A Type Discipline for Message Passing Parallel Programs

VASCO T. VASCONCELOS, LASIGE, Faculdade de Ciências, Universidade de Lisboa, PT
FRANCISCO MARTINS, LASIGE, Faculdade de Ciências, Universidade de Lisboa, and University of the Azores, PT
HUGO-ANDRÉS LÓPEZ, University of Copenhagen and DCR Solutions A/S, DK
NOBUKO YOSHIDA, Imperial College London, UK

We present ParTypes, a type discipline for parallel programs. The model we have in mind comprises a fixed number of processes running in parallel and communicating via collective operations or point-to-point synchronous message exchanges. A type describes a protocol to be followed by each processes in a given program. We present the type theory, a core imperative programming language and its operational semantics, and prove that type checking is decidable (up to decidability of semantic entailment) and that well-typed programs do not deadlock and always terminate. The article is accompanied by a large number of examples drawn from the literature on parallel programming.

CCS Concepts: • **Software and its engineering → Deadlocks**; **Software verification**; **Dynamic analysis**; *Concurrent programming structures*; *Semantics;*

Additional Key Words and Phrases: Parallel programs, message passing computation, dependent types, termination

ACM Reference format:
Vasco T. Vasconcelos, Francisco Martins, Hugo-Andrés López, and Nobuko Yoshida. 2022. A Type Discipline for Message Passing Parallel Programs. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 26 (December 2022), 55 pages.
https://doi.org/10.1145/3552519

1 INTRODUCTION

Parallel programming allows tackling problems so large that it would otherwise be impractical to solve on a single computer. It allows to shorten time to completion with the associated cost savings.

This work was supported by FCT through project Advanced Type Systems for Multicore Programming (PTDC/EIA-CCO/122547) and the LASIGE Research Unit (UID/CEC/00408/2019), EC Cost Action EUTypes (CA15123), EPSRC (EPSRC (EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1, EP/V000462/1, and EP/X015955/1), NCSS/EPSRC VeTSS, the Innovation Fund Denmark project EcoKnow.org (7050-00034A), and the European Union Marie Sklodowska-Curie grant agreement BehAPI (778233).
Authors' addresses: V. T. Vasconcelos, LASIGE, Faculdade de Ciências, Universidade de Lisboa, Departamento de Informática, Lisboa, PT; email: vmvasconcelos@ciencias.ulisboa.pt; F. Martins, LASIGE, Faculdade de Ciências, Universidade de Lisboa, and University of the Azores, Faculty of Sciences and Technology, Ponta Delgada, PT; email: fmartins@acm.org; H.-A. López, University of Copenhagen and DCR Solutions A/S, Department of Computer Science, Copenhagen, DK; email: lopez@di.ku.dk; N. Yoshida, Imperial College London, Department of Computing, London, UK; email: n.yoshida@imperial.ac.uk.

0164-0925/2022/12-ART26
https://doi.org/10.1145/3552519

And parallel hardware is today present on most machines, from a few cores in smartphones to hundreds of thousands in large parallel supercomputers.

When compared to sequential applications, the difficulty of developing parallel applications is exacerbated by the fact that different processes running in parallel synchronise in order to exchange data. It is not difficult to write a program that causes processes to exchange data of unexpected sorts or lengths or that blocks indefinitely waiting for messages that will never arrive. Such faults can become quite costly, if, e.g., the application is running on a large supercomputer. Verifying these sort of properties for parallel programming is far from trivial. The most common techniques include runtime verification [30, 58, 65, 77], symbolic execution and model checking [25, 30, 58, 68, 72].

Runtime verification can never guarantee the absence of faults. In addition, the testing process can become daunting due to the difficulty in producing meaningful tests, the time to run the whole test suite, and the need to run the test suite on hardware similar to that where the final application will eventually be deployed. On the other hand, model checking approaches frequently stumble upon the problem of scalability, given that the search space grows exponentially with the number of processes. It is often the case that the verification of real applications limits the number of processes to less than a dozen [69].

In this article, we concentrate on a model of parallel programming featuring a fixed number of processes, each with its local memory, running its own program and communicating exclusively by point-to-point synchronous message exchanges or by synchronising via collective operations, such as broadcast or reduce. ParTypes is a type discipline for parallel programs; its programme is as follows: Types describe the interactive behaviour of programs. They will never assert that a given program effectively multiplies two matrices, but they will tell the pattern of interaction between the processes involved in a matrix multiplication program. Programs that conform to a type are guaranteed not enter a deadlocked situation where, e.g., one process is trying to broadcast and another to send a message, or where each process is trying to receive a message from the next, in a ring topology. We show that there is an effective procedure to check whether a program conforms to a type, hence that a type checker can be built for an imperative language that uses ParTypes as the type language.

A cornerstone of our approach is that all processes in a parallel program must share the same type. One can write a distinct program for each individual process, a unique program to be shared by all processes, or all the intermediate possibilities (such as programs for the even processes and for the odd processes, or program for one distinguished process and another for all others). In any case, the types for all programs must agree. Then, all processes are forced to comply to a single, global protocol (in the form of a type), forcing processes to synchronise and interact according to the protocol, thus precluding deadlocks by typing [39].

ParTypes is heavily inspired by MPI [17], covering point-to-point communication (with primitives send and receive) and well as different sorts of collective communication (such as, broadcast and reduce). Primitive data include integer scalars, arrays and pairs. All process interaction is blocking (as opposed to buffered, where send operations are non-blocking). We provide an operational semantics according to the standard [17] as we interpret it, and propose a type system for the core that we have selected. We cover only a very small fragment of MPI; the discussion (Section 8) addresses some limitations. We introduce ParTypes progressively, to a point where it can be used to model a large class of parallel programs found in the literature [19, 27, 57, 61]. ParTypes allow writing the same program for all processes (as in MPI) as well as different program for different processes (unlike MPI).

We use violet to write code and boldface **blue** to describe types. The type language includes point-to-point communications such as

message 1 2

detailing an exchange of an integer value from process 1 to process 2. Types depend on terms, so that we often see dark violet bits in types, as in the **message** type above. Collective operation types such as

$$\textbf{reduce}\ 0$$

characterise a program where all processes propose an integer value and process 0 receives the sum of them all. ParTypes is a dependently-typed language along the lines of DML [81]. Dependent types allow for protocols to depend on values transmitted before. If *size* denotes the number of processes, then type

$$\textbf{broadcast}\ 0\ \mathit{sink}\colon \{x\colon \textbf{int} \mid 0 \leq x < \mathit{size}\}.\ \textbf{reduce}\ \mathit{sink}$$

describes a program whose process 0 proposes a process number (the *sink*) and subsequently all processes agree to reduce on this process. The number of a process, an integer value between 0 and $\mathit{size}-1$ is often called the *rank* of the process. The type of identifier *sink* is of a *refined datatype* denoting an integer value greater or equal to 0 and smaller than *size*.

The base type language is completed with a *sequential composition* $T;U$ operator; type **skip** that describes processes without any interaction, *primitive recursion* $\forall\ x{<}i.T$, and a form of *collective conditional* $\textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U$ allowing all processes in a program to take an identical decision based solely on information p exchanged in collective operations (and contained in types). The base language is then extended with a number of type constructors in order to accommodate arrays and array **scatter**/**gather** operations, further collective operations such as **allreduce** and **allgather**, and pairs and **allreducemaxloc** (handy for the parallel implementation of Gaussian elimination [61]).

Type equivalence is at the core of ParTypes. It includes the monoidal rules for sequential composition and **skip**, expansion and elimination of primitive recursion, the elimination of the collective conditional $\textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U$ when p can be shown to be true or false. But perhaps the most vital rule is the *projection rule*, allowing to eliminate **message** types. The rule allows aligning the types of all processes on what concerns point-to-point operations even for processes that are not involved in the communication.

