. More significantly, it obfuscates the logical structure of the program, by having the formal reasoning stick to particular properties of integer arithmetics.

Our formal proof system for verification deals with a data-parallel programming language in which pointers are first-class citizens. Pointer operations are notoriously hard to reason about, even in sequential programs. We solve the difficulty by adopting Separation Logic [8, 11], which has been recently developed for compositional reasoning on pointer manipulating programs.

As we shall discuss later, parallel list scan instantiates the divide-and-conquer strategy, where a data set is decomposed into disjoint subcomponents that are subject to successive parallel processing. The parallel processing of subcomponents is guaranteed safe, as disjointness implies non-interference among parallel threads of sequential execution. Separation Logic allows us to express this property of the program in a notably elegant way.

In addition to the standard features of Separation Logic, we extend it with a modal operator, expressing modal possibility up to alteration of the heap contents. (Although our modal operator is much alike Berger, Honda, and Yoshida's content quantification [1], ours has a fundamental difference from theirs: theirs takes care of a single address, while ours mentions about the entire set of allocation addresses of the current heap.) The modal operator provides a logical means for relating the global property of the execution of a **forall** command with the property local to every sequential execution running in parallel.

The rest of the paper is organized as follows. Section 2 introduces Separation Logic with a modal extension. Section 3 informally explains the logical structure of parallel list scan algorithm and interprets it into a formal specification in Separation Logic. In Section 4, we give a formal definition of the data-parallel programming language and a sound proof system for it. Section 5 gives a formal correctness proof for the parallel list scan. Finally, Section 6 concludes the paper.

2 The Assertion Language of Separation Logic with Modality

This section gives the formal definition of the syntax and semantics of the assertion language of Separation Logic extended with modality. Separation logic extends first-order logic with assertions for mentioning about heap storage. We assume a shared memory model for the heap storage, where a single global memory space is shared among all parallel execution instances.

Given a partial function f, let us write $f[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ for a partial function g such that $dom(g) = dom(f) \cup \{x_1, \ldots, x_n\}$, $g(x_i) = v_i$ for every i, and $g(y) = f(y)$ for every $y \in dom(f) \setminus \{x_1, \ldots, x_n\}$. We write $f \sharp g$ to mean that f and g have disjoint domains. We also write $f * g$ for the disjoint union of the two functions, if $f \sharp g$; $f * g$ is undefined otherwise. Sometimes we represent a partial function f by its *graph* $\{(x, f(x)) \mid x \in dom(f)\}$. The notation $f \upharpoonright A$ expresses a restriction of f to a set A, *i.e.*, $f \upharpoonright A = \{(x, f(x)) \mid x \in dom(f) \cap A\}$.

Let *Var* be a set of of variables. We define the set *Val* of values as the set of integers and the set *Addr* of addresses as the set of non-negative integers, hence $\mathit{Addr} \subseteq \mathit{Val}$.

Our formal model of storage consists of two semantic domains, *store* and *heap*:

$$\mathit{Store} = \mathit{Var} \to \mathit{Val} \qquad \mathit{Heap} = \mathit{Addr} \rightharpoonup_{\mathit{fin}} \mathit{Val} \qquad \mathit{State} = \mathit{Store} \times \mathit{Heap}.$$

A store is a total mapping from variables to values. A heap is a finite partial mapping that associates each address with its content. The state of the entire storage is represented by a pair of a store and a heap.

The syntax of the assertion language is given below.

$$\begin{array}{rcll} e & ::= & \langle \mathit{integers} \rangle \mid \mathsf{nil} \mid x \mid e + e & \text{(expressions)} \\ P & ::= & \textbf{true} \mid \textbf{false} \mid e = e \mid \neg P \mid P \vee P \mid P \wedge P \mid P \Rightarrow P \mid \exists x.P \mid \forall x.P & \\ & & \mid \textbf{emp} \mid e \mapsto e \mid P * P \mid P -\!\!* P \mid \Diamond P & \text{(assertions)} \end{array}$$

An expression e is either an integer value, a constant symbol nil, a variable, or a sum of integers. We assume the symbol nil stands for a non-address value. In addition to the standard connectives of first-order logic, the assertion language provides several separating connectives and a modality. We write $FV(P)$ for the set of free variables, *i.e.*, variables whose occurrence in P is not in the scope of any enclosing quantifier.

Given $(s,h) \in \mathit{State}$, the interpretation of assertions is defined as below. Let us write $[\![e]\!]s$ to denote the value of expression e interpreted under the store s. We define the judgment $s,h \models P$ as follows.

$s,h \models \textbf{true}$	always holds.
$s,h \models \textbf{false}$	never holds.
$s,h \models \neg P$	iff $s,h \models P$ does not hold.
$s,h \models P \vee Q$	iff $s,h \models P$ or $s,h \models Q$ hold.
$s,h \models P \wedge Q$	iff $s,h \models P$ and $s,h \models Q$ hold.
$s,h \models P \Rightarrow Q$	iff $s,h \models P$ implies $s,h \models Q$.
$s,h \models \exists x.P$	iff $s[x \mapsto v],h \models P$ holds for some $v \in \mathit{Val}$.
$s,h \models \forall x.P$	iff $s[x \mapsto v],h \models P$ holds for all $v \in \mathit{Val}$.
$s,h \models \textbf{emp}$	iff $\mathit{dom}(h) = \emptyset$.
$s,h \models e \mapsto e'$	iff $[\![e]\!]s \in \mathit{Addr}$ and $h = \{([\![e]\!]s, [\![e']\!]s)\}$.
$s,h \models P * Q$	iff $h = h_1 * h_2$, $s,h_1 \models P$, and $s,h_2 \models Q$ for some h_1, h_2.
$s,h \models P -\!\!* Q$	iff $h \sharp h'$ and $s,h' \models P$ implies $s,h*h' \models Q$, for all h'.
$s,h \models \Diamond P$	iff $\mathit{dom}(h) = \mathit{dom}(h')$ and $s,h' \models P$ for some h'.

The connectives of first-order logic have the usual interpretation. The assertion **emp** indicates an empty heap that allocates no contents yet. Points-to relation $e \mapsto e'$ indicates a singleton heap, which allocates a single content e' at the address e. Separating conjunction $P * Q$ holds for the current heap h iff there exists a disjoint separation of the heap $h =$

$h_1 * h_2$ such that P holds for h_1 and Q holds for h_2. Separating implication $P \mathrel{-\!\!*} Q$ says that Q holds up to any expansion of the current heap that satisfies P. The modal operator is new to this paper. The assertion $\Diamond P$ means that the assertion P can be made true by appropriately changing currently allocated values.

Here we introduce some notational conventions. We write $e \mapsto e_1, e_2, \ldots, e_n$ for $(e \mapsto e_1) * (e+1 \mapsto e_2) * \cdots * (e+n-1 \mapsto e_n)$ $(n \geq 1)$, namely, a block of size n allocating values $e_1, \ldots, e_n$ at consecutive addresses starting from e. Inexact variant of points-to relation $e \hookrightarrow e_1, \ldots, e_n$ abbreviates $e \mapsto e_1, \ldots, e_n * \mathbf{true}$, which indicates that the current heap *at least* contains the allocation as expressed by $e \mapsto e_1, \ldots, e_n$. A symbol $-$ in the right hand side of $\mapsto$ or $\hookrightarrow$ stands for an existentially quantified variable, *e.g.*, $x \mapsto 1, -, -$ for $\exists yz. x \mapsto 1, y, z$. Throughout the paper, we assume that $*$ binds more tightly than $\wedge$; we follow the usual convention on the precedence of bindings for other connectives.

