Electronic Preprint for Journal of Information Processing Vol.29

Regular Paper

An Extensionally Equivalence-ensured Language for Task Parallel Processing with Backtracking-based Load Balancing

Tatsuya Abe[1,a)] Tasuku Hiraishi[2,b)]

Received: September 25, 2020, Accepted: January 15, 2021

Abstract: Backtracking-based load balancing is a promising method for task parallel processing with work stealing. Tascell is a framework for developing applications with backtracking-based load balancing. Users are responsible for ensuring the consistent behavior of Tascell programs when backtracking is triggered in the Tascell runtimes. Nevertheless, the operational semantics for Tascell programs have not been formally studied. Moreover, no extensional equivalence between Tascell programs is provided. In this paper, we formally specify operational semantics for Tascell programs and define extensional equivalence between Tascell programs using the Church–Rosser modulo equivalence notion in abstract rewriting theory. We propose left invertibility and well-formedness properties for Tascell programs, which ensure extensional equivalence between sequential and concurrent behaviors of Tascell programs. We also propose a domain-specific language based on reversible computation, which allows only symmetric pre/post-processing to update states. Tascell programs written in our language have left invertibility and well-formedness properties by construction. Finally, we confirm that Tascell programs to solve typical search problems such as pentomino puzzles, N-queens, and traveling salesman problems can be written in our language.

Keywords: abstract rewriting theory, backtracking, Church–Rosser modulo equivalence, domain-specific language, extensional equivalence, load balancing, reversible computation, search problem, Tascell, task parallel processing, work stealing

1. Introduction

Task parallel processing is a promising method for using a huge number of computational nodes for solving search problems such as those related to data parallel processing. Unfortunately, task parallel processing often has *the load imbalance problem* whereby tasks of uneven sizes are assigned to computing workers, unlike data parallel processing, in which tasks assigned to workers are usually of uniform size. The most straightforward method for addressing the load imbalance problem is to create numerous fine-grained tasks to be assigned to workers and put them into pools, called *task pools*, in advance. However, such a method raises the overhead of task creation and complicates task pool management.

Lazy task creation (LTC) is a well-known method for reducing the overhead for fine-grained task parallel programs. In LTC, a worker *logically* defines tasks at task creation points but does not create real tasks. When a task is *stolen* by another worker, it is embodied as a real task. Consequently, the cost of task creation can be delayed until the task is required. Mohr et al. used this technique to implement so-called *future* construct of Multilisp [28]. Although they drastically reduced the overhead of task creation, they described that pools are still required for logical tasks.

Inspired by work of Mohr et al., Frigo et al. adopted LTC in the implementation of the Cilk-5 multithreaded language [13]. Cilk-5, an extension of the C programming language [21], is extended mainly with two additional constructs, namely, *spawn* for creating parallel tasks and *sync* for synchronizing tasks. In Cilk-5, doubly ended queues called *ready deques* are used as logical task pools that workers have. Although they demonstrated some experiments that show high performance of task parallel processing in Cilk-5, considerable overhead remains necessary for managing the logical task pools.

Backtracking-based load balancing is a promising alternative technique in task parallel processing [19]. A worker performs sequential computation without creating any logical task until it receives a task request from another idle worker. Although it might seem that the worker *cannot* create a *large* task to be sent if the computation by the worker has proceeded beyond the execution point where it can create the task, we use an example to demonstrate later that this is not the case in backtracking-based load balancing.

Let us consider depth-first backtrack search for solving pentomino puzzles (cf., Golomb's book [14]), which consist of one board and 12 pieces, each of which has 5 cells. The board has 60 cells. A solution is a board on which the 12 pieces are placed without overlap or gaps. The two panels in **Fig. 1** show a search tree. Squares denote boards. Pieces are put on the boards. Every

[1] STAIR Lab, Chiba Institute of Technology, Narashino, Chiba 275–0016, Japan
[2] Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
[a)] abet@stair.center
[b)] tasuku@media.kyoto-u.ac.jp

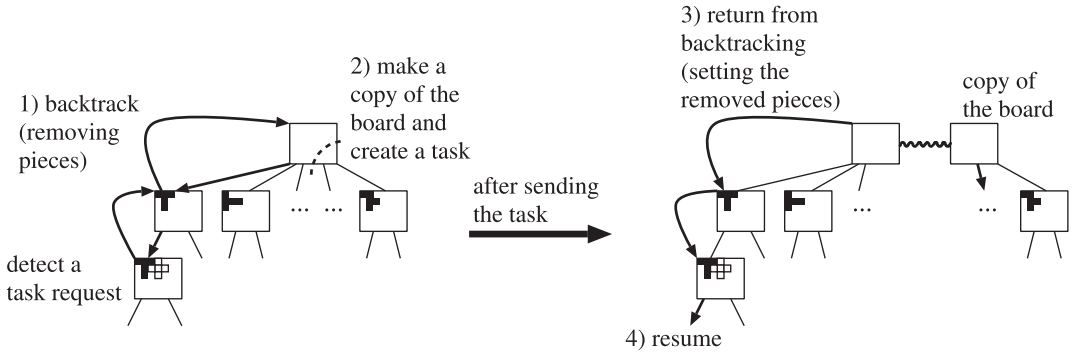

Fig. 1 Search tree for a pentomino puzzle.

plain line denotes an operation that puts one piece on a board.

A case is considered in which a worker called a *victim* that handles the leftmost board in the left panel receives a task request from another worker called a *thief*. If the victim creates a task *without* backtracking to a former state, then the task will correspond to the sub-search tree on the right-hand side, i.e., the task will be small. To enlarge the task to be created, the victim 1) temporarily backtracks to the former state, 2) creates the task to execute part of unexecuted iterations of the loop over pieces at the state and sends it to the thief, 3) returns from the temporary backtracking to the original state immediately, and 4) resumes its own search. The right panel in Fig. 1 presents an execution state after the victim sent the task to the thief. The right-hand search tree of the wavy line has been assigned to the thief.

By definition, the backtracking-based load balancing technique eliminates the overhead caused by unnecessary task creation and task pool management. Another advantage of this technique is that we can delay various additional operations that are required for parallelization. For example, when the backtrack search for pentomino puzzles is implemented with LTC, a worker must allocate a workspace for managing the board states and initialize it by making a copy of the current board state for each logical task creation. When using the backtracking-based technique, however, the cost for such operations can be delayed until a task request, which considerably reduces total allocation/initialization cost and improves the locality of reference.

Careful readers might think that backtrackings to create tasks are more expensive than LTC. However, because the number of task creation in LTC is much larger than the number of work steals in most cases, the backtracking-based technique can improve the overall performance considerably, as experimentally demonstrated [19].

The Tascell framework, which is intended for developing applications with backtracking-based load balancing, consists of a language, a compiler, and runtimes. Like Cilk-5, the Tascell language is an extension of the C programming language. The Tascell language has special clauses *before*, *body*, and *after* for backtracking. Users develop Tascell programs using these special clauses according to conventions described in a natural language in the original paper [19]. Users are responsible for ensuring the consistent behavior of Tascell programs when backtracking is triggered in the Tascell runtimes. In fact, *not all* tasks that can be executed in parallel without backtracking-based load balancing can be executed correctly in the Tascell runtimes because of backtracking.

In this paper, we provide operational semantics for Tascell programs and define extensional equivalence between Tascell programs using the Church–Rosser modulo equivalence notion in abstract rewriting theory [38]. Although semantics for task parallel programs have been reported [5], [12], our extensional equivalence is not only novel but also theoretically simple and elegant based on abstract rewriting theory.

We propose left invertibility and well-formedness properties for Tascell programs, which ensure extensional equivalence between the sequential and concurrent behaviors of Tascell programs. The construction of the proof flow is also a contribution to the task parallel programming research field.

We also design a domain-specific programming language for task parallel processing with backtracking-based load balancing that ensures left invertibility by construction. A key idea is to use reversible computation [4], [25], [39]. The language provides only descriptions to execute reversible computation in before clauses. Operations that should be described in after clauses are generated from descriptions in before clauses. Consequently, left invertibility is ensured by construction. Error-prone descriptions in after clauses are unnecessary.

Finally, we confirm that Tascell programs to solve typical search problems such as pentomino puzzles, N-queens, and traveling salesman problems can be written in our language.

The remainder of this paper is organized as follows. Section 2 introduces the Tascell language, operational semantics, and extensional equivalence. Section 3 describes several proposed properties such as left invertibility and well-formedness properties for extensional equivalence. In Section 4, a domain-specific language is proposed based on reversible computation. In Section 5, we present experimentally obtained results. In Section 6, we discuss related work to clarify the contributions of this paper. In Section 7, we conclude the paper by identifying future research directions.

2. Tascell

This section introduces Tascell task parallel programming with backtracking-based load balancing, along with its formal definitions.

```
1 task pentomino {
2  in: int k, i0, i1, i2;
3  in: int a[12]; // manage unused pieces
4  in: int b[70]; // the board
5  ...
6 };
7
8 worker int search (int k, int j0, int j1, int j2,
9                     task pentomino *tsk) {
10  do_many (int p: j1, j2) {
11   int ap=tsk->a[p];
12   for (each possible direction d of the piece) {
13    ... local variable definitions ...
14    if (Can the ap-th piece in the d-th direction be placed
15        on tsk->b?);
16    else continue;
17    // construct for specifying undo/redo operations
18    dynamic_wind
19     { // before clause: do/redo operation for dynamic_wind
20       Set the ap-th piece onto the board tsk->b
21       and update tsk->a.
22     }
23     { // body clause
24       kk = the next empty cell;
25       if (no empty cell?) ; // a solution found
26       else {
27        // try the next piece
28        search (kk, j0+1, j0+1, 12, tsk);
29       }
30     }
31     { // after clause: undo operation for dynamic_wind
32       Backtrack, i.e., remove the ap-th piece from tsk->b
33       and restore tsk->a.
34     }
35   }
36 } handles pentomino ...
37
38 return 0;
39 }
```

Fig. 2 Task definition and its search function in the Tascell program for solving pentomino puzzles.

2.1 Example

We introduce the Tascell language intuitively using an example program. The Tascell language is an extension of the C programming language [*1]. **Figure 2** presents an example of a Tascell program [*2] for solving pentomino puzzles, as introduced in Section 1.

Lines 1–6 define the task `pentomino`. Variables and arrays have the attribute `out` or `in`. Variables with the attribute `out` hold values that are used for outputs. The variables and arrays with the attribute `in` are used to find solutions. Variable `k` denotes the next empty cell on the board. Variable `i0` denotes the next piece that is to be put on the board. Variables `i1` and `i2` respectively denote the first and last pieces. It is checked whether the range of pieces can be put on the board.

Array `a` is used to manage unused pieces. There are 12 kinds of pieces. The array `b` denotes the board, which consists of $60 = 6 \cdot 10$ cells with sentinels at the x-axis. The array size is 70.

Lines 8–39 define the worker function `search`.

[*1] The Tascell compiler takes programs written in an extension of the SC-0 language [17], which consists of S-expressions and which has exactly the same semantics as those of the C programming language. We can define an extension of SC-0 as explained in Section 4.2. It is easier to do so than to modify a parser directly for C programs.

[*2] Strictly speaking, although Tascell programs consist of S-expressions, as described above, the example program is written in C-like notation in this paper for readers who are familiar with the standard C notation.

Lines 10–36 specify a loop statement. Tascell has loop statements of two kinds: `for` and `do`. The `for` loop statements are standard loop statements of C, which are executed sequentially by one worker. Therefore, the so-called `doall` and `doacross` statements (cf., Ref. [10]) can be written as `for` loop statements.

`do` loop statements are divisible into multiple iterations, which can be stolen by thief workers. Iterations of `do` loop statements must be defined to be executed in parallel. The results will not be ensured to be the same as those obtained when the iterations are sequentially executed if this is not done.

`do` loop statements consist of `do_two` and `do_many` statements. `do_two` statements have exactly two iterations; `do_many` statements have zero or more iteration.

Lines 18–34 describe a `dynamic_wind` statement, for which semantics are derived from those in Scheme [23]. `dynamic_wind` statements have *before*, *body*, and *after* clauses in Tascell. If no work stealing occurs, then statements in before, body, and after clauses, are executed sequentially.

Statements in before and after clauses are executed atomically; that is, while a victim worker executes them, thief workers cannot steal tasks from the victim worker. While a victim worker executes statements in a body clause, thief workers can steal tasks from the victim worker. In a straightforward strategy, the victim worker would create a task at the current state. Also, the task would be very small. In backtracking-based load balancing, the victim worker backtracks to a former state at which large tasks can be created. A procedure for backtracking to a former state is described in the after clause. In fact, `do` statements can have multiple and nested `dynamic_wind` statements. The victim worker backtracks to a former state by executing (multiple) statements in (multiple) after clauses and creates tasks. The thief worker steals some tasks. The victim worker reverts to the original state by executing the statements in the before clause.

In the example, lines 19–22 specify statements in the before clause that choose a piece and update the board. Lines 23–30 specify statements in the body clause that judge whether a solution exists and if not, continue to search for solutions. Lines 31–34 specify statements in the after clause that restore the board.

A problem with the Tascell language is that the user is responsible for keeping the consistency between statements of before, body, and after clauses. If the consistency is destroyed, then sequential and concurrent behaviors of the program do not extensionally coincide, that is, a search tree processed by one worker does not coincide with a search tree processed by multiple workers.