Imagine a program with three processes where process 0 sends an integer message to process 2, and process 1 does not engage in any interactive behaviour. The type of both processes 0 and 2 is $\textbf{message}\ 0\ 2$. But the type of process 1 must be aligned with these two. Because the process does not engage in any interaction, one of its many types is **skip**. But because 1 is different both from 0 and from 2, type **skip** is equivalent to $\textbf{message}\ 0\ 2$ for process 1. In this way, the three processes all have the same type, thus guaranteeing the absence of deadlocks.

This article consolidates the theory of ParTypes introduced in López et al. [46], elaborates on the decidability of type equivalence and type checking, introduces strong normalization for typed processes, adds examples and proofs, and extends the type language with a large number of new type constructors. We do not cover the verification of C+MPI programs using the VCC tool [9, 50]. The detailed differences with respect to López et al. [46] are as follows:

- We introduce the complete set of rules for program and type formation and for type equivalence (Sections 2 and 3);
- We include a new strong normalization theorem (Section 4);
- We introduce an algorithm for type checking programs and prove its correctness against the declarative system (Section 5);
- We extend the base type language with new datatypes and new constructors for collective operations (Section 6).

Outline. The next section introduces a core programming language for parallel programs and its operational semantics. Section 3 presents the notion of types as protocols to discipline process interaction, type equivalence and program typing. Section 4 proves progress, preservation, and strong normalisation for well typed programs. Section 5 presents an algorithmic typing system and proves it sound with respect to the declarative system introduced in Section 3. Section 6 discusses a number of extensions to the type language towards a more encompassing modelling of extant protocols. Section 7 discusses related work and Section 8 concludes this article.

2 A MESSAGE PASSING PARALLEL IMPERATIVE LANGUAGE

This section introduces the syntax and the operational semantics of programs, starting with a few illustrative examples.

2.1 Examples of Common Parallel Programs

Four examples highlight the salient features of ParTypes. To concentrate on the interactive behaviour, we make use of calls to undefined functions (such as `read_int ()`). Finally, we use a wildcard (`_`) in place of a variable when its value is not important.

The Monte Carlo Algorithm. We start with the Monte Carlo method to estimate the value of π [57]. Suppose we toss darts randomly at a circular target inscribed on a square board. If the points that are hit by the darts are uniformly distributed (and darts always hit the square board), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{darts in circle}}{\text{total number of tosses}} = \frac{\pi}{4}$$

since the ratio of the area of the circle to that of the square is $\pi/4$.

Estimating the value of π can be easily parallelised by distributing dart tossing among a given number of parallel processes. Our implementation works by distinguishing one process, the root process (usually at rank 0), to `broadcast` the number of darts each process is supposed to toss. Each individual process then computes the number of hits. A `reduce` operation collects the sum of all such hits at the root process. To optimize resources, the root process also computes its share. Predefined variable `size` denotes the number of processes. We start with the code for the root process, making use of functions to read an integer and to compute the number of hits given a number of darts.

```
1 let darts:int = broadcast 0 read_int() in
2 let darts_in_circle:int ref = mkref 0 in
3 reduce 0 (compute_number_of_hits(darts)) darts_in_circle;
4 print(4 * !darts_in_circle / (darts * size))
```

The root process reads the number of darts per process and broadcasts it to all processes, including itself (line 1). The process then prepares a reference — `darts_in_circle` — to receive the contributions from all processes, including itself (line 2). The `reduce` operation stores in the reference the sum of the contributions; line 4 prints the approximate value of π. The number `0` in both the `broadcast` and the `receive` operations denote the rank of the process that distributes and collects data (the root process).

The code for the remaining processes is somewhat simpler: each process first receives the number of darts to toss, stores it in variable `darts` (line 1) and then computes and propose the number of hits (line 2). ParTypes (as MPI) provides one only broadcast operation for all processes, so that the non-sending partners must provide a mock value (−1 in this case) in the place where the root process provides a concrete value.

```
1 let darts:int = broadcast 0 (−1) in
2 reduce 0 (compute_number_of_hits(darts)) (mkref 0)
```

Similarly, there is one only reduce operation for all processes, hence the mock mkref 0 in line 2. The operational semantics (Section 2.3) will not evaluate the last expressions in both broadcast and reduce operations on non-root processes. By not distinguishing the root process from the remaining processes in collective operations, this unusual syntax and behaviour (standard in MPI) allows the same source code to be run on all processes.

Combining the two snippets of code above, and taking advantage of the predefined variable myrank denoting the rank of a process, we obtain code that can be run on any process:

```
1 let darts:int = broadcast 0 read_int() in
2 let darts_in_circle:int ref = mkref 0 in
3 reduce 0 (compute_number_of_hits(!darts)) darts_in_circle;
4 if myrank == 0 then print(4 * !darts_in_circle / (darts * size))
```

In this case, all processes allocate memory in line 2, but only the root process actually uses it,

Message Exchange on a Ring Topology. This simple example illustrates the exchange of messages on a topology where all processes receive a value from their left neighbour and send another value to their right neighbour. The process at the "left" of rank 0 is rank size −1. The example computes the factorial of size . The invariant is that the process at rank i receives the factorial of i from its left neighbour and sends to its right neighbour the factorial of $i + 1$. At the end, process at rank 0 receives the factorial of size from process at rank size −1.

Our implementation distinguishes three processes: process at rank 0 (the root), processes between rank 1 and size −2, and process at rank size −1. The root process starts the computation by sending number 1 (the factorial of its rank) to its right neighbour. Then, the process at rank i (with $i <$ size −1) receives the factorial of i from its left neighbour, and sends the value the factorial of $i + 1$ to its right neighbour. Finally, the last process (at rank size −1), receives the factorial of size −1 from its left neighbour, and sends the factorial of size to the root that prints it. The source code for the root process is as follows:

```
1 let fact:int ref = mkref 0 in
2 send 1 1;
3 receive size−1 fact;
4 print(!fact)
```

The source code for processes at intermediate ranks is:

```
1 let fact:int ref = mkref 0 in
2 receive myrank−1 fact;
3 send myrank+1 !fact*(myrank+1)
```

and source code for the last process is as follows:

```
1 let fact:int ref = mkref 0 in
2 receive myrank−1 fact;
3 send 0 !fact*size
```

The first argument to a send or receive operation is the target rank, the second is the integer to be sent or the reference on which to store the value received. Notice that the communication pattern for the root process is different from that of the other processes: it first sends to its right neighbour and then receives from its left neighbour, while all other processes first receive from

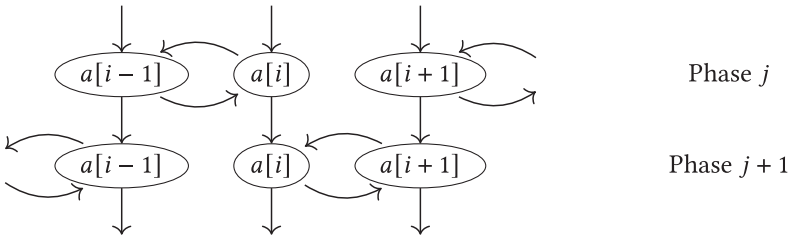

Fig. 1. Communications among processes in an odd-even sort (tasks determining $a[i]$ are labelled with $a[i]$).

their left neighbours and then send to their right neighbours (with "left" and "right" understood as in a ring topology). This asymmetry is sufficient to break the communication deadly embrace. Otherwise, the complete program would run into a deadlock with all processes blocked at either sending or receiving operations.