We also consider the following subclasses of assertions.

- An assertion P is called *pure* if it is independent to heap, that is, $s, h \models P$ implies $s, h' \models P$ for any h'. Syntactically, any assertion is pure if it is free from assertions and connectives that are affected by heap, *i.e.*, **emp**, $*$, $\mathrel{-\!\!*}$, $\mapsto$, and $\Diamond$.

- An assertion P is called *precise* if, for any store s and heap h, there exists at most one subheap h' such that $h' \subseteq h$ and $s, h' \models P$. Assertions which are built from logical expressions only using **emp**, $*$, $\Diamond$, $e \mapsto e'$, and $e \mapsto -$ form a conservative subclass of precise assertions.

- An assertion P is called *strictly exact* if P determines at most one heap, that is, for any store s and heaps h, h', $s, h \models P$ and $s, h' \models P$ implies $h = h'$. Any assertion that is built from logical expressions only using **emp**, $*$, and $e \mapsto e'$ is strictly exact.

  Obviously any strictly exact assertion is precise.

Some logical properties of the assertion language will be given later in Section 5.

3 Specifying Parallel List Scan

A large body of parallel algorithms can be explained as an instance of the divide-and-conquer strategy, where a problem is divided into subproblems of smaller sizes and solutions to the subproblems later combine to give the final solution. The strategy merits parallel implementation, as disjoint subproblems can be safely solved in parallel.

3.1 An informal description of the correctness of parallel scan

Let us show that the data-parallel list scan algorithm is another instance of divide-and conquer. Figure 2-a gives a graphical presentation of a single step of pointer jumping on a linked list. The figure indicates that a single step of pointer jumping corresponds to an odd-even partitioning of the list: the parallel operation splits the list into two disjoint sublists, one consisting of cells sitting at odd positions of the original list and the other

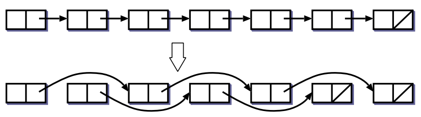
Fig. 2-a: Odd-even list partitioning

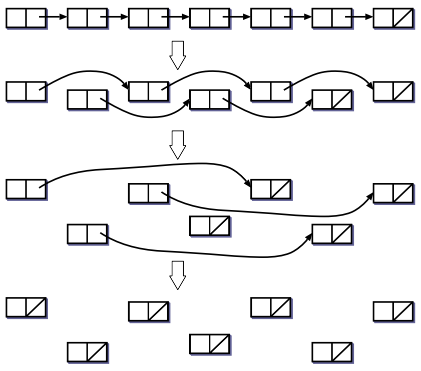
Fig. 2-b: Iterated partitioning

Figure 2: Pointer jumping as divide-and-conquer

consisting of cells sitting at even positions. Successive execution of parallel pointer jumping simultaneously operates on the partitioned sublists, further splitting each sublist into two disjoint smaller sublists. The iteration continues until the original list is decomposed into a set of singleton lists (Figure 2-b).

Note that the iteration respects the following loop invariant: "Each iteration preserves the sum of integers reachable from every cons cell." This invariant implies that, when the program terminates, every cons cell holds the sum of all the integer values reachable from that cell in the original list. Thus, we obtain the list scan.

3.2 Formal specification in Separation Logic

In the rest of this section, we express a formal specification of the properties discussed above in Separation Logic. We give a *specification* of program in Hoare's partial correctness assertion form, written $\{P\}C\{Q\}$, which means that, for any state satisfying the precondition P, the program C safely executes without errors such as memory faults and, if it ever terminates, it ends up with a final state satisfying the postcondition Q.

Let us write $[]$ to stand for an empty sequence of values and $a :: \ell$ for a sequence beginning with a value a followed by a sequence ℓ. We also abbreviate $a_1 :: (a_2 :: (\cdots :: [])\cdots)$ by $[a_1, a_2, \ldots, a_n]$. In what follows, we use meta-variables $\tau, \tau', \ldots$ to range over sequences of integers and $\sigma, \sigma', \ldots$ over sequences of addresses.

Let us define a few predicates for sequences.

- $\mathtt{part}(\sigma,\sigma_1,\sigma_2)$ iff either $\sigma = \sigma_1 = \sigma_2 = []$ or $\sigma = [p_1, p_2, \ldots, p_k]$, $\sigma_1 = [p_1, p_3, \ldots, p_{2\lfloor\frac{k-1}{2}\rfloor+1}]$, and $\sigma_2 = [p_2, p_4, \ldots, p_{2\lfloor\frac{k}{2}\rfloor}]$ $(k \geq 1)$.

- $\mathtt{alts}(\tau,\tau_1,\tau_2)$ iff either $\tau = \tau_1 = \tau_2 = []$ or $\tau = [n_1, n_2, \ldots, n_k]$, $\tau_1 = [\rho(1), \rho(3), \ldots, \rho(2\lfloor\frac{k-1}{2}\rfloor+1)]$, and $\tau_2 = [\rho(2), \rho(4), \ldots, \rho(2\lfloor\frac{k}{2}\rfloor)]$, where $\rho(k) = n_k$ and $\rho(i) = n_i + n_{i+1}$ when $0 \leq i \leq k-1$ $(k \geq 1)$.

- $\mathtt{sum}(\tau, m)$ iff $\tau = [n_1, n_2, \ldots, n_k]$ and $m = \sum_{i=1}^{k} n_i$ $(k \geq 0)$.

The predicate $\mathtt{part}(\sigma,\sigma_1,\sigma_2)$ gives the result of odd-even partitioning of the sequence σ in σ_1 and σ_2, the odd part and the even part, respectively; $\mathtt{alts}(\tau,\tau_1,\tau_2)$ in-

$$\texttt{part}(\sigma, [], \sigma_2) \Leftrightarrow \sigma = \sigma_2 = [] \tag{3.1}$$
$$\texttt{alts}(\tau, [], \tau_2) \Leftrightarrow \tau = \tau_2 = [] \tag{3.2}$$
$$\texttt{alts}([n], \tau_1, \tau_2) \Leftrightarrow \tau_1 = [n] \wedge \tau_2 = [] \tag{3.3}$$
$$\texttt{part}(n :: \sigma, m :: \sigma_1, \sigma_2) \Leftrightarrow n = m \wedge \texttt{part}(\sigma, \sigma_2, \sigma_1) \tag{3.4}$$
$$\texttt{alts}(n :: m :: \tau, n' :: \tau_1, \tau_2) \Leftrightarrow n' = n + m \wedge \texttt{alts}(m :: \tau, \tau_2, \tau_1) \tag{3.5}$$
$$\texttt{alts}(\tau, \tau_1, \tau_2) \wedge \texttt{sum}(\tau, n) \Leftrightarrow \texttt{alts}(\tau, \tau_1, \tau_2) \wedge \texttt{sum}(\tau_1, n) \tag{3.6}$$

Figure 3: Properties of sequences

dicates that τ_1 (τ_2, *resp.*) is the integer sequence obtained by adding every odd (even, *resp.*) element in the sequence τ with its successor element; $\texttt{sum}(\tau, n)$ gives the sum of integer sequence τ in n. Figure 3 lists some properties relevant to these predicates.