Line 36 describes a detailed procedure for creating tasks. It is omitted from the discussion presented in this paper.

Finally, we note that we specifically examine search trees only. We regard counting of the number of solutions, printing solutions to the standard output, etc. as side effects in this paper.

2.2 Syntax

We define a sub-language of the Tascell language specific to search problems. **Figure 3** presents Tascell statements. Statements consist of user statements and other statements that are useful for describing operational semantics. The user statements

(statements) $s ::= u \mid s;s \mid \texttt{enddo} \mid \texttt{pop} \mid \texttt{finish}\ j$

(user statements) $u ::= r \mid u;u \mid \texttt{if}(e)\texttt{then}\{u\}\texttt{else}\{u\}$
$\mid \texttt{for}(x,es)\{u\} \mid \texttt{do}(x,es)\{u\}$
$\mid \texttt{dynamic-wind}\ (r,u,r) \mid \texttt{mark} \mid \texttt{unmarked}\{u\}$

(raw statements) $r ::= \texttt{skip} \mid \texttt{assert}(e) \mid x = e \mid x[e] = e$
$\mid r;r \mid \texttt{if}(e)\texttt{then}\{r\}\texttt{else}\{r\}$

(expressions) $e ::= c \mid x \mid x[e] \mid e \otimes e$

(expression lists) $es ::= \texttt{nil} \mid e :: es$

(binary operators) $\otimes ::= + \mid - \mid \hat{} \mid \& \mid \mid \mid == \mid != \mid < \mid >$

Fig. 3 Sub-language of the Tascell language.

are written by Tascell users.

Expressions consist of constants c, variables x, arrays $x[e]$, and expressions composed by binary operators $e_0 \otimes e_1$. Constant c ranges over non-negative integers. Variables and array elements are initially zero-cleared. Arrays are indexed by integers starting from zero. The binary operators addition, subtraction, bitwise exclusive OR operation (that is, `0^0 = 0`, `0^1 = 1`, `1^0 = 1`, and `1^1 = 0`), conjunction, disjunction, equality, inequality, less than, and greater than are defined in a standard manner. The truth value "true" is represented by any non-zero integer; the truth value "false" is represented by zero.

Statement `skip` denotes no operation. The statement `assert`(e) means `if(`e`==0) exit;` written in the C programming language. The statements $x = e$ and $x[e_0] = e_1$ are assignments.

Raw statements consist of no operation, assertions, assignment statements, concatenations of raw statements, and conditional statements.

User statements consist of raw statements, concatenations of user statements, conditional statements, `for` loop statements, `do` loop statements, `dynamic-wind` statements, `mark` statements, and `unmarked` statements. Strictly speaking, we must distinguish constructors for concatenations of statements and conditional statements in the definition of user statements from those in the definition of raw statements. However, we do not use different notation because meaning is readily apparent from the context.

The difference between `for` and `do` loop statements is explained in Section 2.1. Loop variable x ranges over the expression list es. Symbol `nil` denotes the empty list and `_::_` denotes the list constructor "cons". We assume the so-called variable convention that loop variables are fresh using alpha-conversions. Although this assumption can also be satisfied by introducing the local variable notion in statements, we implicitly assume the convention.

The statement `dynamic-wind` $(r_{before}, u, r_{after})$ denotes the `dynamic-wind` statement consisting of r_{before}, u, and r_{after} in the statements in the before, body, and after clauses. The `mark` statement records the current *node*. Tascell is a domain-specific language for search problems. A node in a search tree is defined by a memory. We assume that a concrete method to make nodes from memories, which is represented as a function $nd(\cdot)$, and which is introduced in Section 2.3, is given externally. The `unmarked` statement `unmarked`$\{u\}$ is a conditional statement. If the node which is defined by the memory is not marked, then u is executed.

$$\frac{[\![e]\!]_\sigma \neq 0}{\langle \texttt{assert}(e), \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip}, \sigma\rangle} \qquad \frac{}{\langle x = e, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip}, \sigma[x := [\![e]\!]_\sigma]\rangle}$$

$$\frac{}{\langle x[e_0] = e_1, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip}, \sigma[\langle x, [\![e_0]\!]_\sigma\rangle := [\![e_1]\!]_\sigma]\rangle}$$

$$\frac{\langle r_0, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip}, \sigma'\rangle}{\langle r_0;r_1, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle r_1, \sigma'\rangle} \qquad \frac{\langle r_0, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle r_0', \sigma'\rangle}{\langle r_0;r_1, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle r_0';r_1, \sigma'\rangle}$$

$$\frac{[\![e]\!]_\sigma \neq 0}{\langle \texttt{if}(e)\texttt{then}\{r_0\}\texttt{else}\{r_1\}, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle r_0, \sigma\rangle} \qquad \frac{[\![e]\!]_\sigma = 0}{\langle \texttt{if}(e)\texttt{then}\{r_0\}\texttt{else}\{r_1\}, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle r_1, \sigma\rangle}$$

Fig. 4 Local semantics for raw statements.

$$\frac{\langle s, \sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle s', \sigma'\rangle}{\langle s, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle s', ds, \langle \varsigma, \sigma'\rangle\rangle_j} \qquad \frac{\langle s_0, ds, stt\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{skip}, ds', stt'\rangle_j}{\langle s_0;s_1, ds, stt\rangle_j \rightsquigarrow_{\mathrm{l}} \langle s_1, ds', stt'\rangle_j}$$

$$\frac{\langle s_0, ds, stt\rangle_j \rightsquigarrow_{\mathrm{l}} \langle s_0', ds', stt'\rangle_j}{\langle s_0;s_1, ds, stt\rangle_j \rightsquigarrow_{\mathrm{l}} \langle s_0';s_1, ds', stt'\rangle_j} \qquad \frac{}{\langle \texttt{mark}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{skip}, ds, \langle nd(\sigma) \cup \varsigma, \sigma\rangle\rangle_j}$$

$$\frac{nd(\sigma) \notin \varsigma}{\langle \texttt{unmarked}\{u\}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle u, ds, \langle \varsigma, \sigma\rangle\rangle_j} \qquad \frac{nd(\sigma) \in \varsigma}{\langle \texttt{unmarked}\{u\}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{skip}, ds, \langle \varsigma, \sigma\rangle\rangle_j}$$

$$\frac{}{\langle \texttt{for}(x,\texttt{nil})\{u\}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{skip}, ds, \langle \varsigma, \sigma[x := 0]\rangle\rangle_j} \qquad \frac{}{\langle \texttt{for}(x,e::es)\{u\}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle u;\texttt{for}(x,es)\{u\}, ds, \langle \varsigma, \sigma[x := [\![e]\!]_\sigma]\rangle\rangle_j}$$

$$\frac{[\![e]\!]_\sigma \neq 0}{\langle \texttt{if}(e)\texttt{then}\{u_0\}\texttt{else}\{u_1\}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle u_0, ds, \langle \varsigma, \sigma\rangle\rangle_j} \qquad \frac{[\![e]\!]_\sigma = 0}{\langle \texttt{if}(e)\texttt{then}\{u_0\}\texttt{else}\{u_1\}, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle u_1, ds, \langle \varsigma, \sigma\rangle\rangle_j}$$

$$\frac{}{\langle \texttt{do}(x,es)\{u\}, ds, stt\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{enddo}, \langle \texttt{do}(x,es)\{u\}, [\![e]\!]_\sigma, \texttt{nil}\rangle :: ds, stt\rangle_j}$$

$$\frac{}{\langle \texttt{enddo}, \langle \texttt{do}(x,e::es)\{u\}, \_, rs\rangle :: ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle u;\texttt{enddo}, \langle \texttt{do}(x,es)\{u\}, [\![e]\!]_\sigma, rs\rangle :: ds, \langle \varsigma, \sigma[x := [\![e]\!]_\sigma]\rangle\rangle_j}$$

$$\frac{}{\langle \texttt{enddo}, \langle \texttt{do}(x,\texttt{nil})\{\_\}, \_, \_\rangle :: ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{skip}, ds, \langle \varsigma, \sigma[x := 0]\rangle\rangle_j}$$

$$\frac{\langle r, \sigma\rangle \rightsquigarrow_{\mathrm{r}}^* \langle \texttt{skip}, \sigma'\rangle}{\langle \texttt{pop}, \langle u, val, \langle \_, r\rangle :: rs\rangle :: ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle \texttt{skip}, \langle u, val, rs\rangle :: ds, \langle \varsigma, \sigma'\rangle\rangle_j}$$

$$\frac{\langle r_{before}, \sigma\rangle \rightsquigarrow_{\mathrm{r}}^* \langle \texttt{skip}, \sigma'\rangle}{\langle \texttt{dynamic-wind}\ (r_{before}, u_{body}, r_{after}), \langle \texttt{do}(x,es)\{u\}, val, rs\rangle :: ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_{\mathrm{l}} \langle u_{body};\texttt{pop}, \langle \texttt{do}(x,es)\{u\}, val, \langle r_{before}, r_{after}\rangle :: rs\rangle :: ds, \langle \varsigma, \sigma'\rangle\rangle_j}$$

Fig. 5 Local semantics.

Otherwise, it is no operation.

Non-user statements are not written by Tascell users but are introduced to define operational semantics for Tascell programs. Statement `enddo` denotes the endpoint of a `do` statement. Statement `pop` denotes the pop operation for *raw statement stacks*, which are introduced into Section 2.3. The statement `finish` j returns marked nodes to the victim worker j.

2.3 Local Semantics

We define transitions that are raised by the actions of a worker, which in turn are the executions of statements, as shown in **Figs. 4** and **5**.

Letting R be a relation, we R^* for its reflexive and transitive closure of R.

We define transition systems between *configurations* $\langle s, ds, stt\rangle_j$. Configurations are indexed by the worker identifier j, which ranges over non-negative integers. They consist of statements s, `do` lists ds, and states stt. Transitions are written as $\rightsquigarrow_{\mathrm{l}}$. Specifically, transitions by raw statements are written as $\rightsquigarrow_{\mathrm{r}}$. Configurations are simplified to what are necessary. The place-

holder _ denotes an arbitrary statement, an arbitrary expression, etc. Statements were defined as explained in Section 2.2.

A do list records nested do statements that the worker executes. An element d is $\langle \texttt{do}(x,es)\{u\}, val, rs\rangle$, where a *raw statement stack* rs comprises pairs of raw statements. We also use the notation for lists nil and _::_ to represent raw statement stacks.

States consist of pairs of marked nodes ς and memories σ. Marked nodes ς are a set of nodes that are marked. Memories σ are functions from variables x and pairs $\langle x, i\rangle$ of variables and integers to values *val*. In this paper, values are integers. The node $nd(\sigma)$ is a node made from σ. We assume that the function $nd(\cdot)$ is given externally.

A function $\sigma[x := val]$ is an update function of σ with the substitution $[x := val]$ such that $(\sigma[x := val])(x) = val$ and $(\sigma[x := val])(x') = \sigma(x')$ if $x' \in \mathrm{dom}(\sigma) \setminus \{x\}$, where $\mathrm{dom}(\sigma)$ is the domain of σ. Similarly, we define updates for $\langle x, i\rangle$.

The assertion assert(e) checks if e is satisfied on the current memory. If it is not satisfied, then the computation gets stuck.

Assignment statements, concatenations of statements, conditional statements, and for statements are defined in a standard manner, where $[\![e]\!]_\sigma$ is defined as shown below.

$$[\![c]\!]_\sigma = \overline{c} \qquad [\![x]\!]_\sigma = \sigma(x)$$
$$[\![x[e]]\!]_\sigma = \sigma(\langle x, [\![e]\!]_\sigma\rangle) \qquad [\![e_0 + e_1]\!]_\sigma = [\![e_0]\!]_\sigma + [\![e_1]\!]_\sigma \cdots .$$

Therein, $\overline{c}$ is an integer denoted by c, such as $\overline{1} = 1$.

The mark statements update marked nodes from the current states. The unmarked statements are conditional statements about marked nodes.

The statement is pushed to its do list with nil for its after processing if worker j meets a do statement for which the loop range is not nil to be executed. It confirms the loop range of the head of its do list if the worker meets enddo. The worker executes the loop iteration with the loop variable assigned to e if the loop range is e::es. The do statement at the head is removed if the loop range is nil.

Statement pop is used to define semantics for dynamic-wind statements.

If a worker meets dynamic-wind (r_{before},u,r_{after}), then r_{before} is first executed and r_{after} is pushed to the last d in its do list. Next, u is executed. The head of d is removed by executing pop. This mechanism enables backtracking-based load balancing, as explained in Section 2.4.

Proposition 2.1. *Assuming that* $\langle s, ds, stt\rangle_j \rightsquigarrow_1 \langle s', ds, stt'\rangle_j$, *then, for any* ds', $\langle s, ds', stt\rangle_j \rightsquigarrow_1 \langle s', ds', stt'\rangle_j$ *holds.*

Proof. By induction on $\rightsquigarrow_1$. A point is that any transition which depends on *ds* changes *ds*. □

A relation $\mathcal{R}$ is called *deterministic* if $\langle A_0, A_1\rangle \in \mathcal{R}$ and $\langle A_0, A_2\rangle \in \mathcal{R}$ imply $A_0 = A_2$. A deterministic relation $\rightsquigarrow$ is called *terminating* if no infinite sequence exists with respect to $\rightsquigarrow$.