ParTypes also allows for using the same source code for all processes, as opposed to different source code for different (classes of) processes. Next, we illustrate how to combine the source code of all processes but the root, making use of the modulus operation. In fact, all processes with 0 < myrank < size −1 exhibit the same behaviour.

```
1 let fact:int ref = mkref 0 in
2 receive myrank-1 fact;
3 send (myrank+1)%size !fact*(myrank+1)
```

Lastly, we show how to combine the code in a single program. The behaviour of the root process is distinguished from all others by checking the value of myrank.

```
1 let fact:int ref = mkref 0 in
2 if myrank == 0 then
3   send 1 1;
4   receive size-1 fact;
5   print(!fact)
6 else
7   receive myrank-1 fact;
8   send (myrank+1)%size !fact*(myrank+1)
```

Odd-Even Sort. Our next example is the parallel transposition sorting algorithm [57]. The algorithm consists of a sequence of odd-even phases. Given an array a to sort, during the even phases compare-swap operations are executed on the following pairs

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots$$

and on odd phases compare-swap operations are executed on

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots$$

Figure 1 shows how the algorithm proceeds in the particular case when the length of the array matches the number of processes. For simplicity, we assume size to be even. It should be clear that distinct source code must be run at rank 0, ranks 1 to size −2 and at rank size −1. By taking advantage of the distinguished variable myrank we can write a single expression that runs on all processors.

```
1  let my_key, partner_key:int ref = mkref read_int(), mkref 0 in
2  for phase < size do
3    ifc even(phase) then
4      if even(myrank) then
5        send (myrank+1) !my_key; receive (myrank+1) partner_key;
6        my_key := min(!my_key, !partner_key)
7      else // odd(myrank)
8        receive (myrank-1) partner_key; send (myrank-1) !my_key;
9        my_key := max(!my_key, !partner_key)
10   else // odd(phase)
11     if myrank != size-1 then
12       if odd(myrank) then
13         send (myrank+1) !my_key; receive (myrank+1) partner_key;
14         my_key := min(!my_key, !partner_key)
15       else // odd(myrank)
16         receive (myrank-1) partner_key; send (myrank-1) !my_key;
17         my_key := max(!my_key, !partner_key)
```

The initial key of each process is obtained via function read_int () (line 1). Line 2 introduces a downwards loop controlling the number of phases. At even phases, even processes exchange keys with the right process (line 5); the smaller key is retained by the process at the left (line 7) and the larger by the process at the right (line 9). At odd phases, the same procedure, this time for odd processes: keys are exchanged with the right process (line 13) and the smaller key retained by the left process (line 14). Because we are assuming that size is even, the last process is of odd rank and, therefore, does not have a right neighbour to exchange keys with. The conditional at line 11 controls this situation.

Lines 3 and 4 exhibit two different flavours of the conditional instruction: ifc stands for a conditional where all processes decide equally (the c is for *collective*); if is a conventional conditional. The conventional conditional is evaluated locally, at the process level, and is independent of the remaining processes. The collective conditional is performed by all processes at the same time so that they all decide equally on the phase.

The n-Body Problem. Our last example looks at another classical problem of parallel programming, the n-body problem, which computes the trajectories of n bodies that influence each other through gravitational forces. The algorithm computes the forces between all pairs of bodies, applying a pipeline technique to distribute and balance the work on a parallel architecture [28, 29, 57]. It then determines the bodies' positions.

The overall idea of the algorithm is as follows: each process obtains the position and velocity of a particle. Then, process rank 0 broadcasts the number of iterations for the simulation. Thereafter, each process enters a loop that computes that many discrete steps. In each iteration, the algorithm computes the forces between all pairs of particles by having each process computing the forces between its particle and those from the neighbour processes. Towards this end, the algorithm relies on a ring topology where each process passes one particle's data to the right process and receives a new particle's data from the left. Then it computes the forces against the received particle. After size −1 steps all processes have visited all particles. Then, each process computes the position of its particle. A more realistic example has each process manipulating an array of particles,

a notion we introduce in Section 6.1. Economising on primitive types, we use integer rather than floating point values.

Our implementation builds on a ring topology. We distinguish the root process (process 0) from other processes (processes 1 to size −1). Once again, taking advantage of variable myrank, we write the behaviour of all processes with a single expressions.

```
1  let iterations:int = broadcast 0 read_int() in
2  let x, y, z, v:int ref = read_particle_position_and_velocity();
3  let c_pipe_x, c_pipe_y, c_pipe_z, c_pipe_v:int ref =
4    mkref !x, mkref !y, mkref !z, mkref !v in
5  let n_pipe_x, n_pipe_y, n_pipe_z, n_pipe_v:int ref =
6    mkref 0, mkref 0, mkref 0, mkref 0 in
7  for _ < iterations do
8    for _ < size-1 do
9      if myrank == 0 then
10       send 1 !c_pipe_x; send 1 !c_pipe_y;
11       send 1 !c_pipe_z; send 1 !c_pipe_v;
12       receive size-1 n_pipe_x; receive size-1 n_pipe_y;
13       receive size-1 n_pipe_z; receive size-1 n_pipe_v
14     else
15       receive myrank-1 n_pipe_x; receive myrank-1 n_pipe_y;
16       receive myrank-1 n_pipe_z; receive myrank-1 n_pipe_v;
17       send (myrank+1)%size !c_pipe_x; send (myrank+1)%size!c_pipe_y;
18       send (myrank+1)%size !c_pipe_z; send (myrank+1)%size !c_pipe_v
19     c_pipe_x, c_pipe_y, c_pipe_z, c_pipe_v :=
20       !n_pipe_x, !n_pipe_y, !n_pipe_z, !n_pipe_v;
21     v := compute_forces(...)
22   x, y, z, v := compute_particle_new_position(...)
```

Process 0 reads and broadcasts the number of iterations (line 1). Each process then reads the particle's position and velocity (line 2) and embarks on a refinement loop (lines 7–21). At each step of the loop, process 0 sends and receives (in this order) messages with particle's data (lines 10–13). All other processes invert message passing: they receive first and send then (line 15–18). Notice the modulo- size arithmetic so that process size −1 correctly communicates with process 0.

2.2 The Syntax of Programs

Figure 2 summarises the syntax of programs. Let r, x, y, z range over *variables* (a countable set) and let k, l, m, n, o range over *integer literals*. Let i, j range over *index terms*, which in addition to variables and literals, further include integer arithmetic operations generically denoted as $i \oplus j$ (such as, +, −, +, /, %). Index terms also comprise the standard operations on references: reference creation, mkref i, dereference, $!i$, and update, $i := j$. Rather than introducing a new syntactic category for reference identifiers we use variables, for this simplifies the theory. We nevertheless use identifier r to refer to a (variable that denotes a) reference.