We define assertion $R(i, p, \sigma, \tau)$ to indicate the heap state when the i-th iteration of pointer jumping has just finished, where p is the pointer to the initial cell of the list and σ and τ are the sequences of allocation addresses and integer values of the original list, respectively. The inductive definition[1] of the assertion is given below.

$$\begin{aligned}
R(i, p, [], []) &\triangleq i \geq 0 \wedge p = \mathsf{nil} \wedge \mathbf{emp} \\
R(0, p, r :: \sigma, n :: \tau) &\triangleq p = r \wedge \exists q.(p \mapsto n, q * R(0, q, \sigma, \tau)) \\
R(i, p, r :: \sigma, n :: \tau) &\triangleq i > 0 \wedge p = r \wedge (R(i-1, p, \sigma_1, \tau_1) * \exists q. R(i-1, q, \sigma_2, \tau_2)) \\
&\qquad\qquad \text{where } \texttt{part}(r :: \sigma, \sigma_1, \sigma_2) \text{ and } \texttt{alts}(n :: \tau, \tau_1, \tau_2).
\end{aligned}$$

When $i = 0$, the assertion represents a non-circular, heap allocated linked list structure [8]. When $i > 0$, the assertion indicates that the heap allocates two disjoint sublists, the odd part $R(i-1, p, \sigma_1, \tau_1)$ and the even part $R(i-1, q, \sigma_2, \tau_2)$, where each sublist is further partitioned $i-1$ times more.

We also define assertion $\Pi^n(\sigma)$ ($n \geq 1$), whose inductive definition is given by:

$$\Pi^n([]) \triangleq \mathbf{emp} \qquad\qquad \Pi^n(p :: \sigma) \triangleq \exists x_1 \cdots x_n . p \mapsto x_1, \ldots, x_n * \Pi^n(\sigma).$$

The assertion $\Pi^n(\sigma)$ indicates that the heap allocates non-overlapping blocks of the same size n at addresses as listed in σ.

The properties about the program given in Figure 1 that we have informally discussed above can be specified as follows.

Proposition 3.1. *Let $C_{\mathbf{forall}}$ and $C_{\mathbf{while}}$ denote the command bodies of the* **forall** *and* **while** *command of the program in Figure 1, respectively. Then the following properties hold.*

(a) $\{R(i, p, \sigma, \tau)\}$ **forall** $x \mapsto n, t$ **in** σ **do** $C_{\mathbf{forall}}$ $\{R(i+1, p, \sigma, \tau)\}$

(b) $\{\exists i. R(i, p, \sigma, w) \wedge p + 1 \hookrightarrow q\}$ **while** $q \neq \mathsf{nil}$ **do** $C_{\mathbf{while}}$ $\{\exists i. R(i, p, \sigma, w) \wedge p + 1 \hookrightarrow q \wedge q = \mathsf{nil}\}$

[1]Though the present assertion language does not formally include inductive definitions, it would be extended with fixed point operators as in [12].

(c) $R(i,p,p_1 :: \sigma, n_1 :: \tau) \wedge p+1 \hookrightarrow \mathsf{nil} \Rightarrow \exists q.(R(i,q,\sigma,\tau) \wedge (q = \mathsf{nil} \vee q+1 \hookrightarrow \mathsf{nil})) *$ $(p = p_1 \wedge p \hookrightarrow n, \mathsf{nil})$, *whenever* $\mathtt{sum}(n_1 :: \tau, n)$.

The specification (a) indicates that every single execution of the **forall** command corresponds to a single step of parallel pointer jumping. The specification (b) identifies $\exists i.R(i,p,\sigma,w) \wedge p+1 \hookrightarrow q$ as the loop invariant. The implication (c) indicates that the invariant property on the sum of integers discussed earlier is intrinsic to the definition of $R(i,p,\sigma,\tau)$.

4 Program Logic for a Data-Parallel Programming Language

We consider a simple data-parallel programming language as given below:

$$\begin{aligned}
b \ &::= \mathbf{true} \mid \mathbf{false} \mid e = e \mid \neg b \mid b \vee b \mid b \wedge b \\
C \ &::= x := e \mid x := [e] \mid [e] := e' \mid \mathbf{skip} \mid C;C \mid \mathbf{begin\ var}\ x;\ C\ \mathbf{end} \\
&\quad \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C \mid \mathbf{forall}\ x \mapsto y_1,\ldots,y_n\ \mathbf{in}\ \sigma\ \mathbf{do}\ C\ (n \geq 0)
\end{aligned}$$

where meta-variable e ranges over the set of expressions and σ in the **forall** command over the set of non-empty sequences of address constants.

The language consists of the following components. Assignment command $x := e$ updates the variable x by the value of the expression e; Lookup command $x := [e]$ assigns x with the dereferenced value of the address e; Mutation $[e] := e'$ updates the content at the address e by the value of e'; The command **skip** does no operation. These atomic commands are composed via sequencing $C_1;\ C_2$, block structures **begin** $\cdots$ **end** with local variable declaration, conditionals, while loops, and the **forall** primitive for parallel execution. As usual, **if** b **then** C abbreviates **if** b **then** C **else skip**.

We write $FV(C)$ for the set of free variables in C, *i.e.*, variables which are not in the scope of local variable declaration of a block structure or a parallel command. We also write $MD(C)$ to denote the set of free variables which can be updated by a variable assignment, *i.e.*, variables that have a free occurrence of the form $x := \cdots$ in C. We assume that, for every parallel command **forall** $x \mapsto y_1,\ldots,y_n$ **in** σ **do** C, $MD(C) \subseteq \{x, y_1, \ldots, y_n\}$.

We give the formal semantics of this language in the style of big-step operational semantics that derives an evaluation relation either of the form $(s,h), C \rightsquigarrow (s',h')$ or $(s,h), C \rightsquigarrow \mathsf{abort}$. The former indicates that the command C with initial state (s,h) ends up with a final state (s',h'), while the latter stands for an execution aborted by an error.

Figure 4 gives derivation rules for commands, where the notation $[\![e]\!]s$ ($[\![b]\!]s$, *resp.*) denotes the result of evaluating the expression e (the boolean expression b, *resp.*) under the store s. We omitted the rules that lead to abort. A program can be aborted either by a memory fault (via a lookup or a mutation into a non-allocated address) or by an underallocation in the execution of **forall**, *i.e.*, the heap allocates fewer addresses than the required set σ of addresses designated in the **forall** command.

In the execution of the command **forall** $x \mapsto y_1,\ldots,y_n$ **in** σ **do** C, the command body C is simultaneously evaluated for every different address x taken from a fixed finite set of heap addresses explicitly given by σ. (In practice, σ could be automatically derived from,

$$\frac{}{(s,h), x := e \rightsquigarrow (s[x \mapsto [\![e]\!]s], h)} \qquad \frac{[\![e]\!]s \in dom(h)}{(s,h), x := [e] \rightsquigarrow (s[x \mapsto h([\![e]\!]s)], h)}$$

$$\frac{[\![e]\!]s \in dom(h)}{(s,h), [e] := e' \rightsquigarrow (s, h[[\![e]\!]s \mapsto [\![e']\!]s])} \qquad \frac{(s,h), C \rightsquigarrow (s'', h'') \qquad (s'', h''), C' \rightsquigarrow (s', h')}{(s,h), C; C' \rightsquigarrow (s', h')}$$

$$\frac{}{\mathbf{skip}, (s,h) \rightsquigarrow (s,h)} \qquad \frac{(s,h), C \rightsquigarrow (s', h')}{(s,h), \mathbf{begin\ var}\ x; C\ \mathbf{end} \rightsquigarrow (s'[x \mapsto s(x)], h')}$$