Proposition 2.2. *The relation* $\rightsquigarrow_1$ *is deterministic.*

Proof. By induction on $\rightsquigarrow_1$. □

Proposition 2.3. *The relation* $\rightsquigarrow_1$ *is terminating.*

Proof. We define a weight function $\|\cdot\|$ inductively as shown below.

$$\|es\| : \quad \|\texttt{nil}\| = 1$$
$$\|e::es\| = 3 \cdot \|es\| + 1$$
$$\|ds\| : \quad \|\texttt{nil}\| = 1$$
$$\|\langle \texttt{do}(x,es)\{u\}, val, rs\rangle :: ds\| = \|1+u\|^{\|es\|} \cdot \|ds\|$$
$$\|s\| : \quad \|\texttt{skip}\| = \|\texttt{enddo}\| = 1$$
$$\|\texttt{assert}(e)\| = \|x = e\| = \|x[e_0] = e_1\| = 2$$
$$\|\texttt{mark}\| = \|\texttt{pop}\| = \|\texttt{finish}\ j\| = 2$$
$$\|\texttt{unmarked}\{u\}\| = 1 + \|u\|$$
$$\|\texttt{if}(e)\texttt{then}\{u_0\}\texttt{else}\{u_1\}\| = \|u_0\| + \|u_1\|$$
$$\|s_0\,;\,s_1\| = \|s_0\| + \|s_1\|$$
$$\|\texttt{for}(x,es)\{u\}\| = 1 + \|es\| \cdot \|u\|$$
$$\|\texttt{do}(x,es)\{u\}\| = 2 \cdot \|1+u\|^{\|es\|}$$
$$\|\texttt{dynamic-wind}\ (r_0,u,r_1)\| = 3 + \|u\|$$

We also define $\|\langle s, ds\rangle\|$ as $\|s\| \cdot \|ds\|$. One can readily confirm that $\langle s, ds, stt\rangle_j \rightsquigarrow_1 \langle s', ds', stt'\rangle_j$ implies $\|\langle s, ds\rangle\| > \|\langle s', ds'\rangle\|$. A point is that loop ranges of do loop statements are heavily weighted. Therefore, the relation $\rightsquigarrow_1$ is terminating. □

Readers who are familiar with proofs for termination might think of using exponentials to define the weight function as an application of Occam's razor. However, we extend the weight function in another proof for termination in Section 2.4.

2.4 Global Semantics

We define transitions $\rightsquigarrow_g$ that are performed by multiple workers that execute statements.

Let N be the number of workers. We use the abbreviation $cfg_j \rightsquigarrow_d cfg'_j$, which describes only a difference, for $\langle\langle cfg_0, \ldots, cfg_j, \ldots, cfg_{N-1}\rangle, \langle cfg_0, \ldots, cfg'_j, \ldots, cfg_{N-1}\rangle\rangle \in \rightsquigarrow_g$. We also use the abbreviation $cfg_j, cfg_k \rightsquigarrow_d cfg'_j, cfg'_k$ for $\langle\langle cfg_0, \ldots, cfg_j, \ldots, cfg_k, \ldots, cfg_{N-1}\rangle, \langle cfg_0, \ldots, cfg'_j, \ldots, cfg'_k, \ldots, cfg_{N-1}\rangle\rangle \in \rightsquigarrow_g$. The transition on N-tuples of configurations is defined as presented in **Fig. 6**.

The state *init* is defined as $\langle \emptyset, \_ \mapsto 0\rangle$, where $\_ \mapsto 0$ denotes the constant function to 0.

Local transitions are global transitions, as shown at (local$_j$).

The transition rule (steal$_{j_0,j_1}$) shows work stealing. Worker j_1 with the initial state requests tasks from worker j_0. Worker j_0 executes after statements to backtrack to a state in which the outermost do statement for which the loop range is more than one is executed, and create tasks where $|es|$ is the length of the list es and after($\langle r_{n,before}, r_{n,after}\rangle\,; \ldots ; \langle r_{k,before}, r_{k,after}\rangle$) denotes $r_{n,after}\,; \ldots ; r_{k,after}$.

$$\frac{\langle s, ds, stt\rangle_j \rightsquigarrow_1 \langle s', ds', stt'\rangle_j}{\langle s, ds, stt\rangle_j \rightsquigarrow_d \langle s', ds', stt'\rangle_j}\ (\text{local}_j) \qquad \frac{}{\langle s, ds, \langle\varsigma_0, \sigma\rangle\rangle_{j_0}, \langle \texttt{finish}\ j_0, \texttt{nil}, \langle\varsigma_1, \_\rangle\rangle_{j_1} \rightsquigarrow_d \langle s, ds, \langle\varsigma_0 \cup \varsigma_1, \sigma\rangle\rangle_{j_0}, \langle \texttt{skip}, \texttt{nil}, init\rangle_{j_1}}\ (\text{merge}_{j_0,j_1})$$

$$\frac{\begin{array}{l} ds \equiv d_n :: \ldots :: d_0 :: \texttt{nil} \quad d_i \equiv \langle \texttt{do}(x_i, es_i)\{u_i\}, val_i, rs_i\rangle \text{ for any } 0 \le i \le n \\ 0 \le k \le n \quad m > 0 \quad es_k \equiv e_m :: \ldots :: e_0 :: \texttt{nil} \quad |es_i| \le 1 \text{ for any } 0 \le i < k \\ \langle x_n = val_n\,; \text{after}(rs_n)\,; \ldots ; x_k = val_k\,; \text{after}(rs_k), \sigma\rangle \rightsquigarrow_r^* \langle \texttt{skip}, \sigma_1\rangle \\ \langle x_k = val_k\,; \text{before}(rs_k)\,; \ldots ; x_n = val_n\,; \text{before}(rs_n), \sigma_1\rangle \rightsquigarrow_r^* \langle \texttt{skip}, \sigma_0\rangle \\ ds_0 \equiv d_n :: \ldots :: \langle \texttt{do}(x_k, e_m :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \texttt{nil})\{u_k\}, val_k, rs_k\rangle :: \ldots :: d_0 :: \texttt{nil} \\ ds_1 \equiv \langle \texttt{do}(x_k, e_{\lceil \frac{m}{2} \rceil - 1} :: \ldots :: e_0 :: \texttt{nil})\{u_k\}, val_k, \texttt{nil}\rangle :: \texttt{nil} \end{array}}{\begin{array}{l} \langle s, ds, \langle\varsigma, \sigma\rangle\rangle_{j_0}, \langle \texttt{skip}, \texttt{nil}, init\rangle_{j_1} \rightsquigarrow_d \\ \langle s, ds_0, \langle\varsigma, \sigma_0\rangle\rangle_{j_0}, \langle \texttt{enddo}\,; \texttt{finish}\, j_0, ds_1, \langle\emptyset, \sigma_1\rangle\rangle_{j_1} \end{array}}\ (\text{steal}_{j_0,j_1})$$

Fig. 6 Global semantics.

Worker j_1 steals the tasks and starts to execute tasks with the state $\langle \varnothing, \sigma_1 \rangle$. Worker j_0 executes before statements to revert to the original state, where before($\langle r_{k,before}, r_{k,after} \rangle$; ...; $\langle r_{n,before}, r_{n,after} \rangle$) denotes $r_{k,before}$; ...; $r_{n,before}$. In fact, state σ_1 is not generally ensured to be the state in which the do statement was formerly executed. The state σ_0 is not generally ensured to be the original state σ. We introduce sufficient conditions to ensure these in Section 3. Worker j_0 has some tasks stolen; its current tasks are ds_0.

The transition rule (merge$_{j_0,j_1}$) shows a merging of states when tasks are completed. Worker j_1 returns ς_1, which is a set of marked nodes. Marked nodes are merged. Worker j_1 prepares to request tasks if necessary.

The relation $\rightsquigarrow_{\mathrm{g}}$ is generally non-deterministic. A relation $\rightsquigarrow$ is called *strongly normalizing* if there exists no infinite sequence with respect to $\rightsquigarrow$. By Proposition 2.3, the deterministic relation $\rightsquigarrow_1$ is strongly normalizing.

Proposition 2.4. *The relation $\rightsquigarrow_{\mathrm{g}}$ is strongly normalizing.*

Proof. We extend the weight function $\|\langle s, ds \rangle\|$ in the proof of Proposition 2.3 to that for the N-tuples of pairs such that

$$\|\langle \langle s_0, ds_0 \rangle, \ldots, \langle s_j, ds_j \rangle, \ldots, \langle s_{N-1}, ds_{N-1} \rangle \rangle\| = \prod \{ \|\langle s_j, ds_j \rangle\| \mid 0 \le j < N \}$$

where $\prod$ is the multiplication operator for sets of integers.

It is readily apparent that $\rightsquigarrow_{\mathrm{g}}$ transitions by (local$_j$) and (merge$_{j_0,j_1}$) are decreasing. Next, we can consider the transition $\langle s, ds, \_ \rangle_{j_0}, \langle \mathtt{skip}, \mathtt{nil}, init \rangle_{j_1} \rightsquigarrow_{\mathrm{d}} \langle s, ds_0, \_ \rangle_{j_0}, \langle \mathtt{enddo}; \mathtt{finish}\ j_0, ds_1, \_ \rangle_{j_1}$ by (steal$_{j_0,j_1}$).

A point is that the divisions of loop ranges of do loop statements reduce weights, i.e., loop ranges that consist of elements of a loop range are weighted more lightly than the original loop range. Because

$$n^{\|e_m :: \ldots :: e_0 :: \mathtt{nil}\|} > 3 \cdot n^{\|e_m :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \mathtt{nil}\| + \|e_{\lceil \frac{m}{2} \rceil - 1} :: \ldots :: e_0 :: \mathtt{nil}\|}$$

holds for any $n \ge 2$ and $m > 0$,

$$\begin{aligned} &\|\langle s, ds \rangle, \langle \mathtt{skip}, \mathtt{nil} \rangle\| \\ &= \|s\| \cdot \|ds\| \\ &> \|s\| \cdot \|ds\| / \|d_k\| \cdot (1 + \|u_k\|)^{\|e_m :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \mathtt{nil}\|} \cdot \\ &\quad 3 \cdot (1 + \|u_k\|)^{\|e_{\lceil \frac{m}{2} \rceil - 1} :: \ldots :: e_0 :: \mathtt{nil}\|} \\ &= \|\langle s, ds_0 \rangle, \langle \mathtt{enddo}; \mathtt{finish}\ j_0, ds_1 \rangle\| \end{aligned}$$

holds. Therefore, the relation $\rightsquigarrow_{\mathrm{g}}$ is strongly normalizing. □

Because the relation $\rightsquigarrow_{\mathrm{g}}$ is non-deterministic, a Tascell program can generally return multiple states. We expect that a correct Tascell program returns multiple answers that can be regarded as extensionally equivalent. In Sections 3, we define some properties for Tascell programs. We also demonstrate that Tascell programs which satisfy the nice properties return states that are extensionally equivalent.

2.5 Extensional equivalence

We define an extensional equivalence on configurations. Because our targets are search problems, the extensional equivalence respects the equality between solutions of search problems.

Some notions of abstract rewriting theory must be recalled [38]. $\langle \mathfrak{A}, \rightsquigarrow \rangle$ is called an *abstract rewriting system* if $\mathfrak{A}$ is a set and $\rightsquigarrow$ is a relation on $\mathfrak{A}$. Letting $\sim$ be an equivalence relation on $\mathfrak{A}$, then one can write $\approx$ for $(\sim \cup \rightsquigarrow \cup \{ \langle b, a \rangle \mid \langle a, b \rangle \in \rightsquigarrow \})^*$. $A_0, A_1 \in \mathfrak{A}$ are called *joinable modulo* $\sim$ if there exist $A_2, A_3 \in \mathfrak{A}$ such that $A_0 \rightsquigarrow^* A_2$, $A_1 \rightsquigarrow^* A_3$, and $A_2 \sim A_3$. $\langle \mathfrak{A}, \rightsquigarrow \rangle$ is called *locally confluent modulo* $\sim$ if for any $A_0, A_1, A_2 \in \mathfrak{A}$, $A_0 \rightsquigarrow A_1$ and $A_0 \rightsquigarrow A_2$ imply that A_1 and A_2 are joinable. $\langle \mathfrak{A}, \rightsquigarrow \rangle$ is called *locally coherent modulo* $\sim$ if for any $A_0, A_1, A_2 \in \mathfrak{A}$, $A_0 \rightsquigarrow A_1$ and $A_0 \sim A_2$ imply that A_1 and A_2 are joinable. $\langle \mathfrak{A}, \rightsquigarrow \rangle$ is called *Church–Rosser modulo* $\sim$ if for any $A_0, A_1 \in \mathfrak{A}$, $A_0 \approx A_1$ implies A_0 and A_1 are joinable.