Let p, q range over *propositions* which include the standard boolean operations generically denoted by $p \otimes q$ (such as || and &&), equality and relational integer operations generically denoted by $i \ominus j$ (such as $>$, $<$, $\leq$, $\geq$, $=$, $\neq$). Rather than introducing literals for Boolean values, we distinguish two closed propositions, p and q, such that p can be proved to be true and q cannot be

Integer constants	k, l, m, n, o	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$
Values	u, v, w	::=	$n \mid x$
Index terms	i, j	::=	$v \mid i \oplus i \mid \mathsf{mkref}\ i \mid\ !i \mid i := i$
Propositions	p, q	::=	$p \owedge q \mid i \oslash i$
Datatypes	D, E	::=	$\mathbf{int} \mid \mathbf{bool} \mid \{x : D \mid p\} \mid D\ \mathbf{ref}$
Expressions	e, f	::=	$\mathsf{skip} \mid c \mid e; e \mid \mathsf{for}\ x < i\ \mathsf{do}\ e \mid$
			$\mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let}\ x : D = i\ \mathsf{in}\ e \mid$
			$\mathsf{lets}\ x : D = i\ \mathsf{in}\ e$
Collective exps	c, d	::=	$\mathsf{send}\ i\ i \mid \mathsf{receive}\ i\ i \mid \mathsf{reduce}\ i\ i\ i \mid$
			$\mathsf{let}\ x : D = \mathsf{broadcast}\ i\ i\ \mathsf{in}\ e \mid \mathsf{ifc}\ p\ \mathsf{then}\ e\ \mathsf{else}\ e$
Stores	$\rho, \theta, \sigma, \tau$	::=	$\varepsilon \mid \rho, x := v$
Processes	P, Q	::=	$\langle \rho, e \rangle$
Programs	R, S	::=	$P \mid \cdots \mid P$

Fig. 2. The syntax of programs.

proved to be true. We call true and false to such propostions and denote them collectively by b. The details are in Section 3.2. Let D, E range over *datatypes* which include $\mathbf{int}$ and refinement datatypes $\{x : D \mid p\}$. To handle references, datatypes include a $D\ \mathbf{ref}$ constructor. There are two *distinguished variables*: *size*, denoting the number of parallel processes and *myrank*, denoting the rank number of a given process. Both variables are 0-based, so that process ranks range from 0 to $\mathit{size} - 1$.

Let e, f range over *expressions*. skip denotes the terminated expression. The send and $\mathsf{receive}$ operations both expect two index terms: the first represents the target or the source rank, respectively; the second the value to be sent or the reference where to store the value received, again respectively. The target and the source ranks must be different from *myrank* so that processes do not deadlock when trying to send or receive a message to or from itself. The reduce operation expects three index terms: the first denotes the root process, the second the value to send, and the last the reference where the root process stores the sum of the values. The $\mathsf{broadcast}$ operation requires two index terms: the root process and the value to be broadcast. Variable x denotes the value broadcasted and can be further used in the continuation expression e. The collective conditional, $\mathsf{ifc}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f$ denotes a conditional operation on which all processes must decide equally. The remaining expression constructs are fairly standard: sequential composition, downwards-for loop and conditional expression.

To simplify type checking, we provide two forms of let-expressions. In expressions of the from $\mathsf{let}\ x : D = i\ \mathsf{in}\ e$, index term i cannot contain operations on references. As such its datatype, D, may be an expressive type, containing refinement types. In turn, in expressions of the from $\mathsf{lets}\ x : D = i\ \mathsf{in}\ e$ (the s is for store operations), index term i is arbitrary and hence may contain operations on references. The price to pay is that D may not contain refinement types. Concrete programming languages are expected to present a uniform syntax to programmers (with let only, for example). Compilers can easily convert the surface language into the appropriate form according to the presence or absence of operations on references. That is the approach we follow in all examples in this article.

Apart from the wildcard (_) examples in Section 2.1, lump together multiple let expressions and multiple assignments. Furthermore, expressions of the form $x := i; e$ abbreviate

Index term evaluation $\boxed{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, v\rangle}$

$$\textsc{IE-Size}\ \langle \rho, size\rangle \Downarrow^n_m \langle \rho, n\rangle \qquad \textsc{IE-Myrank}\ \langle \rho, myrank\rangle \Downarrow^n_m \langle \rho, m\rangle$$

$$\textsc{IE-Ref}\ \frac{r \neq size \qquad r \neq myrank \qquad r := v \in \rho}{\langle \rho, r\rangle \Downarrow^n_m \langle \rho, r\rangle}$$

$$\textsc{IE-Int}\ \langle \rho, k\rangle \Downarrow^n_m \langle \rho, k\rangle \qquad \textsc{IE-Op}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, k\rangle \qquad \langle \sigma, j\rangle \Downarrow^n_m \langle \theta, l\rangle}{\langle \rho, i \oplus j\rangle \Downarrow^n_m \langle \theta, k \oplus l\rangle}$$

$$\textsc{IE-Mkref}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, v\rangle \qquad r \text{ fresh}}{\langle \rho, \mathsf{mkref}\ i\rangle \Downarrow^n_m \langle (\sigma, r := v), r\rangle}$$

$$\textsc{IE-Deref}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, r\rangle \qquad r := v \in \sigma}{\langle \rho, !i\rangle \Downarrow^n_m \langle \sigma, v\rangle} \qquad \textsc{IE-Assign}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, r\rangle \qquad \langle \sigma, j\rangle \Downarrow^n_m \langle \theta, v\rangle}{\langle \rho, i := j\rangle \Downarrow^n_m \langle \theta[r{:=}v], v\rangle}$$

Proposition evaluation $\boxed{\langle \rho, p\rangle \Downarrow^n_m \langle \sigma, b\rangle}$

$$\textsc{pE-Rel}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, k\rangle \qquad \langle \sigma, j\rangle \Downarrow^n_m \langle \theta, l\rangle}{\langle \rho, i \circledcirc j\rangle \Downarrow^n_m \langle \theta, k \circledcirc l\rangle} \qquad \textsc{pE-Op}\ \frac{\langle \rho, p_1\rangle \Downarrow^n_m \langle \sigma, b_1\rangle \qquad \langle \sigma, p_2\rangle \Downarrow^n_m \langle \theta, b_2\rangle}{\langle \rho, p_1 \mathbin{⍟} p_2\rangle \Downarrow^n_m \langle \theta, b_1 \mathbin{⍟} b_2\rangle}$$

Process evaluation $\boxed{P \Downarrow^n_m Q}$

$$\textsc{PE-Skip}\ \langle \rho, \mathsf{skip}\rangle \Downarrow^n_m \langle \rho, \mathsf{skip}\rangle \qquad \textsc{PE-Collective}\ \langle \rho, c\rangle \Downarrow^n_m \langle \rho, c; \mathsf{skip}\rangle \qquad \textsc{PE-SemiCol}\ \langle \rho, c; e\rangle \Downarrow^n_m \langle \rho, c; e\rangle$$

$$\textsc{PE-SemiSemi}\ \frac{e \neq c \qquad \langle \rho, e\rangle \Downarrow^n_m \langle \sigma, c; g\rangle}{\langle \rho, e; f\rangle \Downarrow^n_m \langle \sigma, c; g; f\rangle} \qquad \textsc{PE-SemiSkip}\ \frac{\langle \rho, e\rangle \Downarrow^n_m \langle \sigma, \mathsf{skip}\rangle \qquad \langle \sigma, f\rangle \Downarrow^n_m \langle \theta, g\rangle}{\langle \rho, e; f\rangle \Downarrow^n_m \langle \theta, g\rangle}$$

$$\textsc{PE-IfThen}\ \frac{\langle \rho, p\rangle \Downarrow^n_m \langle \sigma, \mathsf{true}\rangle \qquad \langle \sigma, e\rangle \Downarrow^n_m \langle \theta, g\rangle}{\langle \rho, \mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f\rangle \Downarrow^n_m \langle \theta, g\rangle} \qquad \textsc{PE-IfElse}\ \frac{\langle \rho, p\rangle \Downarrow^n_m \langle \sigma, \mathsf{false}\rangle \qquad \langle \sigma, f\rangle \Downarrow^n_m \langle \theta, g\rangle}{\langle \rho, \mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f\rangle \Downarrow^n_m \langle \theta, g\rangle}$$

$$\textsc{PE-ForStep}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \rho, k\rangle \qquad \langle \rho, e\{k-1/x\}; \mathsf{for}\ x < i-1\ \mathsf{do}\ e\rangle \Downarrow^n_m \langle \sigma, f\rangle \qquad k > 0}{\langle \rho, \mathsf{for}\ x < i\ \mathsf{do}\ e\rangle \Downarrow^n_m \langle \sigma, f\rangle}$$

$$\textsc{PE-ForBase}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \rho, 0\rangle}{\langle \rho, \mathsf{for}\ x < i\ \mathsf{do}\ e\rangle \Downarrow^n_m \langle \rho, \mathsf{skip}\rangle} \qquad \textsc{PE-Let}\ \frac{\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, v\rangle \qquad \langle \sigma, e\{v/x\}\rangle \Downarrow^n_m \langle \theta, f\rangle}{\langle \rho, \mathsf{let}\ x\colon D = i\ \mathsf{in}\ e\rangle \Downarrow^n_m \langle \theta, f\rangle}$$

Rule for lets similar to that of let

Fig. 3. Evaluation.

let _: **int** = $x := i$ in e or lets _: **int** = $x := i$ in e as appropriate. Once again, compilers can easily convert the surface language into the core language discussed in this article.