$$\frac{[\![Q]\!](s,h) = \mathbf{true} \qquad (s,h), C \rightsquigarrow (s', h')}{(s,h), \mathbf{if}\ Q\ \mathbf{then}\ C\ \mathbf{else}\ C' \rightsquigarrow (s', h')} \qquad \frac{[\![Q]\!](s,h) = \mathbf{false} \qquad (s,h), C' \rightsquigarrow (s', h')}{(s,h), \mathbf{if}\ Q\ \mathbf{then}\ C\ \mathbf{else}\ C' \rightsquigarrow (s', h')}$$

$$\frac{[\![b]\!]s = \mathbf{false}}{(s,h), \mathbf{while}\ b\ \mathbf{do}\ C \rightsquigarrow (s,h)} \qquad \frac{[\![b]\!]s = \mathbf{true} \qquad (s,h), C \rightsquigarrow (s'', h''), \mathbf{while}\ b\ \mathbf{do}\ C \rightsquigarrow (s', h')}{(s,h), \mathbf{while}\ b\ \mathbf{do}\ C \rightsquigarrow (s', h')}$$

$$\frac{\begin{array}{c} MD(C) \subseteq \{x, y_1, \ldots, y_n\} \qquad \sigma = [p_1, \ldots, p_m]\ (m \geq 1) \qquad h = h' * h'' \\ h' = h'_1 * \cdots * h'_m \qquad dom(h'_i) = \{p_i + d \mid 0 \leq d \leq n-1\} \text{ for every } i \in \{1, \ldots, m\} \\ (s[x \mapsto p_i, y_1 \mapsto h'(p_i), \ldots, y_n \mapsto h'(p_i + n - 1)], h'), C \rightsquigarrow (s_i, h''_i) \text{ for every } i \in \{1, \ldots, m\} \end{array}}{(s,h), \mathbf{forall}\ x \mapsto y_1, \ldots, y_n\ \ \mathbf{in}\ \sigma\ \mathbf{do}\ C \rightsquigarrow (s, \bigcup_{i=1}^{m} \{(p_i + d, h''_i(p_i + d)) \mid 0 \leq d \leq n-1\} * h'')}$$

Figure 4: Operational semantics of the data-parallel programming language

say, a reference name to a collection of heap allocated data, in a suitable extension of the present language.) Every different execution of the command body is in charge of updating the contents allocated in a contiguous block of size n starting from the address x, with $y_1, \ldots, y_n$ bound to the values stored in the block. The condition $h' = h'_1 * \cdots * h'_m$ in the premise of the operational semantics requires that the allocation addresses of different blocks do not overlap. The variables $x, y_1, \ldots, y_n$ are local to each execution instance; their variable assignments will be restored to the original ones, upon termination of the parallel execution.

In order to avoid inconsistencies that may arise by concurrent writes to the shared heap memory, we assume that every execution instance of a parallel command operates on its own local copy of the entire store and the heap blocks subject to the parallel processing. Every instance is allowed to read and write the local copy of the storage, except that global variables are read only. However, the effect of updates is only reflected to the local copy and is not accessible from other instances during parallel execution. Upon termination of the parallel execution, the store is restored to the original one and the contents of every heap block are updated to those of the local copy of the corresponding block held in the associated execution instance; the contents of irrelevant blocks in the local copy of each execution instance are discarded.

To summarize, an update to the heap is meaningful only if the updated address belongs to the block associated with the instance that executes the update. We also note that no execution instance can change the heap domain, as the language does not include commands for heap allocation and deallocation. There are many data-parallel programs that do not adhere to this limited set of heap operations, of course. However, the primary

$$\textsc{Asgn}\ \frac{}{\{P[e/x]\}\,x := e\,\{P\}} \qquad \textsc{Lkp}\ \frac{}{\{\exists z.(P[z/x] \wedge e \hookrightarrow z)\}\,x := [e]\,\{P\}}\ z \notin FV(e) \cup FV(P)$$

$$\textsc{Mut}\ \frac{}{\{e \mapsto - * (e \mapsto e' -\!\!* P)\}\,e := e'\,\{P\}} \qquad \textsc{Skip}\ \frac{}{\{P\}\,\mathbf{skip}\,\{P\}}$$

$$\textsc{Seq}\ \frac{\{P\}\,C\,\{Q'\} \qquad \{Q'\}\,C'\,\{Q\}}{\{P\}\,C;C'\,\{Q\}} \qquad \textsc{Blk}\ \frac{\{P\}\,C\,\{Q\}}{\{P\}\,\mathbf{begin\ var}\ x;\ C\ \mathbf{end}\,\{Q\}}\ x \notin FV(P) \cup FV(Q)$$

$$\textsc{If}\ \frac{\{P \wedge Q\}\,C\,\{P'\} \qquad \{P \wedge \neg Q\}\,C'\,\{P'\}}{\{P\}\,\mathbf{if}\ Q\ \mathbf{then}\ C\ \mathbf{else}\ C'\,\{P'\}} \qquad \textsc{While}\ \frac{\{P \wedge Q\}\,C\,\{P\}}{\{P\}\,\mathbf{while}\ Q\ \mathbf{do}\ C\,\{P \wedge \neg Q\}}$$

$$\textsc{Par}\ \frac{P \Rightarrow \Pi^n(\sigma) \qquad \left\{\begin{array}{c}((\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true}) \wedge \\ P \wedge x \hookrightarrow y_1, \cdots, y_n \wedge x = z\end{array}\right\} C \left\{\begin{array}{c}\exists y_1 \cdots y_n.(z \hookrightarrow y_1, \cdots, y_n \wedge \\ \diamond(Q \wedge z \hookrightarrow y_1, \cdots, y_n))\end{array}\right\}}{\{P\}\,\mathbf{forall}\ x \mapsto y_1, \cdots, y_n\ \mathbf{in}\ \sigma\ \mathbf{do}\ C\,\{Q\}}$$

$x, y_1, \ldots, y_n, z \notin FV(P) \cup FV(Q)$, $z \notin FV(C)$, and Q is strictly exact.

$$\textsc{Conseq}\ \frac{P \Rightarrow P' \qquad \{P'\}\,C\,\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\,C\,\{Q\}} \qquad \textsc{ExQ}\ \frac{\{P\}\,C\,\{Q\}}{\{\exists x.P\}\,C\,\{\exists x.Q\}}\ x \notin FV(C)$$

$$\textsc{Disj}\ \frac{\{P_1\}\,C\,\{Q_1\} \qquad \{P_2\}\,C\,\{Q_2\}}{\{P_1 \vee P_2\}\,C\,\{Q_1 \vee Q_2\}} \qquad \textsc{Frame}\ \frac{\{P\}\,C\,\{Q\}}{\{P * R\}\,C\,\{Q * R\}}\ MD(C) \cap FV(R) = \emptyset$$

Figure 5: Inference rules for Hoare triples

goal of this paper is to develop a formal proof system that gives a clear logical account for pointer jumping, a common programming technique in parallel computing. We presumed this reduced pattern of heap operations for the sake of a simpler specification of the parallel command.

4.1 Hoare logic for data-parallelism

We define a specification $\{P\}\,C\,\{Q\}$ is valid iff $(s,h), C \rightsquigarrow (s,h')$ implies $s', h' \models Q$, for all $(s,h), (s',h') \in State$ satisfying $s,h \models P$,

The inference rules for deriving valid specifications are given in Figure 5. The upper half of the figure consists of the rules for commands. Most of the rules are standard except for Lkp, Mut, and Par. Lkp and Mut give a weakest (liberal) precondition for the lookup and mutation command, respectively, where logically equivalent variant rules can substitute for them [8, 11].