We define that an N-tuple consisting of $\{ cfg_{0,j} \mid 0 \le < N \}$ is related to an N-tuple consisting of $\{ cfg_{1,j} \mid 0 \le j < N \}$ by $\sim_{\mathrm{g}}$ if there exists a permutation f (i.e., a bijection on $\{ j \mid 0 \le j < N \}$) such that the N-tuple consisting of $\{ cfg_{0,j} \mid 0 \le j < N \}$ is the same as the N-tuple consisting of $\{ f(cfg_{1,j}) \mid 0 \le j < N \}$, where $f(\langle s, ds, stt \rangle_j) = \langle f(s), ds, stt \rangle_{f(j)}$ and

$$\begin{aligned} &f(u) = u && f(\mathtt{enddo}) = \mathtt{enddo} \\ &f(\mathtt{pop}) = \mathtt{pop} && f(\mathtt{finish}\ j) = \mathtt{finish}\ f(j) \\ &f(s_0; s_1) = f(s_0); f(s_1)\ . \end{aligned}$$

Intuitively, $\sim_{\mathrm{g}}$ is the relation up to differences between worker identifiers.

We define that $\mathfrak{S}$ is a set of N-tuples of configurations that are *extensionally equivalent* if $\langle \mathfrak{S}, \rightsquigarrow_{\mathrm{g}} \rangle$ is Church–Rosser modulo $\sim_{\mathrm{g}}$. It means that every concurrent behavior with work stealing is equal to its sequential behavior with no work stealing.

3. Properties for Extensional Equivalence

This section introduces several properties of Tascell programs, which are used to ensure consistent behaviors of Tascell programs.

First, a statement $\langle s, ds \rangle$ is called *parallelizable* if for any

$$\begin{aligned} &\langle \mathtt{enddo}, \langle \mathtt{do}(x, e :: es)\{u\}, \_, rs \rangle :: ds', \langle \varsigma, \sigma \rangle \rangle_j \\ &\rightsquigarrow_1^* \langle \mathtt{enddo}, \langle \mathtt{do}(x, es)\{u\}, [\![e]\!]_\sigma, rs \rangle :: ds', \langle \varsigma', \sigma' \rangle \rangle_j \text{ or} \\ &\langle \mathtt{enddo}; s', \langle \mathtt{do}(x, e :: es)\{u\}, \_, rs \rangle :: ds', \langle \varsigma, \sigma \rangle \rangle_j \\ &\rightsquigarrow_1^* \langle \mathtt{enddo}; s', \langle \mathtt{do}(x, es)\{u\}, [\![e]\!]_\sigma, rs \rangle :: ds', \langle \varsigma', \sigma' \rangle \rangle_j \end{aligned}$$

that occurs in the sequence of the transitions from $\langle s, ds, init \rangle_j$, $\sigma[x := [\![e]\!]_\sigma] = \sigma'$ holds. The parallelizability of a statement enables the parallel processing of do loop statements in the statement because no iterations change states except values of loop variables at its entry and exit. Programs that have no parallelizability are beyond the scope of Tascell.

Next, we propose the backtrackability notion, which is used to ensure that backtrackings do not change states. A triple $\langle s, ds, \sigma \rangle$ is called *backtrackable* if for any σ_1 and $0 \le i \le n$, $\langle x_n = val_n; \mathrm{after}(rs_n); \ldots; x_i = val_i; \mathrm{after}(rs_i), \sigma \rangle \rightsquigarrow_{\mathrm{r}}^* \langle \mathtt{skip}, \sigma_1 \rangle$ implies that $\langle x_i = val_i; \mathrm{before}(rs_i); \ldots; x_n = val_n; \mathrm{before}(rs_n), \sigma_1 \rangle \rightsquigarrow_{\mathrm{r}}^* \langle \mathtt{skip}, \sigma \rangle$ where $ds \equiv d_n :: \ldots :: d_i :: \ldots :: d_0 :: \mathtt{nil}$ and $d_i \equiv \langle \mathtt{do}(x_i, es_i)\{u_i\}, val_i, rs_i \rangle$ $(0 \le i \le n)$. We note that $\langle s, \mathtt{nil}, \sigma \rangle$ is backtrackable by definition for any s and σ.

A pair $\langle r_{before}, r_{after}\rangle$ is called *left invertible* if $\langle r_{before}\,;\, r_{after}, \sigma\rangle \rightsquigarrow_r^* \langle \mathtt{skip}, \sigma\rangle$ for any σ. A statement s is called *left invertible* if $\langle r_{before}, r_{after}\rangle$ is left invertible for any `dynamic-wind` $(r_{before}, u, r_{after})$ that occurs in s.

Proposition 3.1. *Let $\langle r_{before}, r_{after}\rangle$ be left invertible. If $\langle r_{before}, \sigma\rangle \rightsquigarrow_r^* \langle \mathtt{skip}, \sigma'\rangle$ holds, then $\langle r_{after}\,;\, r_{before}, \sigma'\rangle \rightsquigarrow_r^* \langle \mathtt{skip}, \sigma'\rangle$ holds.*

Proof. By left invertibility and the assumption, $\langle r_{before}\,;\, r_{after}\,;\, r_{before}, \sigma\rangle \rightsquigarrow_r^* \langle \mathtt{skip}, \sigma'\rangle$ holds. By determinacy of $\rightsquigarrow_r$ and the assumption, $\langle r_{after}\,;\, r_{before}, \sigma'\rangle \rightsquigarrow_r^* \langle \mathtt{skip}, \sigma'\rangle$ holds. □

A pair $\langle s, ds\rangle$ is called *well-formed* if

- there exists w such that $\langle s, ds\rangle \equiv \langle w, \mathtt{nil}\rangle$ holds or there exists p such that $s \equiv p$ holds,
- for any $\mathtt{do}(x, es)\{u\}$ that occurs in ds, there exists t such that $u \equiv t$, and
- for any $\mathtt{do}(x, es)\{t\}$ that occurs in $\langle s, ds\rangle$, t has no assignment to x

where w, p, and t are defined as shown below.

$$
\begin{aligned}
w &::= r \mid p \mid w\,;w\\
p &::= t \mid p\,;p \mid \mathtt{enddo} \mid \mathtt{pop} \mid \mathtt{finish}\ j\\
t &::= \mathtt{skip} \mid \mathtt{assert}(e) \mid t\,;t \mid \mathtt{if}(e)\mathtt{then}\{t\}\mathtt{else}\{t\}\\
&\quad \mid \mathtt{for}(x, es)\{t\} \mid \mathtt{do}(x, es)\{t\}\\
&\quad \mid \mathtt{dynamic\text{-}wind}\ (r, t, r) \mid \mathtt{mark} \mid \mathtt{unmarked}\{t\}\ .
\end{aligned}
$$

Intuitively, well-formed pairs of statements and `do` lists are generated by statements that have no assignment statement except at top levels, in before or after clauses of `dynamic-wind`, and which has no assignment to loop variables in `do` loop statements. Therefore, the following holds:

Proposition 3.2. *Let $\langle s, ds\rangle$ be well-formed, where $ds \not\equiv \mathtt{nil}$. If $\langle s, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_l \langle s', ds, \langle \varsigma', \sigma'\rangle\rangle_j$ holds, then σ and σ' coincide except the values on the loop variables in s.*

Proof. Statements that do not update `do`-lists cannot update memories except the values on the loop variables. □

Left invertibility and well-formedness can easily be checked statically because they are preserved by transitions as follows:

Proposition 3.3. *1) Left invertibility is preserved by $\rightsquigarrow_l$,*
2) left invertibility is preserved by $\rightsquigarrow_g$,
3) well-formedness is preserved by $\rightsquigarrow_l$, and
4) well-formedness is preserved by $\rightsquigarrow_g$.

Proof. The former holds because $\rightsquigarrow_l$ and $\rightsquigarrow_g$ raise no new `dynamic-wind` statements. The latter immediately holds by definition. □

Left invertibility and well-formedness preserve backtrackability as follows.

Proposition 3.4. *Assuming that $\langle s, ds, \sigma\rangle$ is backtrackable, s is left invertible, and $\langle s, ds\rangle$ is well-formed, then if $\langle s, ds, \langle \varsigma, \sigma\rangle\rangle_j \rightsquigarrow_l \langle s', ds', \langle \varsigma', \sigma'\rangle\rangle_j$ holds, $\langle s', ds', \sigma'\rangle$ is backtrackable.*

Proof. By case analysis of s. □

Proposition 3.5. *Assuming that $\langle s_j, ds_j, \sigma_j\rangle$ is backtrackable, s_j is left invertible, and $\langle s_j, ds_j\rangle$ is well-formed for any $0 \le j < N$, then if $\langle s_0, ds_0, \langle \varsigma_0, \sigma_0\rangle\rangle_0, \ldots, \langle s_j, ds_j, \langle \varsigma_j, \sigma_j\rangle\rangle_j, \ldots, \langle s_{N-1}, ds_{N-1}, \langle \varsigma_{N-1}, \sigma_{N-1}\rangle_{N-1}\rangle \rightsquigarrow_g \langle s'_0, ds'_0, \langle \varsigma'_0, \sigma'_0\rangle\rangle_0, \ldots, \langle s'_j, ds'_j, \langle \varsigma'_j, \sigma'_j\rangle\rangle_j, \ldots, \langle s'_{N-1}, ds'_{N-1}, \langle \varsigma'_{N-1}, \sigma'_{N-1}\rangle\rangle_{N-1}$ holds, then $\langle s'_j, ds'_j, \sigma'_j\rangle$ is backtrackable for any $0 \le j < N$.*

Proof. This proposition immediately holds by the definition of $\rightsquigarrow_g$ and Proposition 3.4. □

Left invertibility and well-formedness imply parallelizability as follows.

Proposition 3.6. *Assume that s is left invertible and $\langle s, ds\rangle$ is well-formed. Then, $\langle s, ds\rangle$ is parallelizable.*

Proof. Let us take a transition:

$$
\begin{aligned}
&\langle \mathtt{enddo}, \langle \mathtt{do}(x, e::es)\{t\}, \_, rs\rangle::ds', \langle \varsigma, \sigma\rangle\rangle_j\\
&\rightsquigarrow_l^* \langle \mathtt{enddo}, \langle \mathtt{do}(x, es)\{t\}, [\![e]\!]_\sigma, rs\rangle::ds', \langle \varsigma', \sigma'\rangle\rangle_j \text{ or}\\
&\langle \mathtt{enddo}\,;p, \langle \mathtt{do}(x, e::es)\{t\}, \_, rs\rangle::ds', \langle \varsigma, \sigma\rangle\rangle_j\\
&\rightsquigarrow_l^* \langle \mathtt{enddo}\,;p, \langle \mathtt{do}(x, es)\{t\}, [\![e]\!]_\sigma, rs\rangle::ds', \langle \varsigma', \sigma'\rangle\rangle_j
\end{aligned}
$$

that occurs in the sequence of the transitions from $\langle s, ds, init\rangle_j$. It is noteworthy that well-formedness and Proposition 3.4 ensures that we can write t (instead of u) for the body of the loop statement. Because t has no assignment, left invertibility ensures coincidence of σ and σ' except values on the loop variable x. □

At a glance, the left invertibility and well-formedness properties seem too strong. However, it is noteworthy that typical search problems such as pentomino puzzles, N-queens, and traveling salesman problems can be solved by Tascell programs that are left invertible and well-formed, as explained in Section 5.

An objective is to show that we can obtain unique set of marked nodes for any execution trace of a Tascell program with backtracking.

The following is a well-known theorem about Church–Rosser modulo equivalence.

Theorem 3.7 (Huet [20])**.** *Assuming that $\langle \mathfrak{A}, \rightsquigarrow\rangle$ is strongly normalizing, locally confluent modulo $\sim$, and locally coherent modulo $\sim$, then, $\langle \mathfrak{A}, \rightsquigarrow\rangle$ is Church–Rosser modulo $\sim$.*

Let $\mathfrak{U}$ be the set of N-tuples of configurations that are reachable from elements in

$$\{\, \langle u, \mathtt{nil}, init\rangle_0 \mid u \text{ is left invertible and } \langle u, \mathtt{nil}\rangle \text{ is well-formed} \,\}.$$

As seen in Fig. 5, we adopt the integer 0 as the default value for loop variables. Because it is tedious to distinguish loop variables from non-loop variables on states that corresponds to the variable convention for loop statements introduced in Section 2.2, we consider abstract rewriting systems consisting of configurations that are reachable from the state *init* only. Also, we consider abstract rewriting systems consisting of configurations that are reachable from the empty `do` list `nil` because non-empty `do` list may change the initial state.