Stores associate reference identifiers r to values v (integer or Boolean values or variables denoting references). *Processes* are pairs composed of a store ρ and an expression e. *Programs* are ordered lists of *size* processes.

2.3 Program Reduction

We introduce index term, proposition, and process evaluation in Figure 3, and program reduction in Figure 4. Notation $\rho[r{:=}v]$ denotes store ρ where the entry for r, of the form $r := v'$, is replaced by $r := v$.

Program reduction $\boxed{R \longrightarrow S}$

PR-Msg

$$\frac{\langle \rho_l, i_l\rangle \Downarrow_l^n \langle \rho_l, m\rangle \qquad \langle \rho_l, j_l\rangle \Downarrow_l^n \langle \sigma_l, k\rangle \qquad \langle \rho_m, i_m\rangle \Downarrow_m^n \langle \rho_m, l\rangle \qquad \langle \rho_m, j_m\rangle \Downarrow_m^n \langle \sigma_m, r\rangle}{S_1 \mid \langle \rho_l, \mathsf{send}\ i_l\ j_l; e_l\rangle \mid S_2 \mid \langle \rho_m, \mathsf{receive}\ i_m\ j_m; e_m\rangle \mid S_3 \longrightarrow S_1 \mid \langle \sigma_l, e_l\rangle \mid S_2 \mid \langle \sigma_m[r{:=}k], e_m\rangle \mid S_3}$$

PR-Red

$$\frac{\langle \rho_k, i_k\rangle \Downarrow_k^n \langle \rho_k, l\rangle \qquad \langle \rho_k, j_k\rangle \Downarrow_k^n \langle \sigma_k, v_k\rangle \qquad \langle \sigma_l, j'_l\rangle \Downarrow_l^n \langle \theta, r\rangle \qquad (k = 0..n-1)}{\langle \rho_k, \mathsf{reduce}\ i_k\ j_k\ j'_k; e_k\rangle_{k=0}^{n-1} \longrightarrow \langle \sigma_k, e_k\rangle_{k=0}^{l-1} \mid \langle \theta[r{:=}v_0 + \cdots + v_{n-1}], e_l\rangle \mid \langle \sigma_k, e_k\rangle_{k=l+1}^{n-1}}$$

PR-BCast

$$\frac{\langle \rho_k, i_k\rangle \Downarrow_k^n \langle \rho_k, l\rangle \qquad \langle \rho_l, j_l\rangle \Downarrow_l^n \langle \sigma, m\rangle \qquad (k = 0..n-1)}{\begin{array}{l}\langle \rho_k, \mathsf{let}\ x\colon D = \mathsf{broadcast}\ i_k\ j_k\ \mathsf{in}\ e_k; e'_k\rangle_{k=0}^{n-1} \longrightarrow \\ \qquad \langle \rho_k, e_k\{m/x\}; e'_k\rangle_{k=0}^{l-1} \mid \langle \sigma, e_l\{m/x\}; e'_l\rangle \mid \langle \rho_k, e_k\{m/x\}; e'_k\rangle_{k=l+1}^{n-1}\end{array}}$$

PR-IfcT

$$\frac{\langle \rho_k, p_k\rangle \Downarrow_k^n \langle \rho_k, \mathsf{true}\rangle \qquad (k = 0..n-1)}{\langle \rho_k, \mathsf{ifc}\ p_k\ \mathsf{then}\ e_k\ \mathsf{else}\ f_k; e'_k\rangle_{k=0}^{n-1} \longrightarrow \langle \rho_k, e_k; e'_k\rangle_{k=0}^{n-1}}$$

PR-Proc

$$\frac{e \neq \mathsf{skip} \qquad e \neq c; e' \qquad \langle \rho, e\rangle \Downarrow_m^n P}{S_1 \mid \langle \rho, e\rangle \mid S_2 \longrightarrow S_1 \mid P \mid S_2}$$

Omitting the rule for receive-send (dual to PR-Msg) and that for ifc-false (dual to PR-IfcT).

Fig. 4. Program reduction.

Index term evaluation takes a store ρ, a term i, the number n of processes and the number m of a given process; it returns a new store σ and a value v. Integer n is used to resolve variable *size*; integer m to resolve variable *myrank*. In rule IE-Mkref it is insufficient to take r from the complement of the domain of σ, for references are plain variables and an unwise choice would clash with some variable in the expression where the index term is taken from. Index term evaluation imposes a strict consistency model in which write and read operation are performed in order [52].

Proposition evaluation takes a store ρ, a proposition p, the number n of processes and the number m of the rank of the process, and returns a new store σ and a simplified expression q. The values of n and m are used to evaluate index terms. The rules should be straightforward.

Process evaluation describes the local behaviour of processes, that is, computations that do not include synchronisation or communication with other processes. In Figure 3, one finds two rules for the conditional, to be used when the guard evaluates to true or to false. The figure further includes rules for loop expansion and termination (to be used when the loop upper bound is larger than 0 and when it is lower than 1, respectively), for let-processes, and for sequential composition. Lemma 4.13 ensures that if $P \Downarrow_m^n \langle \rho, e\rangle$, then e is skip or of the form $(c; f)$ with c a collective operation ready to synchronise at program-level.

Program reduction is in Figure 4. Rule PR-Msg deals with message passing. It requires a send process and a receive process. The send process is l-ranked and that must be the value of the "from" index (i_m) in the receive process. The receive process is m-ranked and that must be the value of the "to" index (i_l) in the send process. The index to be sent (j_l) is evaluated in the send process. The index denoting the reference to hold the incoming value (j_m) is evaluated at the receive process. After reduction, both processes progress to their continuations (e_l and e_m), and the value (v) in the message is stored in the store of the receive process. All other processes remain still. There is an unwritten proviso in rule PR-Msg that program S_1 is of length l, program S_2 of length $m - l - 1$, and program S_3 of length $n - m + 1$.

Rule PR-Red engages all processes. Each process locally evaluates the first index term to identify the root process (l) and the second index term to identify the value to send (v_k). The root process alone evaluates the third index term, to obtain the reference r on which to store the sum of the

values.[1] All processes progress to their continuations (e_k) with stores unchanged, except for the root process that records in r the sum of the values.

PR-BCast is another rule that engages all processes. The first index term (i_k) identifies the root process (l). The root process alone evaluates the second index term (j_k) to obtain the value v to transmit. All processes progress to their continuations with variable x replaced by the transmitted value, that is, $e_k\{v/x\}$.

Rule PR-IfcT requires all processes to evaluate their propositions (p_k) to true. All processes then advance to their then-branches (e_k). The last rule in the figure allows one process (P_l) to progress individually, via local process reduction (Figure 3).