The inference rule Par is explained as follows. The condition $P \Rightarrow \Pi^n(\sigma)$ in the premise indicates that the heap mentioned by the precondition P has to disjointly allocate a contiguous block of size n at every address listed in σ. The conjunct $(\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true}$ of the precondition implies that in any execution instance x must be bound to the initial address of a separate block. The rest of conjuncts $P \wedge x \hookrightarrow y_1, \ldots, y_n$ indicates that $y_1, \ldots, y_n$ must be bound to the contents allocated in the block.

Every different execution of the command body C is intended to update a contiguous block (referred to by x) with contents as mentioned by the postcondition Q. However, simply putting $\exists y_1 \cdots y_n.(Q \wedge x \hookrightarrow y_1,\ldots,y_n)$ overspecifies the postcondition of the command body, since every single execution instance of the command body is only in charge of updating the block allocated at the address x and does not care about other blocks. (If it ever updates other blocks, the effect will be canceled by other parallel execution instances that are in charge of updating those blocks.)

Here we utilize the modal operator as follows in order to mention about such a limited portion of heap addresses out of the entire set of addresses:

$$\exists y_1,\cdots,y_n.(x \hookrightarrow y_1,\cdots,y_n \wedge \Diamond(Q \wedge x \hookrightarrow y_1,\cdots,y_n)).$$

The first conjunct indicates that the heap allocates contents $y_1,\ldots,y_n$ at the address x; The second conjunct requires the contents $y_1,\ldots,y_n$ to respect the assertion Q but leaves those contents allocated at other addresses unspecified.

The inference rule comes with a few side conditions for technical reasons. In the precondition of the premise, the variable x is aliased to a fresh variable z, because the denotation of x may be altered by an assignment. Variables $x,y_1,\ldots,y_n$ are local to every sequential execution of the subcommand C and hence they are assumed not free in P or Q. Finally, we require that Q be a strictly exact assertion.

The condition that Q be a strictly exact assertion is vital for the soundness of the inference rule. When we reason about, somewhat informally, a data-parallel execution, we resort to the assumption that every different parallel execution of the same command ends up with a single unique result as specified by the postcondition. If Q is strictly exact, this uniqueness is guaranteed. Otherwise, inconsistencies may arise. For instance, consider a derivation for a parallel command $\{P\}\,\mathbf{forall}\,x \mapsto y\ \mathbf{in}\,[1001,1002]\,\mathbf{do}\,C\,\{Q\}$ with $Q \triangleq (1001 \mapsto 1 * 1002 \mapsto 3) \vee (1001 \mapsto 4 * 1002 \mapsto 6)$, which is not a strictly exact assertion. Then the execution of the parallel command can result in a heap, say, $h = \{(1001,1),(1002,6)\}$, as this adheres to $\exists y.(\Diamond(Q \wedge z \hookrightarrow y) \wedge z \hookrightarrow y)$. However h does not satisfy Q and therefore the inference is not sound.

Theorem 4.1 (Soundness). *If $\{P\}\,C\,\{Q\}$ is derivable, then $\{P\}\,C\,\{Q\}$ is valid.*

Proof. The soundness of the FRAME rule follows from the safety monotonicity and the frame property [13], which are proved by induction on the size of derivation of evaluation relations.

To show the soundness of the PAR rule, suppose we have a derivation that ends up with a conclusion $\{P\}\,\mathbf{forall}\,x \mapsto y_1,\cdots,y_n\ \mathbf{in}\,\sigma\,\mathbf{do}\,C\,\{Q\}$, with $\sigma = [p_1,\ldots,p_m]$ $(m \geq 1)$. Let (s,h) be any state such that $s,h \models P$. By the premise $P \Rightarrow \Pi^n(\sigma)$ of the inference rule, we have $h = h_1 * \cdots * h_m$ where $h_i = h \upharpoonright \{p_i + d \mid 0 \leq d \leq n-1\}$ $(i \in \{1,\cdots,m\})$. Let $s_i = s[x \mapsto p_i, y_1 \mapsto h(p_i),\ldots,y_n \mapsto h(p_i+n-1)]$. Since $x,y_1,\ldots,y_n,z \notin FV(P)$, we have $s_i[z \mapsto p_i],h \models ((\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true}) \wedge P \wedge x \hookrightarrow y_1,\cdots,y_n \wedge x = z$. Hence, by induction hypothesis, for every i, we have $(s_i[z \mapsto p_i],h),C \rightsquigarrow (s'_i,h'_i)$, for some $(s'_i,h'_i) \in \mathit{State}$ satisfying $\mathit{dom}(h'_i) = \mathit{dom}(h)$ and

$$s'_i,h'_i \models \exists y_1 \cdots y_n.(z \hookrightarrow y_1,\cdots,y_n \wedge \Diamond(Q \wedge z \hookrightarrow y_1,\cdots,y_n)). \qquad (\dagger)$$

Here we notice that $s'_i[z \mapsto s(z)]$ agrees with s for any variable other than $MD(C)$. Hence, it follows from $x, y_1, \ldots, y_n, z \notin FV(Q)$ and (†) that $s, h''_i \models Q$ for some h''_i such that $h''_i(q) = h'_i(q)$ for every $i \in \{1, \ldots, m\}$ and $q \in \{p_i + d \mid 0 \leq d \leq n-1\}$. Since Q is strictly exact, we have $h''_i = h''_j$ for every $i, j \in \{1, \cdots, m\}$. This implies that $s, h' \models Q$ holds, where $h' = \bigcup_{i=1}^{m} \{(p_i + d, h'_i(p_i + d)) \mid 0 \leq d \leq n-1\}$.

The proof of other rules is rather a routine and is omitted. □

5 The Correctness Proof for Parallel List Scan

This section gives a proof for Proposition 3.1 and shows the correctness of the parallel list scan program given in Figure 1.

In the proof, we will exploit the properties of assertions listed in Figure 6. In addition to general properties of the classical BI logic [8], the list contains those specific to connectives $\mapsto$, $\hookrightarrow$ and also to the subclasses of pure and precise assertions. Note that the given set of axioms and inference rules are by no means complete; neither they are not minimal in the sense that some properties are derivable from others. In the subsequent formal derivation, the rules in Figure 6 that have no reference number will be used in the subsequent proof without explicitly mentioned.

Lemma 5.1. *Let n be any positive integer. Then $R(i, p, \sigma, \tau)$ is strictly exact, $\Pi^n(\sigma)$ is precise. Also, the following formulas are all valid.*

(a) $\Pi^n(\sigma) \Rightarrow \Pi^1(\sigma) * \mathbf{true}$

(b) $\Pi^n(\sigma) \Leftrightarrow \Pi^n(\sigma_1) * \Pi^n(\sigma_2)$, *whenever* $\mathtt{part}(\sigma, \sigma_1, \sigma_2)$.