Proposition 3.8. *Any triple $\langle s, ds, \sigma\rangle$ such that $\langle s, ds, \langle \varsigma, \sigma\rangle\rangle_j$ belongs to $\mathfrak{U}$ for some ς and $0 \le j < N$, is backtrackable.*

Proof. By Proposition 3.5. We note that $\langle u, \mathtt{nil}, \_ \mapsto 0\rangle$ is backtrackable for any u. □

Lemma 3.9. *$\langle \mathfrak{U}, \rightsquigarrow_g\rangle$ is locally confluent modulo $\sim_g$.*

Proof. We show the lemma by case analysis of pairs of $\rightsquigarrow_g$-transition rules. A key case is that an N-tuple of configurations transits to two distinct N-tuples by (local_{j_0}) and (steal_{j_0,j_1}) where j_0 and j_1 are distinct. Let

$$\langle s, ds, \langle \varsigma, \sigma\rangle\rangle_{j_0} \rightsquigarrow_d \langle s', ds', \langle \varsigma', \sigma'\rangle\rangle_{j_0}$$

$\langle s, ds, \langle \varsigma, \sigma\rangle\rangle_{j_0}, \langle \mathtt{skip}, \mathtt{nil}, \mathit{init}\rangle_{j_1} \rightsquigarrow_{\mathrm{d}}$
$\langle s, ds_0, \langle \varsigma, \sigma\rangle\rangle_{j_0}, \langle \mathtt{enddo};\mathtt{finish}\ j_0, ds_1, \langle \varnothing, \sigma_1\rangle\rangle_{j_1}$

be the transitions by (local$_{j_0}$) and (steal$_{j_0,j_1}$), respectively, where

$ds \equiv d_n :: \ldots :: d_0 :: \mathtt{nil}$
$d_i \equiv \langle \mathtt{do}(x_i, es_i)\{t_i\}, val_i, rs_i\rangle$ for any $0 \le i \le n \qquad 0 \le k \le n$
$m > 0 \quad es_k \equiv e_m :: \ldots :: e_0 :: \mathtt{nil} \quad |es_i| \le 1$ for any $0 \le i < k$
$\langle x_n = val_n; \mathrm{after}(rs_n); \ldots; x_k = val_k; \mathrm{after}(rs_k), \sigma\rangle \rightsquigarrow_{\mathrm{r}}{}^{*} \langle \mathtt{skip}, \sigma_1\rangle$
$\langle x_k = val_k; \mathrm{before}(rs_k); \ldots; x_n = val_n; \mathrm{before}(rs_n), \sigma_1\rangle$
$\rightsquigarrow_{\mathrm{r}}{}^{*} \langle \mathtt{skip}, \sigma\rangle$
$ds_0 \equiv d_n :: \ldots :: \langle \mathtt{do}(x_k, e_m :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \mathtt{nil})\{t_k\}, val_k, rs_k\rangle$
$:: \ldots :: d_0 :: \mathtt{nil}$
$ds_1 \equiv \langle \mathtt{do}(x_k, e_{\lceil \frac{m}{2} \rceil - 1} :: \ldots :: e_0 :: \mathtt{nil})\{t_k\}, val_k, \mathtt{nil}\rangle :: \mathtt{nil}.$

It is noteworthy that backtrackability ensures that the state reverts to σ after backtracking. We do case analysis of ds'.
i) Assume $ds' \equiv \langle \mathtt{do}(x_n, es_n)\{u_n\}, val_n, rs'_n\rangle :: d_{n-1} :: \ldots :: \mathtt{nil}$ where there exist r_{before} and r_{after} such that $rs_n \equiv \langle r_{before}, r_{after}\rangle :: rs'_n$. Then, s is a statement beginning with pop, $\langle r_{after}, \sigma\rangle \rightsquigarrow_{\mathrm{r}}{}^{*} \langle \mathtt{skip}, \sigma'\rangle$, and $\varsigma = \varsigma'$ hold.

We also assume $k < n$. Let ds'_0 be $\langle \mathtt{do}(x_n, es_n)\{u_n\}, val_n, rs'_n\rangle :: \ldots :: \langle \mathtt{do}(x_k, e_m :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \mathtt{nil})\{t_k\}, val_k, rs_k\rangle :: \ldots :: d_0 :: \mathtt{nil}$. The two N-tuples can transit to

$\langle s', ds'_0, \langle \varsigma, \sigma'\rangle\rangle_{j_0}, \langle \mathtt{enddo};\mathtt{finish}\ j_0, ds_1, \langle \varnothing, \sigma_1\rangle\rangle_{j_1}.$

Cases of $k = n$ are similar.
ii) Assume $ds' \equiv \langle \mathtt{do}(x_n, es'_n)\{t_n\}, val_n, rs_n\rangle :: d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}$ where $es_n \equiv e :: es'_n$. Then, s is a statement beginning with enddo, $rs_n \equiv \mathtt{nil}$, $\sigma' = \sigma[x_n \coloneqq [\![e]\!]_\sigma]$, and $\varsigma = \varsigma'$.
a) Assume that $k < n$. Then, the two N-tuples can transit to

$\langle s', \langle \mathtt{do}(x_n, es'_n)\{t_n\}, val_n, rs_n\rangle :: \ldots$
$:: \langle \mathtt{do}(x_k, e_m :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \mathtt{nil})\{t_k\}, val_k, rs_k\rangle :: d_{n-1} :: \ldots$
$:: d_0 :: \mathtt{nil}, \langle \varsigma', \sigma\rangle\rangle_{j_0}, \langle \mathtt{enddo};\mathtt{finish}\ j_0, ds_1, \langle \varnothing, \sigma_1\rangle\rangle_{j_1}.$

b) Assume that $k = n$ and m is an even integer. Then, the two N-tuples can transit to

$\langle s', \langle \mathtt{do}(x_k, e_{m-1} :: \ldots :: e_{\lceil \frac{m}{2} \rceil} :: \mathtt{nil})\{t_k\}, val_k, rs_k\rangle :: d_{n-1} :: \ldots$
$:: d_0 :: \mathtt{nil}, \langle \varsigma', \sigma\rangle\rangle_{j_0}, \langle \mathtt{enddo};\mathtt{finish}\ j_0, ds_1, \langle \varnothing, \sigma_1\rangle\rangle_{j_1}.$

c) Assume that $k = n$ and $m = 1$. Then, there exists s'' such that $s' \equiv t_n; \mathtt{enddo}; s''$. Also, $ds' \equiv \langle \mathtt{do}(x_n, e' :: \mathtt{nil})\{t_n\}, val_n, rs_n\rangle :: d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}$, $\varsigma = \varsigma'$ hold where $es_n \equiv e :: e' :: \mathtt{nil}$. Since $rs_n \equiv \mathtt{nil}$ holds, $\sigma = \sigma_1$ holds.

By Propositions 2.3, there exist ς_0 and ς_1 such that

$\langle s, ds_0, \langle \varsigma, \sigma\rangle\rangle_{j_0}, \langle \mathtt{enddo};\mathtt{finish}\ j_0, ds_1, \langle \varnothing, \sigma\rangle\rangle_{j_1}$
$\rightsquigarrow_{\mathrm{d}}{}^{*} \langle s'', d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}, \langle \varsigma_0, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_0},$
$\langle \mathtt{finish}\ j_0, \mathtt{nil}, \langle \varsigma_1, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_1}$

holds using (local$_{j_0}$) and (local$_{j_1}$). By Proposition 3.6, the memories after the enddos are executed coincide with σ except values on x_n. Therefore, using (local$_{j_0}$),

$\langle \mathtt{enddo}; t_n; \mathtt{enddo}; s'', ds, \langle \varsigma, \sigma\rangle\rangle_{j_0}$
$\rightsquigarrow_{1}{}^{*} \langle \mathtt{enddo}; s'', ds', \langle \varsigma_0, \sigma[x_n \coloneqq [\![e]\!]_\sigma]\rangle\rangle_{j_0}$
$\rightsquigarrow_{1}{}^{*} \langle \mathtt{enddo}; s'', \langle \mathtt{do}(x_n, \mathtt{nil})\{t_n\}, val_n, rs_n\rangle :: d_{n-1} :: \ldots$
$:: d_0 :: \mathtt{nil}, \langle \varsigma_0 \cup \varsigma_1, \sigma[x_n \coloneqq [\![e']\!]_\sigma]\rangle\rangle_{j_0}$

holds. It is noteworthy that the obtained marked nodes ς_0 and ς_1 are invariant to worker identifiers since $\sigma = \sigma_1$ holds. The two N-tuples can transit to

$\langle s'', d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}, \langle \varsigma_0 \cup \varsigma_1, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_0},$
$\langle \mathtt{skip}, \mathtt{nil}, \mathit{init}\rangle_{j_1}.$

d) Assume that $k = n$ and m is an odd integer more than 1. Since $rs_n \equiv \mathtt{nil}$ holds, $\sigma = \sigma_1$ holds. Similar to the case (c), there exist s'', ς_0, and ς_1 such that

$\langle s, ds_0, \langle \varsigma, \sigma\rangle\rangle_{j_0}, \langle \mathtt{enddo};\mathtt{finish}\ j_0, ds_1, \langle \varnothing, \sigma\rangle\rangle_{j_1}$
$\rightsquigarrow_{\mathrm{d}}{}^{*} \langle s'', d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}, \langle \varsigma_0, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_0},$
$\langle \mathtt{finish}\ j_0, \mathtt{nil}, \langle \varsigma_1, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_1}.$

Similar to the case (c), we also obtain

$\langle s', ds', \langle \varsigma, \sigma\rangle\rangle_{j_0}$
$\rightsquigarrow_{1}{}^{*} \langle s'', d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}, \langle \varsigma_0 \cup \varsigma_1, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_0}.$

Therefore, the two N-tuples can transit to

$\langle s'', d_{n-1} :: \ldots :: d_0 :: \mathtt{nil}, \langle \varsigma_0 \cup \varsigma_1, \sigma[x_n \coloneqq 0]\rangle\rangle_{j_0},$
$\langle \mathtt{skip}, \mathtt{nil}, \mathit{init}\rangle_{j_1}.$

The other cases are also similar. □

Sufficient conditions for Church–Rosser modulo equivalence have been energetically studied (e.g., Refs. [2], [16], [22]) because the original theorem by Huet cannot be as-is applied to various abstract rewriting systems as discussed in some literatures (cf., Section 7.7 in Ref. [16]). A reason is that Theorem 3.7 assumes local coherence modulo equivalence, which is often difficult to be checked. However, we can directly use Theorem 3.7 because local coherence modulo $\sim_{\mathrm{g}}$ is easy to be checked as follows:

Lemma 3.10. $\langle \mathfrak{U}, \rightsquigarrow_{\mathrm{g}}\rangle$ *is locally coherent modulo* $\sim_{\mathrm{g}}$.
Proof. This lemma is derived from two facts: the transition rules in **Figure 4**–6 are defined to be polymorphic to worker identifiers and $\sim_{\mathrm{g}}$ is the relation up to the differences between worker identifiers. □

Theorem 3.11. $\langle \mathfrak{U}, \rightsquigarrow_{\mathrm{g}}\rangle$ *is Church–Rosser modulo* $\sim_{\mathrm{g}}$.
Proof. This theorem is derived from Proposition 2.4, Lemmas 3.9 and 3.10, and Theorem 3.7. □

4. A Reversible Computation Programming Language: RTascell

This section presents design and implementation of a programming language based on *reversible computation* [4], [25], [39]. Programs written in the language are translated into left invertible and well-formed statements in the Tascell language.

$$
\begin{aligned}
S &::= r \mid U \mid S\,;S \mid \texttt{enddo} \mid \texttt{pop} \mid \texttt{finish}\ j \\
U &::= U\,;U \mid \texttt{if}(e)\texttt{then}\{U\}\texttt{else}\{U\} \mid \texttt{for}(x,es)\{U\} \mid \texttt{do}(x,es)\{U\} \\
&\quad \mid \texttt{r-dynamic-wind}(R,U) \mid \texttt{mark} \mid \texttt{unmarked}\{U\} \\
R &::= \texttt{skip} \mid \texttt{assert}(e) \mid x \odot\!= e \mid x[e] \odot\!= e \\
&\quad \mid R\,;R \mid \texttt{r-if}(e)\texttt{then}\{R\}\texttt{else}\{R\}\texttt{fi}(e) \\
&\qquad\qquad \text{where a syntactic rule } (*) \text{ is assumed} \\
\odot &::= + \mid - \mid \hat{}
\end{aligned}
$$

Fig. 7 The RTascell language.

$$
\frac{[\![e]\!]_\sigma \neq 0}{\langle \texttt{assert}(e),\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma\rangle} \qquad \frac{}{\langle x \odot\!= e,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma[x := [\![x \odot e]\!]_\sigma]\rangle}
$$

$$
\frac{}{\langle x[e_0] \odot\!= e_1,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma[\langle x,[\![e_0]\!]_\sigma\rangle := [\![x[e_0] \odot e_1]\!]_\sigma]\rangle}
$$

$$
\frac{\langle R_0,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma'\rangle}{\langle R_0\,;R_1,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle R_1,\sigma'\rangle} \qquad \frac{\langle R_0,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle R_0',\sigma'\rangle}{\langle R_0\,;R_1,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle R_0'\,;R_1,\sigma'\rangle}
$$

$$
\frac{[\![e_0]\!]_\sigma \neq 0}{\langle \texttt{r-if}(e_0)\texttt{then}\{R_0\}\texttt{else}\{R_1\}\texttt{fi}(e_1),\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle R_0\,;\texttt{assert}(e_1),\sigma\rangle}
$$

$$
\frac{[\![e_0]\!]_\sigma = 0}{\langle \texttt{r-if}(e)\texttt{then}\{R_0\}\texttt{else}\{R_1\}\texttt{fi}(e_1),\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle R_1\,;\texttt{assert}(e_1\texttt{==}\texttt{0}),\sigma\rangle}
$$

Fig. 8 Local semantics for reversible raw statements.

4.1 Design

A key idea to ensure left invertibility is to restrict statements in before clauses to *reversible* statements such as compound assignments x += e, which is $x = x+e$. We adopt a syntactic rule that guarantees the reversibility of assignments that is adopted by Refs. [44] and [43]. For any assignment to x, there exists no x on the right-hand side. Similarly, for any assignment $x[e] \odot\!= e'$, there exists no x in the expressions e and e'. We also introduce reversible conditional statements, which are fixed-form conditional statements with assertions in a standard manner of reversible computation.