Let us follow a program that computes $\Sigma_{i=5}^{8} i$ by using two processes, three reduce operations, and that leaves the result in reference r_0 at process 0. The expressions for the two processes are as follows:

$$(\mathsf{reduce}\ 1\ 5\ r_0; \mathsf{reduce}\ 1\ 7\ r_0); \mathsf{reduce}\ 0\ 8\ r_0$$
$$\mathsf{reduce}\ 1\ 6\ r_1; (\mathsf{reduce}\ 1\ !r_1\ r_1; \mathsf{reduce}\ 0\ !r_1\ r_1)$$

Considering the stores $\rho_0 \triangleq r_0 := 322$, $\rho_0' \triangleq r_0 := 26$, $\rho_1 \triangleq r_1 := 178$, $\rho_1' \triangleq r_1 := 11$, $\rho_1'' \triangleq r_1 := 18$, we successively have:

$$\langle \rho_0, (\mathsf{reduce}; \mathsf{reduce}); \mathsf{reduce}\rangle \mid \langle \rho_1, \mathsf{reduce}; (\mathsf{reduce}; \mathsf{reduce})\rangle \longrightarrow$$
$$\langle \rho_0, \mathsf{reduce}; ((\mathsf{skip}; \mathsf{reduce}); \mathsf{reduce})\rangle \mid \langle \rho_1, \mathsf{reduce}; (\mathsf{reduce}; \mathsf{reduce})\rangle \longrightarrow$$
$$\langle \rho_0, (\mathsf{skip}; \mathsf{reduce}); \mathsf{reduce}\rangle \mid \langle \rho_1', \mathsf{reduce}; \mathsf{reduce}\rangle \longrightarrow$$
$$\langle \rho_0, \mathsf{reduce}; \mathsf{reduce}\rangle \mid \langle \rho_1', \mathsf{reduce}; \mathsf{reduce}\rangle \longrightarrow$$
$$\langle \rho_0, \mathsf{reduce}\rangle \mid \langle \rho_1'', \mathsf{reduce}\rangle \longrightarrow\longrightarrow$$
$$\langle \rho_0, \mathsf{reduce}; \mathsf{skip}\rangle \mid \langle \rho_1'', \mathsf{reduce}; \mathsf{skip}\rangle \longrightarrow$$
$$\langle \rho_0', \mathsf{skip}\rangle \mid \langle \rho_1'', \mathsf{skip}\rangle$$

where reduction alternates between the rule that allows processes to progress individually (rule PR-Proc) and rule PR-Red. In this case, local reduction simply rearranges the expression in order to expose the next collective operation (reduce). Notice that there is no global synchronisation, except on collective operations; each process locally knows what to do: expose the next collective operation.

3 TYPES FOR MESSAGE PASSING PARALLEL PROGRAMS

This section introduces the notion of types, type equivalence, and type assignment to programs, starting with examples of types for the source code introduced in the previous section.

3.1 Typing Common Parallel Programs

We discuss types for all four examples introduced in Section 2.1.

The Monte Carlo Algorithm. Recall that process 0 broadcasts the number of darts and then collects the sum of darts (in all processes) that hit the target via a reduce operation. Here is a possible type:

```
1 broadcast 0 n:int.
2 reduce 0
```

[1]Variants of the reduce operator can be easily added (e.g., that compute the maximum). Alternatively, the operator could be parametric on the reducing function at the expense of requiring support for function types.

Section 2.1 introduces three expressions: for the root process, for all non-root processes, and for all processes (including the root). The type above characterises the interactive behaviour of all these expressions, given that the broadcast / reduce pattern is common to all three. Variable n denotes the value broadcasted and could be used in the rest of the protocol since it denotes a value equally known to all ranks. This dependence is discussed in the following variant, where process 0 further chooses (and broadcasts) the sink, the process that must collect the number of hits.

```
1 broadcast 0 n:int.
2 broadcast 0 sink:rank.
3 reduce sink
```

Notice that the value of the sink, broadcast by process 0 to all processes including itself, influences the rest of the protocol: all processes agree to reduce on this value. Datatype **rank** abbreviates a natural number smaller than size and could be written as {x : **nat** | x< size }, with **nat** (another handy type) being {x : **int** | x≥0}.

Message Exchange on a Ring Topology. This example highlights message passing between processes. A type for the code of each process is as follows:

```
1 forall i<size.
2   message (size−i−1) (size−i)%size
```

Each individual process send one message and receives another, so that exactly size messages are exchanged in total. This fact is captured by the **forall** i< size type, where i ranges from size −1 down to 0. In this case, each "step" in the **forall** type describes one message exchange between two different processes. Because size messages are exchanged and there are size processes, we expect to see exactly one send and one receive operation in the code of each process. Referring to the code in Section 2.1 we can see that *this* **forall** type does not correspond to any loop in the source code.

Messages are exchanged to the right in all cases but one: rank 0 sends to rank 1, which in turn sends to rank 2, ...which sends to rank size −1 which sends to rank 0. The "wrap-around" happens for i = 0 in which case a message is sent to the left. This reversal with respect to the receive-first-send-then pattern of all other processes breaks the potential communication deadlock and is in clear sync with the code.

Odd-even Sort. The type associated with the algorithm can be easily derived.

```
1 forall phase < size.
2   ifc even(phase) then
3     forall i<size/2
4       message 2*i   2*i+1;
5       message 2*i+1 2*i
6   else // odd(phase)
7     forall i < (size−1)/2
8       message 2*i+1 2*i+2;
9       message 2*i+2 2*i+1
```

The outer **forall** type describes the number of iterations in the program and matches the for loop in line 2 of the program. In each phase messages are either sent left or right, but this decision must be unanimously taken among all processes, otherwise a deadlock will happen. The decision is based on the phase, a variable that is introduced by the **forall** type (and the for loop

in the program) and is hence common to all ranks. As such, both the type and the program may use the **ifc** / ifc constructor. Notice that the for / ifc dependency on program variable phase (lines 2–3 in the source code) is shared by the **forall** / **ifc** part of the type (lines 1–2). Each branch features a further **forall** to state that each processes must exchange two messages. This type, however, does not correspond to a loop in the program, as discussed in the example above.

A little arithmetic will show that the indices in the **message** type correspond to those in the send and receive operators in the program. From the type one may infer the number of messages exchanged in the program: size in the outer loop times 2 * size / 2 (in the worst case) in the inner loop, that is size * size , as expected.

The n-Body Problem. The type below captures the interactive behaviour of the code for the *n*-body problem in Section 2.1.

```
1 broadcast 0 iterations : int.
2 forall _ < iterations.
3   forall _ < size-1.
4     forall i < size.
5       message (size-i-1) (size-i)%size;
6       message (size-i-1) (size-i)%size;
7       message (size-i-1) (size-i)%size;
8       message (size-i-1) (size-i)%size
```

One can easily notice the alignment of the type against the code: type **broadcast** corresponds to the broadcast operation in the source code (line 1); the **forall** type controlling the number of iterations corresponds to the for loop in line 7 and that controlling the pipe corresponds to for loop in line 8. The inner **forall** type (in line 4 above) simply says that each of the size processes is to exchange four messages (lines 5–8 above), and does not correspond to any loop in the source code, as discussed in the examples above. The modulo- size arithmetic enforces the reversal of rank 0 with respect to the receive-first-send-then pattern of all other processes, thus breaking the communication deadlock.

3.2 Type Formation

The syntax of datatypes, propositions and index terms are in Figure 2; the associated intuitions are discussed in Section 2. Figure 5 introduces the formation rules for all these syntactic categories. In addition, it introduces typing contexts as lists of $x : D$ binds. The syntax of types and the type formation rules are in Figure 6.