(c) $R(i, p, p' :: \sigma, n :: \tau) \Rightarrow p \hookrightarrow n, - \wedge p = p'$

(d) $R(i, p, [p'], [n]) \Leftrightarrow p' \mapsto n, \mathsf{nil} \wedge p = p' \wedge i \geq 0$

(e) $\Diamond R(i, p, \sigma, \tau) \Leftrightarrow \Diamond R(i+1, p, \sigma, \tau) \wedge i \geq 0$

(f) $(\Pi^1(\sigma) \wedge p \hookrightarrow -, -) * \mathbf{true} \wedge R(0, p, \sigma, \tau) \Rightarrow \mathbf{false}$

(g) $R(i, p_1, p_1 :: \sigma_1, \tau_1) * (R(i, p_2, \sigma_2, \tau_2) \wedge p_2 + 1 \hookrightarrow \mathsf{nil}) \Rightarrow p_1 + 1 \hookrightarrow \mathsf{nil}$,
whenever $\mathtt{part}(\sigma, p_1 :: \sigma_1, \sigma_2)$ *and* $\mathtt{alts}(\tau, \tau_1, \tau_2)$ *for some* σ *and* τ.

We omit the proof, which is by routine induction on i and the length of σ.

Proof of proposition 3.1(a)

By the inference rule PAR, it is sufficient to show the derivability of the specification:

$$\left\{ \begin{array}{c} ((\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true}) \wedge \\ R(i, p, \sigma, \tau) \wedge x \hookrightarrow n, t \wedge x = z \end{array} \right\} C_{\mathbf{forall}} \left\{ \begin{array}{l} \exists nt.(z \hookrightarrow n, t \wedge \\ \quad \Diamond(R(i+1, p, \sigma, \tau) \wedge z \hookrightarrow n, t)) \end{array} \right\}. \quad (5.11)$$

Proof is by induction on i and the length of σ (and τ).

$$(P*Q)*R \Leftrightarrow P*(Q*R) \qquad P*Q \Leftrightarrow Q*P \qquad P*\mathbf{emp} \Leftrightarrow P$$
$$(P_1 \vee P_2)*Q \Leftrightarrow (P_1*Q)\vee(P_2*Q) \qquad (P_1\wedge P_2)*Q \Rightarrow (P_1*Q)\wedge(P_2*Q)$$
$$(\exists x.P)*Q \Leftrightarrow \exists x.(P*Q) \quad \text{and} \quad (\forall x.P)*Q \Rightarrow \forall x.(P*Q) \quad \text{where } x \text{ is not free in } Q$$
$$\frac{P_1 \Rightarrow Q_1 \quad P_2 \Rightarrow Q_2}{P_1*P_2 \Rightarrow Q_1*Q_2} \qquad \frac{P*Q \Rightarrow R}{P \Rightarrow (Q \mathrel{-\!\!*} R)} \qquad \frac{P \Rightarrow (Q \mathrel{-\!\!*} R)}{P*Q \Rightarrow R} \qquad \frac{P \Rightarrow Q}{\Diamond P \Rightarrow \Diamond Q}$$
$$P \Rightarrow \Diamond P \qquad \Diamond(P\vee Q) \Leftrightarrow \Diamond P \vee \Diamond Q \qquad \Diamond(P\wedge Q) \Rightarrow \Diamond P \wedge \Diamond Q$$
$$\exists x.\Diamond P \Leftrightarrow \Diamond\exists x.P \qquad \Diamond(P*Q) \Leftrightarrow \Diamond P * \Diamond Q$$

$$x \mapsto y_1,\cdots,y_n \wedge x' \mapsto y'_1,\cdots,y'_n \Leftrightarrow x \mapsto y_1,\cdots,y_n \wedge x = x' \wedge \textstyle\bigwedge_{i=1}^n y_i = y'_i \tag{5.1}$$
$$x = x' \wedge x \hookrightarrow y \wedge x' \hookrightarrow y' \Rightarrow y = y' \tag{5.2}$$
$$x \mapsto - * x' \mapsto - * \mathbf{true} \Rightarrow x \neq x' \tag{5.3}$$
$$\textstyle\bigwedge_{i=1}^n (x+i-1 \hookrightarrow y_i) \Leftrightarrow x \hookrightarrow y_1,\ldots,y_n \tag{5.4}$$
$$x \mapsto y_1,\cdots,y_n \wedge x' \mapsto y'_1,\cdots,y'_n \Leftrightarrow x \mapsto y_1,\cdots,y_n \wedge x' \hookrightarrow y'_1,\cdots,y'_n \tag{5.5}$$
$$\Diamond(x \mapsto y) \Leftrightarrow x \mapsto - \tag{5.6}$$
$$(P_1*P_2)\wedge x \hookrightarrow y \Leftrightarrow (P_1 \wedge x \hookrightarrow y)*P_2 \vee P_1*(P_2 \wedge x \hookrightarrow y) \tag{5.7}$$

For any pure assertions P, P_1, P_2,

$$P_1 \wedge P_2 \Leftrightarrow P_1*P_2 \quad (P\wedge Q)*R \Leftrightarrow P\wedge(Q*R) \quad P \Leftrightarrow \Diamond P \quad \Diamond(P\wedge Q) \Leftrightarrow P \wedge \Diamond Q$$
$$((P\wedge Q) \mathrel{-\!\!*} (P\wedge R))\wedge P \Rightarrow Q \mathrel{-\!\!*} (P\wedge R) \tag{5.8}$$

For any precise assertions P, P_1, P_2,

$$(P*Q_1)\wedge(P*Q_2) \Leftrightarrow P*(Q_1\wedge Q_2) \tag{5.9}$$
$$(P_1\wedge Q_1)*P_2 \wedge P_1*(P_2\wedge Q_2) \Leftrightarrow (P_1\wedge Q_1)*(P_2\wedge Q_2) \tag{5.10}$$

Figure 6: Properties of assertions

Case $\sigma = \tau = []$. It vacuously holds by the rule CONSEQ.

Case $\sigma = [p']$ and $\tau = [n']$. We have $R(i,p,\sigma,\tau)\wedge x \hookrightarrow n,t \wedge x = z \Leftrightarrow i \geq 0 \wedge x \mapsto n,t \wedge n = n' \wedge x = z \wedge z = p \wedge p = p' \wedge t = \mathsf{nil}$ by lemma 5.1(d), (5.5), and (5.1). Since the conjunction of this formula and $t \neq \mathsf{nil}$ leads to absurdity, by the rules IF and CONSEQ, (5.11) is derived from the implication $x \mapsto n,t \wedge n = n' \wedge x = z \wedge z = p \wedge p = p' \wedge t = \mathsf{nil} \Rightarrow \exists nt.(z \mapsto n,t \wedge \Diamond(p' \mapsto n',\mathsf{nil} \wedge p = p' \wedge z \hookrightarrow n,t))$, which follows from (5.1), (5.5), and (5.6).

Case $\sigma = p_1 :: p_2 :: \sigma'$, $\tau = n_1 :: n_2 :: \tau'$, and $i = 0$. We prove by case analysis on the equality of p and z.