We formally define a reversible computation programming language called *RTascell*, as shown in **Fig. 7**. Symbols r, x, e, and es are the same as those in Fig. 3. Operators $\odot$ are either +, -, or ^.

First, reversible raw statements consist of no operation, assertions, reversible assignment statements, concatenations of reversible raw statements, and reversible conditional statements. As described above, we assume the syntactic rule $(*)$.

Operational semantics for reversible raw statements are naturally induced as shown in **Fig. 8**. We write $\rightsquigarrow_{\mathrm{r}}$ for transitions by reversible raw statements in RTascell because it is clear from context.

Readers who are familiar with reversible computation might wonder if the so-called swap function is definable in the language. Swap statement `swap`(z,z') swaps the values of z and z'. Swap statement `swap`(z,z') is definable as `r-if`$(z$ `!=` $z')$`then`$\{z$ `^=` z'`;`z' `^=` z`;`z `^=` $z'\}$`else{skip}fi`$(z$ `!=` $z')$ in the language, where z and z' are variable or an array, as follows.

Proposition 4.1. *If* $\langle \texttt{swap}(z,z'),\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma'\rangle$ *holds, then* $\sigma(z) = \sigma'(z')$, $\sigma(z') = \sigma'(z)$, *and* $\sigma \upharpoonright \{z,z'\}^{\mathrm{c}} = \sigma' \upharpoonright \{z,z'\}^{\mathrm{c}}$ *hold where* $\sigma \upharpoonright \{z,z'\}^{\mathrm{c}}$ *denotes the partial function for which the domain is the restriction of the domain of* σ *by removing* $\{z,z'\}$.

Proof. It immediately holds by definition. □

Next, we define an inverse statement $(R)^{-1}$ for any reversible raw statement R as shown in **Fig. 9**. Reversible raw statements have reversibility, i.e., for a transition by any statement, its *inverse* statement raises the reverse transition as explained below.

$$
\begin{aligned}
(\texttt{skip})^{-1} &= \texttt{skip} & (\texttt{assert}(e))^{-1} &= \texttt{assert}(e) \\
(x \texttt{ += } e)^{-1} &= x \texttt{ -= } e & (x \texttt{ -= } e)^{-1} &= x \texttt{ += } e \\
(x \texttt{ ^= } e)^{-1} &= x \texttt{ ^= } e & (x[e_0] \texttt{ += } e_1)^{-1} &= x[e_0] \texttt{ -= } e_1 \\
(x[e_0] \texttt{ -= } e_1)^{-1} &= x[e_0] \texttt{ += } e_1 & (x[e_0] \texttt{ ^= } e_1)^{-1} &= x[e_0] \texttt{ ^= } e_1 \\
(R_0\,;R_1)^{-1} &= (R_1)^{-1}\,;(R_0)^{-1} & & \\
(\texttt{r-if}(e_0)\texttt{then}\{R_0\}\texttt{else}\{R_1\}\texttt{fi}(e_1))^{-1} & & & \\
= \texttt{r-if}(e_1)\texttt{then}\{(R_0)^{-1}\}\texttt{else}\{(R_1)^{-1}\}\texttt{fi}(e_0) & & &
\end{aligned}
$$

Fig. 9 Inverse statements of reversible raw statements.

$$
\begin{aligned}
\iota(r) &= r & \iota(\texttt{enddo}) &= \texttt{enddo} & \iota(\texttt{pop}) &= \texttt{pop} \\
\iota(\texttt{finish}\ j) &= \texttt{finish}\ j & \iota(S_0\,;S_1) &= \iota(S_0)\,;\iota(S_1) & & \\
\iota(U_0\,;U_1) &= \iota(U_0)\,;\iota(U_1) & & & & \\
\iota(\texttt{for}(x,es)\{U\}) &= \texttt{for}(x,es)\{\iota(U)\} & & & & \\
\iota(\texttt{do}(x,es)\{U\}) &= \texttt{do}(x,es)\{\iota(U)\} & & & & \\
\iota(\texttt{if}(e)\texttt{then}\{U_0\}\texttt{else}\{U_1\}) &= \texttt{if}(e)\texttt{then}\{\iota(U_0)\}\texttt{else}\{\iota(U_1)\} & & & & \\
\iota(\texttt{mark}) &= \texttt{mark} & & & & \\
\iota(\texttt{unmarked}\{U\}) &= \texttt{unmarked}\{\iota(U)\} & & & & \\
\iota(\texttt{r-dynamic-wind}(R,U)) &= \texttt{dynamic-wind}\ (\iota(R),\iota(U),\iota((R)^{-1})) & & & &
\end{aligned}
$$

$$
\begin{aligned}
\iota(\texttt{skip}) &= \texttt{skip} & \iota(\texttt{assert}(e)) &= \texttt{assert}(e) \\
\iota(x \odot\!= e) &= x = x \odot e & \iota(x[e_0] \odot\!= e_1) &= x[e_0] = x[e_0] \odot e_1 \\
\iota(R_0\,;R_1) &= \iota(R_0)\,;\iota(R_1) & & \\
\iota(\texttt{r-if}(e_0)\texttt{then}\{R_0\}\texttt{else}\{R_1\}\texttt{fi}(e_1)) & & & \\
= \texttt{if}(e_0)\texttt{then}\{\iota(R_0)\,;\texttt{assert}(e_1)\}\texttt{else}\{\iota(R_1)\,;\texttt{assert}(e_1\texttt{==}\texttt{0})\} & & &
\end{aligned}
$$

Fig. 10 Translation from RTascell into Tascell.

Proposition 4.2. $\langle R,\sigma\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma'\rangle$ *implies* $\langle (R)^{-1},\sigma'\rangle \rightsquigarrow_{\mathrm{r}} \langle \texttt{skip},\sigma\rangle$.

Proof. By induction on R. □

Proposition 4.3. $(\cdot)^{-1}$ *is involutive. That is,* $((R)^{-1})^{-1}$ *is the same as* R.

Proof. By induction on R. □

Finally, we give a translation $\iota(\cdot)$ from RTascell into Tascell as shown in **Fig. 10**. The following is the main theorem in this section.

Theorem 4.4. *1)* $\iota(S)$ *is left invertible and*
2) $\langle \iota(S), \texttt{nil}\rangle$ *is well-formed.*

Proof. By induction on S. We use Propositions 4.2 and 4.3 in the base cases. □

4.2 Implementation

We implemented an RTascell compiler by modifying the Tascell compiler available at GitHub [18]. The Tascell compiler is implemented using the SC Language Framework [17]. This framework, which is implemented in Common Lisp [35], provides a compiler from the *SC-0* language, which features an S-expression based syntax and the C semantics, into C, and an interpreter of pattern matching-based transformation rules over S-expressions. A language developer can implement a language as a translator to C by writing such transformation rules. Programmers can get executable files using this translator and a C compiler. The Tascell compiler is also implemented as such a transla-

```
1 (defvar *dynamic-wind-allowed* nil)
2
3 (extendrule statement tcell
4  ;; Existing pattern for Tascell's dynamic-wind statement
5  (#?(dynamic-wind
6       (:before ,@bef-body)
7       (:body ,@body)
8       (:after ,@aft-body))
9   (unless *dynamic-wind-allowed*
10    (error "dynamic-wind is not allowed in a user program."))
11  (let ((*dynamic-wind-allowed* nil))
12    ...))
13  ;; Existing rules for other Tascell statements including
14  ;; do-two and do-many
15  ...
16  ;; Additional pattern for RTascell's r-dynamic-wind statement
17  (#?(r-dynamic-wind
18       (:before ,@bef-body)
19       (:body ,@body))
20   (let ((aft-body (reverse (mapcar #'r-statement
21                                    bef-body)))
22         (*dynamic-wind-allowed* t))
23     (statement
24      ~(dynamic-wind
25         (:before ,@bef-body)
26         (:body ,@body)
27         (:after ,@aft-body)))))
28  ;; Additional pattern for RTascell's r-if statement
29  (#?(r-if ,exp1 ,stat1 ,stat2 ,exp2)
30   (statement
31    ~(if ,exp1
32         (begin ,stat1 (assert ,exp2))
33         (begin ,stat2 (assert (not ,exp2))))))
34  ;; Additional pattern for swap operations in RTascell
35  (#?(swap ,exp1 ,exp2)
36   (let ((tmp (fresh variable name))
37         (type (type of exp1)))
38     ~(begin
39        (def ,tmp ,type)
40        (= ,tmp ,exp1)
41        (= ,exp1 ,exp2)
42        (= ,exp2 ,tmp))))
43 )
```

Fig. 11 Transformation rule for implementing RTascell.

tor defined by transformation rules. Therefore, we implemented an RTascell compiler by modifying those rules.

Figure 11 portrays a transformation rule for implementing the RTascell compiler. A transformation rule named *rule-name* is definable as shown below.

`(extendrule` *rule-name ruleset-name*
`(#?`*pattern*$_1$ *lisp-forms*$_1$`)` ... `(#?`*pattern*$_n$ *lisp-forms*$_n$`))`

A transformation rule is applicable in the same manner as for an ordinary Common Lisp function that takes a single argument. When a rule is applied, the argument is tested to ascertain whether it is matched by *pattern*$_1$, ..., or *pattern*$_n$ in this order.

Each pattern is written using notation similar to backquote macros of Common Lisp. `,`*symbol* and `,@`*symbol* in a pattern respectively match a single list element and zero or more list elements. When *pattern*$_k$ first matches the argument, *lisp-forms*$_k$ are evaluated by a Common Lisp evaluator. The value of the last form is returned as the transformation result. This evaluation is executed in an environment where a part of the argument S-expression that matched `,`*symbol* or `,@`*symbol* is bound to *symbol*.

For example, the pattern in line 29 matches any list with five elements whose first element is the symbol `r-if`. The Lisp form in lines 30–33 is evaluated in an environment where the second, third, fourth, and fifth element of the argument S-expression are bound respectively to `exp1`, `exp2`, `stat1`, and `stat2`.

The tilde (`~`) characters used in lines 24, 31, and 38 function almost identically as a backquote (`‘`) character, which signals that the backquote macro is applied to the following expression, except that symbols appearing in the expression are treated as those in a special namespace for SC code.

To implement our transformation ruleset for RTascell by modifying the original ruleset for Tascell, we added patterns for the `r-dynamic-wind` (lines 17–27) and `r-if` (lines 29–33), and `swap` (lines 35–42) statements to the `statement` transformation rule.

The transformation rule for `r-dynamic-wind` statements follows the translation in Fig. 10. It generates inverse statements of the statements in the given `:before` clause in line 20–21. It constructs a `dynamic-wind` statement by placing the resulting inverse statements to its `:after` clause in lines 24–27. This `dynamic-wind` statement is applied to the `statement` rule recursively so that it is transformed to SC-0 by employing existing transformation rules written for the Tascell compiler. Here, `r-statement` used in line 20 is another transformation rule we defined, which takes a statement and returns its inverse statement following the definition in Fig. 9.

The transformation defined for `r-if` statements follows the last rule in Fig. 10. A given `r-if` statement is transformed to a Tascell's `if` statement that contains `asserts` at the ends of its then and else clauses.

The transformation defined for swap statements generates code for swapping values of two variables using a fresh temporary variable in a standard manner. Although we confirmed that the swap statements are definable using an `r-if` statement and bitwise exclusive OR operations, we did not adopt the implementation because it has shortcomings in terms of performance.

We also made a minor modification to the transformation defined for `dynamic-wind` statements (lines 5–12) so that the compiler aborts when a `dynamic-wind` statement appears in a user program, because RTascell has no `dynamic-wind` statements. This transformation is applicable only to a `dynamic-wind` statement generated from a `r-dynamic-wind` statement in lines 24–27. We implemented this mechanism using the dynamic variable `*dynamic-wind-allowed*`. We can also implement the language that has both `dynamic-wind` and `r-dynamic-wind` statements only by removing the code at lines 9–10.

5. Experiments

In this section, we demonstrate experiments. First, we show that typical search problems that can be written in Tascell can also be written in RTascell. We next present differences between the Tascell and RTascell programs to confirm reduction of descriptions. Finally, we confirm that only small differences exist in performance between the Tascell and RTascell programs.