Metavariables T, U range over types. The type of terminated processes is **skip**. Sequential composition is denoted by $T; U$. The primitive recursion operator is $\forall\ x{<}i.T$ denoting the sequential composition of i copies of type T (modulo a substitution introduced below). Point-to-point communication is denoted by type **message** $i\ j$ describing a message from the process at rank i to the process at rank j. Reduce operations are introduced by type **reduce** i whose value is to be collected by the process at rank i. Broadcast operations are denoted by **broadcast** $i\ x : D.T$, issued from process rank i and distributing a value of datatype D denoted by variable x in the continuation type T. Finally, collective conditional types are denoted by **ifc** p **then** T **else** U which behave as T or U depending on the truth value of proposition p.

A few constructors introduce *bindings* for type variables. In datatype $\{x : D \mid p\}$, variable x is bound in proposition p. In types $\forall\ x{<}i.T$ and **broadcast** $i\ x : D.T$, variable x is bound in type T. The set of free variables in object A, denoted by $\mathrm{fv}(A)$, is defined accordingly.

Contexts $\quad \Gamma, \Delta \;::=\; \varepsilon \mid \Gamma, x : D$

Index term formation $\quad \boxed{\Gamma \vdash i : D}$

$$\textsc{I-Var}\ \frac{\Gamma\ \mathbf{context} \qquad x : D \in \Gamma}{\Gamma \vdash x : D} \qquad \textsc{I-Int}\ \frac{\Gamma\ \mathbf{context}}{\Gamma \vdash n : \mathbf{int}} \qquad \textsc{I-Op}\ \frac{\Gamma \vdash i : \mathbf{int} \qquad \Gamma \vdash j : \mathbf{int}}{\Gamma \vdash i \oplus j : \mathbf{int}}$$

$$\textsc{I-Refin}\ \frac{\Gamma \vdash i : D \qquad \Gamma, x : D \vdash p\ \mathbf{prop}}{\Gamma \vdash i : \{x : D \mid p\}} \qquad \textsc{I-Sub}\ \frac{\Gamma \vdash i : D \qquad \Gamma \vdash D <: E}{\Gamma \vdash i : E}$$

Proposition formation $\quad \boxed{\Gamma \vdash p\ \mathbf{prop}}$

$$\textsc{P-Rel}\ \frac{\Gamma \vdash i : \mathbf{int} \qquad \Gamma \vdash j : \mathbf{int}}{\Gamma \vdash i \lesseqgtr j\ \mathbf{prop}} \qquad \textsc{P-Op}\ \frac{\Gamma \vdash p\ \mathbf{prop} \qquad \Gamma \vdash q\ \mathbf{prop}}{\Gamma \vdash p \owedge q\ \mathbf{prop}}$$

Datatype formation $\quad \boxed{\Gamma \vdash D\ \mathbf{dtype}}$

$$\textsc{D-Int}\ \frac{\Gamma\ \mathbf{context}}{\Gamma \vdash \mathbf{int}\ \mathbf{dtype}} \qquad \textsc{D-Refin}\ \frac{\Gamma, x : D \vdash p\ \mathbf{prop}}{\Gamma \vdash \{x : D \mid p\}\ \mathbf{dtype}}$$

Context formation $\quad \boxed{\Gamma\ \mathbf{context}}$

$$\textsc{C-Empty}\ \frac{}{\varepsilon\ \mathbf{context}} \qquad \textsc{C-Bind}\ \frac{\Gamma\ \mathbf{context} \qquad \Gamma \vdash D\ \mathbf{dtype} \qquad x \notin \Gamma, \mathrm{fv}(D)}{\Gamma, x : D\ \mathbf{context}}$$

Proposition entailment $\quad \boxed{\Gamma \vdash p\ \mathbf{true}}$

$$\frac{\Gamma \vdash p\ \mathbf{prop} \qquad \Gamma \models p}{\Gamma \vdash p\ \mathbf{true}}$$

Datatype subtyping and equivalence $\quad \boxed{\Gamma \vdash D <: D} \quad \boxed{\Gamma \vdash D \equiv D}$

$$\textsc{S-Int}\ \frac{\Gamma\ \mathbf{context}}{\Gamma \vdash \mathbf{int} <: \mathbf{int}} \qquad \textsc{S-RefinL}\ \frac{\Gamma \vdash D <: E \qquad \Gamma, x : D \vdash p\ \mathbf{prop}}{\Gamma \vdash \{x : D \mid p\} <: E} \qquad \textsc{S-RefinR}\ \frac{\Gamma \vdash D <: E \qquad \Gamma, x : D \vdash p\ \mathbf{true}}{\Gamma \vdash D <: \{x : E \mid p\}}$$

$$\textsc{DE-Sub}\ \frac{\Gamma \vdash D <: E \qquad \Gamma \vdash E <: D}{\Gamma \vdash D \equiv E}$$

Fig. 5. Datatype, proposition, index term, and context formation rules; datatype subtyping and equivalence.

The distinguished variable *size* belongs to datatype $\{x : \mathbf{int} \mid x \geq 0\}$. Contexts Γ, Δ keep track of variables and their datatypes. The language of datatypes allows defining a few important derived constructors, including **nat** standing for the datatype $\{x : \mathbf{int} \mid x \geq 0\}$, and **rank** abbreviating datatype $\{x : \mathbf{nat} \mid x < \mathit{size}\}$. As discussed in Section 2.2, true is a distinguished proposition such that $\vdash$ true **true** holds, and false is another distinguished proposition such that $\vdash$ false **true** does *not* hold.

All these notions—index terms, propositions, datatypes, types, and contexts—are subject to formation rules that discard non well formed syntax. The rules should be self-explanatory. We essentially require that all variables (in index terms, propositions or datatypes) are declared in the

Types $T, U, V ::= \mathbf{skip} \mid \mathbf{message}\ i\ i \mid \mathbf{reduce}\ i \mid \forall\, x{<}i.T \mid T;T \mid \mathbf{broadcast}\ i\ x{:}\ D.T \mid \mathbf{ifc}\ p\ \mathbf{then}\ T\ \mathbf{else}\ T$

Type formation $\boxed{\Gamma \vdash T\ \mathbf{type}}$

$$\textsc{T-Skip}\ \frac{\Gamma\ \mathbf{context}}{\Gamma \vdash \mathbf{skip}\ \mathbf{type}} \qquad \textsc{T-Msg}\ \frac{\Gamma \vdash i : \mathbf{rank} \quad \Gamma \vdash j : \mathbf{rank} \quad \Gamma \vdash i \neq j\ \mathbf{true}}{\Gamma \vdash \mathbf{message}\ i\ j\ \mathbf{type}}$$

$$\textsc{T-Red}\ \frac{\Gamma \vdash i : \mathbf{rank}}{\Gamma \vdash \mathbf{reduce}\ i\ \mathbf{type}} \qquad \textsc{T-Bcast}\ \frac{\Gamma \vdash i : \mathbf{rank} \quad \Gamma, x{:}\ D \vdash T\ \mathbf{type}}{\Gamma \vdash \mathbf{broadcast}\ i\ x{:}\ D.T\ \mathbf{type}}$$

$$\textsc{T-Semi}\ \frac{\Gamma \vdash T\ \mathbf{type} \quad \Gamma \vdash U\ \mathbf{type}}{\Gamma \vdash T;U\ \mathbf{type}} \qquad \textsc{T-All}\ \frac{\Gamma, x{:}\ \{y{:}\ \mathbf{nat} \mid y < i\} \vdash T\ \mathbf{type} \quad y \notin \mathrm{fv}(i)}{\Gamma \vdash \forall\, x{<}i.T\ \mathbf{type}}$$

$$\textsc{T-Ifc}\ \frac{\Gamma \vdash p\ \mathbf{prop} \quad \Gamma, \_{:}\ \{p\} \vdash T\ \mathbf{type} \quad \Gamma, \_{:}\ \{\neg p\} \vdash U\ \mathbf{type}}{\Gamma \vdash \mathbf{ifc}\ p\ \mathbf{then}\ T\ \mathbf{else}\ U\ \mathbf{type}}$$

Fig. 6. Type formation rules.

context. For example, $\{x{:}\ \mathbf{int} \mid x = y\}$ is not a datatype when considered under the empty context, ε, but becomes so if one chooses a context of the form $y{:}\ \mathbf{int}$.