First consider the case $p = z$. Below we show the proof outline for the **then** clause of the conditional in $C_{\mathbf{forall}}$:

$\{(p_1 \mapsto n_1, p_2 \wedge x \hookrightarrow n,t) * p_2 \mapsto n_2, q \wedge p = p_1 \wedge x = z\}$

$$\{(x \mapsto -,- * p_2 \mapsto -,-) \wedge t \hookrightarrow n_2, q \wedge p = p_1 \wedge p_1 = z \wedge n_1 = n \wedge x = z\} \quad \text{by (5.5)}$$

$$m := [t];\, t := [t+1]$$

$$\{(x \mapsto -,- * p_2 \mapsto -,-) \wedge p = p_1 \wedge p_1 = z \wedge n_1 = n \wedge n_2 = m \wedge q = t \wedge x = z\}$$

$$\begin{aligned}&\{(x \mapsto -,- * p_2 \mapsto -,-) * (x \mapsto (n+m), t \mathrel{-\!\!*} x \mapsto (n_1+n_2), q) \\ &\quad \wedge p = p_1 \wedge p_1 = z \wedge x = z\}\end{aligned} \quad \text{— (*)}$$

$$[x] := n+m;\, [x+1] := t$$

$$\{(x \mapsto (n_1+n_2), q * p_2 \mapsto -,-) \wedge p = p_1 \wedge p_1 = z \wedge x = z\}$$

$$\{(z \mapsto (n_1+n_2), q * p_2 \mapsto -,-) \wedge p = p_1 \wedge p_1 = z\}.$$

The proof step $(*)$ is derived as follows. Let $P_0 \triangleq n_1 = n \wedge n_2 = m \wedge q = t$. Then we have $\mathbf{emp} \wedge P_0 \Rightarrow (x \mapsto (n_1+n_2), q \wedge P_0 \mathrel{-\!\!*} x \mapsto (n_1+n_2), q \wedge P_0) \wedge P_0 \Rightarrow x \mapsto (n+m), t \mathrel{-\!\!*} x \mapsto (n_1+n_2), q$ by $\mathbf{emp} \Rightarrow Q \mathrel{-\!\!*} Q$, (5.8) and (5.1).

Thus we have the proof outline for $C_{\mathbf{forall}}$ as below, where σ'_1, σ'_2, τ'_1, τ'_2 denote sequences satisfying $\mathtt{part}(p_1 :: p_2 :: \sigma', p_1 :: \sigma'_1, p_2 :: \sigma'_2)$ and $\mathtt{alts}(n_1 :: n_2 :: \tau', (n_1 + n_2) :: \tau'_1, \tau'_2)$.

$$\{p = z \wedge (\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true} \wedge R(0, p, \sigma, \tau) \wedge x \hookrightarrow n, t \wedge x = z\}$$

$$\{(p_1 \mapsto n_1, p_2 \wedge x \hookrightarrow n, t) * (p_2 \mapsto n_2, q) * R(0, q, \sigma', \tau') \wedge p = p_1 \wedge x = z\}$$

$$C_{\mathbf{forall}} \quad \text{by FRAME, EXQ, (5.7), (5.3)}$$

$$\{(z \mapsto (n_1+n_2), q * p_2 \mapsto -,-) * R(0, q, \sigma', \tau') \wedge p = p_1 \wedge p_1 = z\}$$

$$\begin{aligned}&\{z \hookrightarrow (n_1+n_2), q \wedge \Diamond(p = p_1 \wedge p_1 = z \wedge (p_1 \mapsto (n_1+n_2), q \\ &\qquad \wedge z \hookrightarrow (n_1+n_2), q) * R(0, q, \sigma'_1, \tau'_1) * R(0, p_2 :: \sigma'_2, \tau'_2))\}\end{aligned} \quad \begin{aligned}&\text{by (5.1), (5.6),}\\ &\text{and lemma 5.1(e)}\end{aligned}$$

$$\{\exists nt.(z \hookrightarrow n, t \wedge \Diamond(R(1, p, \sigma, \tau) \wedge z \hookrightarrow n, t))\} \quad \text{by (5.7) and EXQ.}$$

Let $\sigma'_1, \sigma'_2, \tau'_1, \tau'_2$ be defined as above. The other case $p \neq z$ is proved as follows.

$$\{p \neq z \wedge (\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true} \wedge R(0, p, \sigma, \tau) \wedge x \hookrightarrow n, t \wedge x = z\}$$

$$\begin{aligned}&\{p_1 \mapsto - * (\Pi^1(p_2 :: \sigma') \wedge x \hookrightarrow -) * \mathbf{true} \\ &\quad \wedge p_1 \mapsto n_1, p_2 * (R(0, p_2, p_2 :: \sigma', n_2 :: \tau') \wedge x \hookrightarrow n, t) \wedge p = p_1 \wedge x = z\}\end{aligned} \quad \text{by EXQ, (5.7), (5.3)}$$

$$\begin{aligned}&\{(p = p_1 \wedge p_1 \mapsto n_1, p_2) * (\Pi^1(p_2 :: \sigma') \wedge x \hookrightarrow -) * \mathbf{true} \\ &\qquad \wedge R(0, p_2, p_2 :: \sigma', n_2 :: \tau') \wedge x \hookrightarrow n, t \wedge x = z\}\end{aligned} \quad \begin{aligned}&\text{by (5.5), (5.1), (5.9), (5.7),}\\ &\text{(5.4), lemma 5.1(f)}\end{aligned}$$

$$C_{\mathbf{forall}}$$

$$\begin{aligned}&\{(p = p_1 \wedge p_1 \mapsto n_1, p_2) * \\ &\quad \exists nt.(z \hookrightarrow n, t \wedge \Diamond(R(1, p_2, p_2 :: \sigma, n_2 :: \tau) \wedge z \hookrightarrow n, t))\}\end{aligned} \quad \text{by I.H. and FRAME}$$

$$\begin{aligned}&\{\exists nt.(z \hookrightarrow n, t \wedge \Diamond(p_1 \mapsto (n_1+n_2), q * \\ &\qquad R(0, q, \sigma'_1, \tau'_1) * R(0, p_2 :: \sigma'_2, \tau'_2) \wedge p = p_1 \wedge z \hookrightarrow n, t)\}\end{aligned} \quad \text{by (5.6)}$$

$$\{\exists nt.(z \hookrightarrow n, t \wedge \Diamond(R(1, p, \sigma, \tau) \wedge z \hookrightarrow n, t))\}. \quad \text{by EXQ}$$

Combining the two cases, we derive the specification (5.11) by the rule DISJ.

Case $\sigma = p_1 :: p_2 :: \sigma'$, $\tau = n_1 :: n_2 :: \tau'$, and $i > 0$. Let σ'_1, σ'_2, τ_1, τ_2 be sequences satisfying $\texttt{part}(\sigma', \sigma'_1, \sigma'_2)$ and $\texttt{alts}(\tau, \tau_1, \tau_2)$, and also define $R_j(i) \triangleq R(i, p_j, p_j :: \sigma'_j, \tau_j)$ for every $j \in \{1,2\}$. Then for every $j \in \{1,2\}$ we have:

$\{p = p_1 \wedge (\Pi^1(\sigma) \wedge x \hookrightarrow -) * \mathbf{true} \wedge R_j(i-1) * (R_{3-j}(i-1) \wedge x \hookrightarrow n,t) \wedge x = z\}$

$\{(p = p_1 \wedge R_j(i-1)) * ((\Pi^1(\sigma_{3-j}) \wedge x \hookrightarrow -) * \mathbf{true}$
$\quad \wedge R_{3-j}(i-1) \wedge x \hookrightarrow n,t \wedge x = z)\}$ — by lemma 5.1(a) and (b), (5.7), (5.10), (5.3)

$\quad C_{\mathbf{forall}}$

$\{(p = p_1 \wedge R_j(i-1)) * \exists nt.(z \hookrightarrow n,t \wedge \Diamond(R_{3-j}(i) \wedge z \hookrightarrow n,t))\}$ by I.H. and FRAME

$\{\exists nt.(z \hookrightarrow n,t \wedge \Diamond(p = p_1 \wedge R_j(i) * R_{3-j}(i) \wedge z \hookrightarrow n,t))\}$ by lemma 5.1(e) and (5.7).