5.1 Writing Programs in RTascell

We employ three programs: Pentomino, Nqueens, and TSP. Pentomino and Nqueens respectively represent implementations of simple backtracking search algorithms for pentomino puzzles [14] and N-queens problems. These programs were used

```
1 int ps = k, ap = tsk->a[p];
2 ...
3 dynamic_wind
4 { // before clause
5                   tsk->b[ps] = p+'A';
6   ps += pss[0]; tsk->b[ps] = p+'A';
7   ps += pss[1]; tsk->b[ps] = p+'A';
8   ps += pss[2]; tsk->b[ps] = p+'A';
9   ps += pss[3]; tsk->b[ps] = p+'A';
10  tsk->a[p] = tsk->a[j0];
11  tsk->a[j0] = ap;
12 }
13 { // body clause
14   for (kk=k; kk<70; kk++)
15     if (tsk->b[kk] == 0) break;
16   if (kk==70)
17     s += 1;
18   else {
19     if((kk+7>=70 || tsk->b[kk+7]!=0)
20        && (tsk->b[kk+1] != 0
21            || (kk+8 >= 70
22                || tsk->b[kk+8] != 0)
23            && tsk->b[kk+2] != 0))
24       {}
25     else
26       s+=search(kk,j0+1,j0+1,12,tsk);
27   }
28 }
29 { // after clause
30   ap = tsk->a[j0];
31   tsk->a[j0] = tsk->a[p];
32   tsk->a[p] = ap;
33   tsk->b[k] = 0;
34   ps = k;       tsk->b[ps] = 0;
35   ps += pss[0]; tsk->b[ps] = 0;
36   ps += pss[1]; tsk->b[ps] = 0;
37   ps += pss[2]; tsk->b[ps] = 0;
38   ps += pss[3]; tsk->b[ps] = 0;
39 }
```

Fig. 12 `dynamic_wind` statement in Pentomino in Tascell.

```
1 int ps = k;
2 ...
3 r_dynamic_wind
4 { // before clause
5                   tsk->b[ps] += p+'A';
6   ps += pss[0]; tsk->b[ps] += p+'A';
7   ps += pss[1]; tsk->b[ps] += p+'A';
8   ps += pss[2]; tsk->b[ps] += p+'A';
9   ps += pss[3]; tsk->b[ps] += p+'A';
10  swap (tsk->a[p], tsk->a[j0]);
11
12 }
13 { // body clause
14   for (kk=k; kk<70; kk++)
15     if (tsk->b[kk] == 0) break;
16   if (kk==70)
17     s += 1;
18   else {
19     if((kk+7>=70 || tsk->b[kk+7]!=0)
20        && (tsk->b[kk+1] != 0
21            || (kk+8 >= 70
22                || tsk->b[kk+8] != 0)
23            && tsk->b[kk+2] != 0))
24       {}
25     else
26       s+=search(kk,j0+1,j0+1,12,tsk);
27   }
28 }
29
30
31
32
33
34
35
36
37
38
39
```

Fig. 13 `r_dynamic_wind` statement in Pentomino in RTascell.

for performance evaluations in the original paper on Tascell [19]. Nqueens is often used as a benchmark program or as an example for task parallel systems [8], [11], [13], [26]. The source code of these Tascell programs is bundled with the Tascell Implementation. It is available on GitHub [18]. Also, TSP is a traveling salesman problem solver using a simple backtracking algorithm. We implemented a Tascell program for TSP, which is representative of a Tascell implementation of a graph algorithm.

5.1.1 Pentomino

Pentomino is a parallel program that finds all solutions for pentomino puzzles. **Figure 12** presents the `dynamic_wind` statement that appears in the Pentomino program and corresponds to lines 18–34 in Fig. 2, except for the difference that this program counts solutions using the variable `s` as a counter.

In the before clause, a worker first puts the `p`-th pentomino piece by setting the five elements of `tsk->b[]`, corresponding to the five cells which form the piece, to `p+'A'`. Here, the variable `ps` is used for calculating the positions of the cells filled by the `p`-th piece. `'A'` is used only to make the values of `tsk->b[]` print-friendly. Then, the worker updates `tsk->a[]` to record the fact that the `p`-th piece has been used by swapping `tsk->a[p]` and `tsk->a[j0]`. The local variable `ap`, which has been initialized to the value of `tsk->a[p]` when the before clause is executed, is used as a temporary variable for this swapping operation.

In the body clause, the worker checks the number of filled cells on the board. We introduce variable `s` only to count solutions, that is, `s` is defined on the domain of the marked nodes. If the board is full, the worker increments `s` to count the solutions. It corresponds to remembering the current state to get solutions later using `mark` explained in Section 2.3. Counting the solutions can be implemented in the Pentomino code because no state is doubly visited. Then, the worker performs a recursive call to try the remaining pieces unless it is clear that no space exists to place any piece.

In the after clause, the worker undoes the update to `tsk->a[]` by swapping `tsk->a[p]` and `tsk->a[j0]` again by reusing `ap` as a temporary variable. The worker removes the piece put in the before clause by setting the five elements of `tsk->b[]` to `0`.

We wrote an RTascell program for Pentomino. **Figure 13** shows the `r_dynamic_wind` statement in the RTascell. No difference exists between the Tascell and RTascell programs except for that shown in Figs. 12 and 13.

Because we cannot write assignment expressions in the before clause, we used increment operations to update `tsk->b[]` in lines 5–9, and a `swap` statement for swapping `tsk->a[p]` and `tsk->a[j0]` at line 10. The elements of `tsk->b[]` to be updated has been always initialized to `0` when the before clause is executed. By executing this before clause, a worker performs the same updates as that in the Tascell program.

After executing the body clause, the worker undoes the updates to `tsk->b[]` by executing the inverse operations of the series of increments at lines 5–9 in reverse order: `tsk->b[ps] -= p+'A';`

```
1 dynamic_wind
2 { // before clause
3   tsk->lb[n-1-ai+k] = 1;
4   tsk->rb[ai+k] = 1;
5   tsk->a[i] = tsk->a[k];
6   tsk->a[k] = ai;
7 }
8 { // body clause
9   s += nqueens(n, k+1, k+1, n, tsk);
10 }
11 { // after clause
12   tsk->lb[n-1-ai+k] = 0;
13   tsk->rb[ai+k] = 0;
14   tsk->a[k] = tsk->a[i];
15   tsk->a[i] = ai;
16 }
```

Fig. 14 dynamic_wind statement in Nqueens in Tascell.

```
1 r_dynamic_wind
2 { // before clause
3   tsk->lb[n-1-ai+k] += 1;
4   tsk->rb[ai+k] += 1;
5   swap (tsk->a[i], tsk->a[k]);
6
7 }
8 { // body clause
9   s += nqueens(n, k+1, k+1, n, tsk);
10 }
11
12
13
14
15
16
```

Fig. 15 r_dynamic_wind statement in Nqueens in RTascell.

`ps -= pss[3]; tsk->b[ps] -= p+'A'; ... ps -= pss[0]; tsk->b[ps] -= p+'A';`. The worker also undoes the updates to `tsk->a[]` by executing the inverse operation of the swapping: swapping the same two values again.

In summary, this RTascell program is no different from the Tascell one in the sense that the series of changes in `tsk->a[]` and `tsk->b[]` are the same in both programs. It is apparent that the code size can be reduced using RTascell.

5.1.2 Nqueens

Nqueens is a parallel program that finds all solutions for the N-queens problem. **Figure 14** shows the `dynamic-wind` statement that appears in the Nqueens program in Tascell. In the before clause, a worker sets the `k`-th queen at the `(i,k)`-th square by updating `tsk->lb[]`, `tsk->rb[]`, and `tsk->a[]`. In the body clause, the worker performs a recursive call to set the remaining queens. In the after clause, the worker undoes the updates in the before clause.

As we did for Pentomino, we wrote an RTascell program for Nqueens using increment operations and the swap function instead of assignment operations. **Figure 15** shows the `r_dynamic_wind` statement in the RTascell program. The Tascell and RTascell programs show no difference except for those shown in Figs. 14 and 15. We can reduce the code size using RTascell.

5.1.3 TSP

TSP is a parallel program that solves traveling salesman problems, that is, it finds the shortest possible path in a given weighted graph that visits each vertex exactly once and returns to the original vertex.

```
1 dynamic_wind
2 { // before clause
3   t[k] = next;
4   u[next] = 1;
5 }
6 { // body clause
7   int dis0;
8   int cost = plastv->costs[i];
9   dis0 = search (k+1,0,V[next].ne,t,u);
10  if (dis0<INT_MAX && dis0+cost<dis)
11    dis = dis0 + cost;
12 }
13 { // after clause
14  /* t[k] = 0; is unnecessary. */
15  u[next] = 0;
16 }
```

Fig. 16 dynamic_wind statement in TSP in Tascell.

```
1 r_dynamic_wind
2 { // before clause
3   t[k] += next;
4   u[next] += 1;
5 }
6 { // body clause
7   int dis0;
8   int cost = plastv->costs[i];
9   dis0 = search (k+1,0,V[next].ne,t,u);
10  if (dis0<INT_MAX && dis0+cost<dis)
11    dis = dis0 + cost;
12 }
13
14
15
16
```

Fig. 17 r_dynamic_wind statement in TSP in RTascell.

Figure 16 shows the `dynamic-wind` statement that appears in the TSP program in Tascell. Letting $V = \{v_0, \ldots, v_{|V|-1}\}$ be the set of all the vertices of the given graph, then in this program, `t[]` and `u[]` represent a path. Setting `t[`k`]` to i represents that the k-th visited vertex in the path is v_i. Setting `u[`i`]` to 1 represents that v_i is included in the path. Also, `u[]` is redundant, but it is used so that a worker can know instantly whether a vertex is included in the path.

The worker traverses the graph using the depth-first strategy updating `t[]` and `u[]`. In the before clause, the worker adds $v_{\texttt{next}}$ to the path as the `k`-th visited vertex. In the body clause, it performs a recursive call to traverse the remaining part of the graph and updates `dis`, which represents the shortest length of the remaining path starting from $v_{\texttt{t[k-1]}}$. In the after clause, the worker removes $v_{\texttt{next}}$ from the path. Here, the worker need not undo the update to `t[k]` because this element will not be referenced before the next update.

We wrote an RTascell program for TSP using increment operations instead of assignment operations, as was done for Pentomino and Nqueens. **Figure 17** shows the `r_dynamic_wind` statement in the RTascell program.

In this program, a worker performs almost identical computations to those of the Tascell program, but performs additional operations for undoing updates to `t[k]` by `t[k]+=next` at line 3. Furthermore, whereas a worker need not initialize the elements of `t[]` to `0` at the beginning of the search in Tascell, this initialization is necessary for RTascell. We expect that the effects on performance which are imposed by these additional operations are

not considerable. This supposition is confirmed in Section 5.2.

No difference exists between the Tascell and RTascell programs except for those shown in Figs. 16 and 17, and the initialization of `t[]`. The code size can be reduced using RTascell also for TSP.

5.2 Performance Evaluation

We compared the performance of Tascell and RTascell using the Pentomino, Nqueens, and TSP programs. We also measured the performance of sequential C programs and Cilk-5 programs for these applications.

The C and Cilk-5 programs we used are the same as were used in the experiments in the original paper on Tascell [19] for Pentomino and Nqueens. We wrote C and Cilk-5 programs for TSP.

As discussed in Section 1, when backtrack search algorithms are implemented with LTC, a worker must allocate a workspace and initialize it by a copying operation for each logical task creation. Although this applies to Cilk-5, we can reduce the number of such workspace allocations by exploiting a pseudovariable `SYNCHED` [36], which tells us whether there are any uncompleted tasks created in a current procedure. All of the Cilk-5 programs are implemented using this technique. Nevertheless, we would like to note that such allocations cannot be completely removed unlike Tascell.

We solved pentomino puzzles with 14 pieces (using two additional pieces and an expanded board as described in the original paper on Tascell [19]) using Pentomino, 16-queens problems using Nqueens, and traveling salesman problems for a 2D-torus with 6^2 vertices using TSP.

We used two Xeon E5-2695 v4 processors, each of which has 18 cores. We used Cilk 5.4.6 for compiling the Cilk-5 programs. Like Tascell and RTascell, this Cilk-5 compiler is implemented as a translator to C. We used a GCC Compiler 9.3.0 with the `-O3` option for compiling the C programs. We also used it as a backend compiler for Tascell, RTascell, and Cilk-5. We measured execution times of one-worker and 36-worker executions for each parallel program. We executed the program three times for each measurement setting and presented the median of the execution times. The relative errors of the median samples were less than ±3.28% for every setting.

Table 1 presents the measured results. The execution times of the Cilk-5 programs (t_C) were worse than those of the Tascell programs (t_T) due to costs for managing logical task pools and allocating/initializing additional workspaces. The differences in the execution times between Cilk-5 and Tascell were larger for TSP due to larger cost for workspace initialization. These results reconfirm the advantages of Tascell over Cilk-5.

The execution times of the RTascell programs (t_R) were slightly worse than those of the Tascell programs (t_T) for Pentomino and TSP because increment and decrement operations are used instead of assignment operations in the `r_dynamic_wind` statements. They can prevent optimization of the C compiler and can cause overhead. On the other hand, the execution times of the RTascell program for Nqueens are almost the same as those of the Tascell program. This is probably because the overhead caused by increment/decrement operations is quite small for Nqueens.

Table 1 Execution times of Cilk-5, Tascell and RTascell programs.

Program	n_w	Execution time [s] t_s	t_C	t_T	t_R	t_R/t_T
Pentomino	1	65.5	102.5	70.1	74.8	1.07
	36	—	2.86	2.04	2.17	1.06
Nqueens	1	63.6	145	81.2	81.1	0.999
	36	—	4.30	2.32	2.32	1.00
TSP	1	415	4035	574	601	1.05
	36	—	119	16.0	16.8	1.05

n_w: # of workers
t_s: Execution time of sequential C programs
t_C: Execution time of Cilk-5 programs
t_T: Execution time of Tascell programs
t_R: Execution time of RTascell programs

In summary, slight differences were found between the Tascell and RTascell programs in terms of the performance, but the differences are not remarkable.

6. Related Work and Discussion

Flanagan and Felleisen were the first to provide formal operational semantics for concurrent programs in task parallel processing [12]. Their programs consist of so-called `future` constructs for task creation (e.g., Ref. [15]). They defined transition rules using evaluation contexts to construct an abstract machine. The transition rules consist of sequential and concurrent transitions similar to our $\rightsquigarrow_r$, $\rightsquigarrow_l$, and $\rightsquigarrow_g$ transitions. They also verified touch optimization by showing equivalence between original and optimized programs on the abstract machine. Although their work and our work share the same motivation (i.e., to define operational semantics for task parallel programs), they have some mutual differences. In the present paper, we provided small-step operational semantics for concurrent programs with task parallel processing, adopted Church–Rosser modulo equivalence to define observable equivalence, and verified backtracking-based load balancing.

Yasugi provided denotational and operational semantics for sequential and concurrent programs, respectively, and described safety in task parallel processing and synchronization [42]. However, he did not provide any sufficient condition for the safety unlike the present study.

Moore and Grossman provided small-step operational semantics for concurrent programs [29]. They studied transactions. They constructed a type system and proved that every typable program has the same result under its sequential and concurrent executions. There are three important differences compared to our work. They studied transactions, not backtracking-based load balancing. Another difference is that they used typability, not backtrackability, to specify programs that have consistent behavior. Furthermore, they adopted the coincidence of results under two transition relations to define their equivalence relation, whereas we used Church–Rosser modulo equivalence of the unique transition relation $\rightsquigarrow_g$.

Burckhardt and Leijen provided operational semantics for task parallel programs with concurrent revisions rather than locks or transactions [7]. For that research, equivalence is defined using confluence in abstract rewriting theory. Burckhardt et al.

also clarified that their concurrent revision model can be implemented efficiently enough to achieve satisfactory parallelization speedups [6]. Their work and ours share similar methods and contributions. However, whereas they studied concurrent revisions for determinacy and consistency of concurrent programs, we studied sufficient conditions for consistency of concurrent programs with backtracking-based load balancing in the present study. Their work and ours illustrate that abstract rewriting theory is useful to define various equivalences between concurrent programs.

Khaldi et al. defined intermediate representations for parallel processing, including so-called *spawn* instructions, with operational semantics [24]. They demonstrated that future constructs can be supported by their intermediate representations in the X10 programming language [9]. These representations are used in high-performance computing. However, they did not formally prove that sequential and concurrent behaviors coincide. Atzeni and Gopalakrishnan provided operational semantics for OpenMP programs, which are used in high-performance computing [3]. However, they did not define equivalence between OpenMP programs.

Church–Rosser modulo equivalence is useful to define equivalence in theoretical work, for example, to define λ-terms in λ-calculi (e.g., Ohta and Hasegawa's paper [31]). There is also some work in more applied computer science. The work described herein demonstrated that task parallel processing with work stealing is one interesting application.

Various verifications of task parallel programs [37], [40] and several static verifications of task parallel programs [27], [30] have been reported recently. For example, Mercer et al. proposed a method for verification of ensuring deadlock freedom and data race freedom based on model checking [27]. Such studies differ from ours. Our definition of correctness is equivalence between behaviors in sequential and parallel executions. In addition, because we designed a domain-specific language and implemented a compiler with lightweight checking for correctness, our verification theory is not based on heavyweight verification methods such as model checking.

It is possible to define parallelizability for programs in task parallel programming languages such as Cilk-5 [13] and OpenMP [33] and confirm extensional equivalences between sequential and concurrent behaviors of programs that are parallelizable more easily because their sequential and concurrent executions share the same states without temporal backtrackings. That is, it is sufficient to write operational semantics (like our previous work [1] for an OpenMP-like language XcalableMP [41]) and define properties corresponding to the parallelizability and well-formedness properties without considering backtrackability or left invertibility. Thus, although Cilk-5 and OpenMP are theoretically easier to tame, backtracking-based load balancing neither keeps nor copies states and Tascell has benefits of performance over other task parallel processing languages, as reported [19].

7. Conclusion and Future Work

As described herein, we provided operational semantics for Tascell programs and defined extensional equivalence between Tascell programs using Church–Rosser modulo equivalence in abstract rewriting theory. We characterized the left invertible and well-formedness properties to ensure consistent behaviors of Tascell programs. In addition, we verified the correctness of backtracking-based load balancing, i.e., programs that satisfy the properties return the same results irrespective of whether backtracking-based load balancing is activated. We also designed and implemented a programming language RTascell based on reversible computation to ensure that their programs have left invertible and well-formedness properties. To confirm the effectiveness of our work, we also demonstrated experiments using typical search problems such as pentomino puzzles, N-queens problems, and traveling salesman problems.

We have found that some programs cannot be described in RTascell. For example, Okuno et al. proposed more efficient algorithms [32] that solve the itemset-sharing subgraph extraction problem [34] proposed by Sese et al. A graph for which the vertices are labeled by their own sets of items (itemsets for short) and a threshold are given. A solution is a connected subgraph in the graph that satisfies the condition that the cardinality of its common itemset, i.e., the intersection of the itemsets of all of its vertices, is not less than the given threshold. The ISS extraction problem is to extract all solutions. This kind of graph mining is applicable to real problems such as the analysis of social and biological networks. Because the algorithms use aggressive *pruning* with global tables to remember traversed graphs, we cannot currently describe them in RTascell. The construction of a theory that can describe such algorithms remains as a great challenge for future work.

Acknowledgments We thank Daisuke Kimura for reading the first draft, finding a few flaws, and providing several useful comments. We also thank Makoto Hamana and Kentaro Kikuchi for providing literature information of abstract rewriting theory, which keeps the proofs in the present paper simple. We also thank David Castro-Perez for the fruitful discussion, which makes the contributions of the present work clearer. We also thank the anonymous reviewer for providing several suggestions. This work was supported by JSPS KAKENHI Grant Numbers JP17K00099 and JP19H04087.

References

[1] Abe, T.: A Type System for Data Independence of Loop Iterations in a Directive-Based PGAS Language, *Proc. MPLR*, pp.50–62 (2019).
[2] Aoto, T. and Toyama, Y.: A Reduction-Preserving Completion for Proving Confluence of Non-Terminating Term Rewriting Systems, *Logical Methods in Computer Science*, Vol.8, No.1, pp.1–29 (2012).
[3] Atzeni, S. and Gopalakrishnan, G.: An Operational Semantic Basis for Building an OpenMP Data Race Checker, *Proc. HIPS*, pp.395–404 (2018).
[4] Bennett, C.H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol.17, No.6, pp.525–532 (1973).
[5] Berry, D., Milner, R. and Turner, D.: A Semantics for ML Concurrency Primitives, *Proc. POPL*, pp.119–129 (1992).
[6] Burckhardt, S., Baldassin, A. and Leijen, D.: Concurrent Programming with Revisions and Isolation Types, *Proc. OOPSLA*, pp.691–707 (2010).
[7] Burckhardt, S. and Leijen, D.: Semantics of Concurrent Revisions, *Proc. ESOP*, LNCS, Vol.6602, pp.116–135, Springer (2011).
[8] Castelló, A., Seo, S., Mayo, R., Balaji, P., Quintana-Ortí, E.S. and Peña, A.J.: GLT: A Unified API for Lightweight Thread Libraries, *Proc. Euro-Par*, pp.470–481 (2017).
[9] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A.,

Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, *SIGPLAN Notices*, Vol.40, No.10, pp.519–538 (2005).
[10] Cytron, R.: DOACROSS: Beyond Vectorization for Multiprocessors, *Proc. ICPP*, pp.836–844 (1986).
[11] Duran, A., Corbalán, J. and Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies, *Proc. IWOMP*, pp.100–110 (2008).
[12] Flanagan, C. and Felleisen, M.: The Semantics of Future and Its Use in Program Optimization, *Proc. POPL*, pp.209–220 (1995).
[13] Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. PLDI*, pp.212–223 (1998).
[14] Golomb, S.W.: *Polyominoes: Puzzles, Patterns, Problems, and Packings*, Princeton University Press, 2nd edition (1996).
[15] Halstead, Jr, R.H.: New ideas in parallel Lisp: Language design, implementation, and programming tools, *Parallel Lisp: Languages and Systems*, LNCS, Vol.441, pp.2–57 (1990).
[16] Hamana, M.: How to Prove Decidability of Equational Theories with Second-Order Computation Analyser SOL, *Journal of Functional Programming*, Vol.29, No.e20, pp.1–53 (2019).
[17] Hiraishi, T.: Transformation-based Implementation of S-expression Based C Languages, PhD Thesis, Kyoto University (2008).
[18] Hiraishi, T.: *Tascell: Backtracking-based Load Balancing Framework* (2020), available from ⟨https://github.com/tascell/sc-tascell⟩ (accessed 2020-04-27).
[19] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, *Proc. PPoPP*, pp.55–64 (2009).
[20] Huet, G.: Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, *Journal of the ACM*, Vol.27, No.4, pp.797–821 (1980).
[21] ISO/IEC 9899:1999: Programming Language C (2000).
[22] Jouannaud, J.-P., Kirchner, H. and Remy, J.-L.: Church–Rosser Properties of Weakly Terminating Term Rewriting Systems, *Proc. IJCAI*, pp.909–915 (1983).
[23] Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *SIGPLAN Notices*, Vol.33, No.9, pp.26–76 (1998).
[24] Khaldi, D., Jouvelot, P., Irigoin, F. and Ancourt, C.: SPIRE: A Sequential to Parallel Intermediate Representation Extension, *Proc. CPC* (2013).
[25] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol.5, No.3, pp.183–191 (1961).
[26] Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley Professional, 2nd edition (1999).
[27] Mercer, E.G., Anderson, P., Vrvilo, N. and Sarkar, V.: Model Checking Task Parallel Programs Using Gradual Permissions, *Proc. ASE*, pp.535–540 (2015).
[28] Mohr, E., Kranz, D.A. and Halstead, Jr, R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).
[29] Moore, K.F. and Grossman, D.: High-level small-step operational semantics for transactions, *Proc. POPL*, pp.51–62 (2008).
[30] Nakade, R., Mercer, E., Aldous, P. and McCarthy, J.: Model-Checking Task Parallel Programs for Data-Race, *Proc. NFM*, LNCS, Vol.10811, pp.367–382 (2018).
[31] Ohta, Y. and Hasegawa, M.: A Terminating and Confluent Linear Lambda Calculus, *Proc. RTA*, LNCS, Vol.4098, pp.166–180 (2006).
[32] Okuno, S., Hiraishi, T., Nakashima, H., Yasugi, M. and Sese, J.: Parallelization of Extracting Connected Subgraphs with Common Itemsets, *IPSJ Trans. Programming*, Vol.7, No.3, pp.22–39 (2014).
[33] OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 4.5* (2015).
[34] Sese, J., Seki, M. and Fukuzaki, M.: Mining Networks with Shared Items, *Proc. CIKM*, pp.1681–1684 (2010).
[35] Steele, Jr, G.L.: *Common Lisp: The Language*, 2nd edition, Digital Press (1990).
[36] Supercomputing Technologies Group, MIT Laboratory for Computer Science: *Cilk 5.4.6 Reference Manual*, available from ⟨http://supertech.lcs.mit.edu/cilk/⟩.
[37] Surendran, R. and Sarkar, V.: Dynamic Determinacy Race Detection for Task Parallelism with Futures, *Proc. RV*, LNCS, Vol.10012, pp.368–385 (2016).
[38] Terese: *Term Rewriting Systems*, Cambridge University Press (2003).
[39] Toffoli, T.: Computation and Construction Universality of Reversible Cellular Automata, *Journal of Computer and System Sciences*, Vol.15, No.2, pp.213–231 (1977).
[40] Westbrook, E., Raman, R., Zhao, J., Budimlić, Z. and Sarkar, V.: Dynamic Determinism Checking for Structured Parallelism, *Proc. Wodet* (2014).
[41] XcalableMP Specification Working Group: *XcalableMP Application Program Interface Version 1.4* (2018).
[42] Yasugi, M.: Hierarchically Structured Synchronization and Exception Handling in Parallel Languages Using Dynamic Scope, *Proc. PDSIA*, World Scientific, pp.122–148 (2000).
[43] Yokoyama, T., Axelsen, H.B. and Glück, R.: Principles of a Reversible Programming Language, *Proc. CF*, pp.43–54 (2008).
[44] Yokoyama, T. and Glück, R.: A Reversible Programming Language and Its Invertible Self-Interpreter, *Proc. PEPM*, pp.144–153 (2007).

Tatsuya Abe was born in 1979. He received his B.Sc. and Ph.D. degrees from Kyoto University and the University of Tokyo in 2002 and 2007, respectively. He worked for National Institute of Advanced Industrial Science and Technology, Kyoto University, and RIKEN. He is currently a senior research scientist at STAIR Lab, Chiba Institute of Technology. His research interests include programming languages, program verification, and concurrency. He won the IPSJ Yamashita SIG Research Award in 2018. He is a member of IPSJ, JSSST, and ACM.

Tasuku Hiraishi was born in 1981. He received his B.E. in Information Science in 2003, Master of informatics in 2005, and a Ph.D. in informatics in 2008, all from Kyoto University. In 2007–2008, he was a JSPS fellow at Kyoto University. Since 2008, he has been working at Kyoto University as an assistant professor at Supercomputing Research Laboratory, Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of IPSJ, JSSST, and ACM.