Therefore (5.11) follows from $R(i,p,\sigma,\tau) \Leftrightarrow p = p_1 \wedge R_1(i-1) * R_2(i-1)$ and (5.7) by the rule DISJ. □

Proof of proposition 3.1(b)

When $\sigma = []$, the assertion vacuously holds. Suppose $\sigma \neq []$. Then we have:

$\{R(i,p,\sigma,w) \wedge p+1 \hookrightarrow q\}$

$\{R(i,p,\sigma,w)\}$

$\quad C_{\mathbf{forall}}$ — by proposition 3.1(a)

$\{R(i+1,p,\sigma,w)\}$

$\{\exists z.(R(i+1,p,\sigma,w) \wedge p+1 \hookrightarrow z \wedge p+1 \hookrightarrow z)\}$ — by lemma 5.1(c)

$\quad q := [p+1]$

$\{(R(i+1,p,\sigma,w) \wedge p+1 \hookrightarrow q\}$

Thus the assertion follows by the rules EXQ and WHILE. □

Proof of proposition 3.1(c)

By induction on i and the length of σ (and τ).

When $i = 0$, we have $R(0,p,p_1 :: \sigma, n_1 :: \tau) \wedge p+1 \hookrightarrow \mathsf{nil} \Rightarrow \exists q.(p \hookrightarrow n_1, q * R(0,q,\sigma',\tau')) \wedge p = p_1 \wedge p+1 \hookrightarrow \mathsf{nil} \Rightarrow \exists q.(R(0,q,\sigma,\tau) \wedge q = \mathsf{nil}) * (p = p_1 \wedge p \hookrightarrow n_1, \mathsf{nil})$ by (5.2). This indicates that $\sigma = \tau = []$ and thus $\texttt{sum}(n_1 :: \tau, n_1)$.

Suppose $i > 0$. The case $\sigma = \tau = []$ is likewise proved. Let $\sigma', \sigma'_1, \sigma'_2, \tau', \tau'_1, \tau'_2$ be sequences satisfying $\sigma = p_2 :: \sigma'$, $\tau = n_2 :: \tau'$, $\texttt{part}(\sigma', \sigma'_1, \sigma'_2)$, and $\texttt{alts}(n_2 :: \tau', n_2 :: \tau'_2, \tau'_1)$, and also let n' be an integer such that $\texttt{alts}(n_1 :: \tau, n')$. Then we have:

$\quad R(i,p,p_1 :: \sigma, n_1 :: \tau) \wedge p+1 \hookrightarrow \mathsf{nil}$

$\Rightarrow (R(i-1,p_1,p_1 :: \sigma'_1, (n_1+n_2) :: \tau'_1) \wedge p_1 + 1 \hookrightarrow \mathsf{nil})$
$\quad * R(i-1,p_2,p_2 :: \sigma'_2, n_2 :: \tau'_2) \wedge p = p_1$ — by (5.7), (5.3)

$\Rightarrow \exists q.(R(i-1,q,\sigma'_1,\tau'_1) \wedge (q = \mathsf{nil} \vee q+1 \hookrightarrow \mathsf{nil}))$
$\quad * R(i-1,p_2,p_2 :: \sigma'_2, n_2 :: \tau'_2) * (p = p_1 \wedge p_1 \hookrightarrow n', \mathsf{nil})$ — by induction hypothesis

$$\begin{aligned}
&\Rightarrow\ (R(i-1,p_2,p_2::\sigma_2',n_2::\tau_2')\wedge p_2+1\hookrightarrow \mathsf{nil}) \\
&\qquad *\,\exists q.R(i-1,q,\sigma_1',\tau_1')*(p=p_1\wedge p_1\hookrightarrow n',\mathsf{nil}) \qquad \text{— (**)}\\
&\Rightarrow\ (R(i,p_2,\sigma,\tau)\wedge p_2+1\hookrightarrow \mathsf{nil})*(p=p_1\wedge p_1\hookrightarrow n',\mathsf{nil}) \qquad \text{by (3.4), (3.5), (5.7)}
\end{aligned}$$

The implication $(**)$ is derived as follows. If $q=\mathsf{nil}$, then $R(i-1,q,\sigma_1',\tau_1')$ implies $\sigma_1'=[]$ and thus $\sigma_2'=[]$ by (3.1). Hence $R(i-1,p_2,p_2::\sigma_2',n_2::\tau_2')\Rightarrow p_2+1\hookrightarrow\mathsf{nil}$. Otherwise, we have $(R(i-1,q,\sigma_1',\tau_1')\wedge q+1\hookrightarrow\mathsf{nil})*R(i-1,p_2,p_2::\sigma_2',n_2::\tau_2')\Rightarrow$ $p_2+1\hookrightarrow\mathsf{nil}$ by lemma 5.1(g). $\square$

Theorem 5.1. *Let $C_{\mathbf{scan}}$ denote the program in Figure 1. Then the following specification is valid.*

$$\{R(0,p,\sigma,\tau)\wedge p\neq\mathsf{nil}\}\,C_{\mathbf{scan}}\,\{\exists i.R(i,p,\sigma,\tau)\wedge p+1\hookrightarrow\mathsf{nil}\}$$

This theorem follows from proposition 3.1(b). As a corollary, we can deduce from proposition 3.1(c) that the program computes the list scan, *i.e.*, when the program terminates, every cons cell holds the sum of the corresponding sublist.

6 Conclusion

We have proposed a program logic for reasoning about data-parallel programs. We have worked out a formal correctness proof for the parallel list scan algorithm that employs the pointer jumping, a common method for parallel processing of linked data structures. The proof system adopts Separation Logic as a formal means to represent disjoint partitioning of linked data structures and further extends it with modality to provide a sound specification for the data-parallel command **forall**. This enables us to formally present the parallel list scan algorithm as another instance of the divide-and-conquer strategy.

We believe that our program logic can also apply to other variants of parallel algorithms based on parallel prefix or scan [2]. However, in the present paper, it is assumed that each parallel execution instance can only update heap contents owned by the instance itself. In some parallel algorithms, it is vital that every execution instance is possible to update heap contents owned by other instances. Such algorithms are more difficult to verify, because of possible race conditions caused by concurrent writes to the heap storage. It would be an interesting future topic to refine the present proof system for allowing concurrent writes. We hope that the notion of ownership transfer [10] might give a relevant solution to this issue.

References

[1] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *International Conference on Functional Programming (ICFP 2005)*, pages 280–293. ACM Press, 2005.

[2] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, November 1990.

[3] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 213–225. ACM Press, 1996.

[4] Luc Bougé, David Cachera, Yann Le Guyadec, Gil Utard, and Bernard Virot. Formal validation of data-parallel programs: A two-component assertional proof system for a simple language. *Theoretical Computer Science*, 189(1-2):71–107, 1997.

[5] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[6] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[7] W.D. Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of ACM*, 29(12):1170–1183, 1986.

[8] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–26. ACM Press, 2001.

[9] Susumu Nishimura and Atsushi Ohori. Parallel functional programming via data-parallel recursion. *Journal of Functional Programming*, 9(4):427–463, 1999.

[10] Peter O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 — Concurrency Theory, 15th International Conference*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.

[11] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[12] Élodie-Jane Sims. Extending separation logic with fixpoints and postponed substitution. In *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004*, volume 3116 of *LNCS*, pages 475–490. Springer, 2004.

[13] Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.