The formation rules for types are new and deserve a brief explanation. All type constructors that use index terms i to talk about process ranks require i to be a valid rank. That is the case of **message**, **reduce**, and **broadcast**. In addition, rule T-Msg requires the source and the target rank, i and j, to be distinct, for an attempt to send a message to itself would lead to a deadlocked situation. Types **message** and **reduce** do not explicitly mention the value transmitted, for programs cannot rely on this information. The case for **broadcast** is different: the rest of the protocol may rely on the value transmitted.

Proposition entailment includes an hypothesis of the form $\Gamma \vDash p$ that refers to logical deducibility. This proof obligation is usually passed to an SMT solver. A few assumptions are required on deducibility, namely on what concerns subtyping, weakening, strengthening, and substitution (Lemmas 3.1, 4.1, 4.2, and 4.4). The following assumption are expected to hold for most SMT solvers.

Tautology $\Gamma, x{:}\ D \vdash p\ \mathbf{prop}$ implies $\Gamma, x{:}\ \{y{:}\ D \mid p\{y/x\}\} \vDash p$;
Weakening $\Gamma \vDash p$ implies $\Gamma, x{:}\ D \vDash p$;
Strengthening $\Gamma, x{:}\ D \vDash p$ and $x \notin \mathrm{fv}(p)$ and $\mathrm{fv}(D) = \emptyset$ imply $\Gamma \vDash p$;
Substitution $\Gamma \vdash i : D$ and $\Gamma, x{:}\ D, \Delta \vDash p$ imply $\Gamma, \Delta\{i/x\} \vDash p\{i/x\}$.

Decidability of logical deducibility is discussed together with algorithmic type checking (Section 5.2).

Datatype subtyping is defined as usual for refinement types [26], and equivalence is defined from subtyping again as usual. Informally, a refinement type D is a subtype of a refinement type E if the formulas in D are "more precise" than those in E. In rule S-RefinL, proposition p appears in the "more precise" side, hence we require only its good formation. In the case of rule S-RefinR, proposition p appears in the "less precise" side, hence we require p to hold (which in turn implies that it is well formed).

We conclude this section with an example attesting the importance of type formation. Consider a program where all processes are to send some integer value to a fixed process. Process 0 decides and broadcasts the rank of this process. One could try a type of the form:

```
broadcast 0 n:rank.
forall i<size.
  message i n
```

but it would not be well formed, for rank n sends a message to itself, which would lead to a deadlocked situation. Sketching a derivation for the formation of the type above, one stumbles at an unsatisfiable premise to rule T-Msg, namely:

$$size\colon \{x\colon \mathbf{int} \mid x > 1\}, n\colon \{y\colon \mathbf{int} \mid 0 \le y < size\}, i\colon \{z\colon \mathbf{int} \mid z \le size\} \vdash i \neq n\ \mathbf{true}$$

which yields the unsatisfiable verification condition $(size > 1 \land 0 \le n < size \land i \le size) \longrightarrow i \neq n$.

We can fix this type in a number of different ways. But before we do that, we introduce two handy derived constructs. Notation `forall i : j .. k. T` abbreviates the type `forall i < k-j+1. U`, where type U is obtained from type T via the appropriate translation on index `i`. Notation `ifc p then T` abbreviates the type `ifc p then T else skip`.

So, here is one of the possible fixes, a solution using two loops:

```
broadcast 0 n:rank.
forall i: 0 .. n-1.
  message i n
forall i: n+1 .. size-1.
  message i n
```

and another that makes use of a collective conditional:

```
broadcast 0 n:rank.
forall i < size.
  ifc i != n then
    message i n
```

3.3 Type Equivalence

The complete set of rules is in Figure 7. Premises in parenthesis are used solely for agreement purposes (Lemma 4.3). Recall that variable *myrank* denotes the rank of a given process. The variable is of datatype **rank** and should be present in the typing context, if ever used in a derivation. Type equivalence rules can be grouped under six different categories.

Congruence Rules. Aligns propositions, index types, and datatypes in types, allowing, for example, to show the following equivalences.

$$size\colon \mathbf{nat} \vdash \mathbf{ifc}\ size \ge 1\ \mathbf{then\ skip\ else\ reduce}\ 0 \equiv \mathbf{ifc}\ \neg(0 \ge size)\ \mathbf{then\ skip\ else\ reduce}\ 0$$

$$size = 2 \vdash \mathbf{message}\ 0\ 1 \equiv \mathbf{message}\ 0\ (size - 1)$$

$$size\colon \mathbf{nat} \vdash \mathbf{reduce}\ \{x\colon \mathbf{int} \mid 0 \le x < size\} \equiv \mathbf{reduce}\ \{x\colon \{y\colon \mathbf{int} \mid 0 \le y\} \mid x < size\}$$

The group includes the first seven rules in Figure 7, one for each of the seven type constructors in basic ParTypes (Figure 6). In all these rules, we require propositions, index terms, and datatypes to be equivalent. In rules that introduce new entries in the yping context (broadcast), we push one of the datatypes into the context. This group of rules ensures that type equivalence is reflexive.

Monoidal Rules. Turns the set of types into a monoid with respect to sequential composition and the terminated type. Allows to show that:

$$size \ge 1 \vdash \mathbf{reduce}\ 0; (\mathbf{skip}; \mathbf{reduce}\ 1) \equiv \mathbf{reduce}\ 0; \mathbf{reduce}\ 1$$

Type equivalence $\boxed{\Gamma \vdash T \equiv T}$

$$\textsc{Eq-Skip}\ \frac{(\Gamma\ \textbf{context})}{\Gamma \vdash \textbf{skip} \equiv \textbf{skip}} \qquad \textsc{Eq-Red}\ \frac{\Gamma \vdash i = j\ \textbf{true} \qquad (\Gamma \vdash i : \textbf{rank})}{\Gamma \vdash \textbf{reduce}\ i \equiv \textbf{reduce}\ j}$$

$$\textsc{Eq-Msg}\ \frac{\Gamma \vdash i = k\ \textbf{true} \qquad \Gamma \vdash j = l\ \textbf{true} \qquad (\Gamma \vdash i : \textbf{rank}) \qquad (\Gamma \vdash j : \textbf{rank})}{\Gamma \vdash \textbf{message}\ i\ j \equiv \textbf{message}\ k\ l}$$

$$\textsc{Eq-Bcast}\ \frac{\Gamma \vdash i = j\ \textbf{true} \qquad \Gamma \vdash D \equiv E \qquad \Gamma, x : E \vdash T \equiv U \qquad (\Gamma \vdash i : \textbf{rank})}{\Gamma \vdash \textbf{broadcast}\ i\ x : D.T \equiv \textbf{broadcast}\ j\ x : E.U}$$

$$\textsc{Eq-Semi}\ \frac{\Gamma \vdash T \equiv V \qquad \Gamma \vdash U \equiv W}{\Gamma \vdash T; U \equiv V; W} \qquad \textsc{Eq-Ifc}\ \frac{\Gamma \vdash p \Leftrightarrow q\ \textbf{true} \qquad \Gamma \vdash T \equiv V \qquad \Gamma \vdash U \equiv W}{\Gamma \vdash \textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U \equiv \textbf{ifc}\ q\ \textbf{then}\ V\ \textbf{else}\ W}$$

$$\textsc{Eq-All}\ \frac{\Gamma \vdash i = j\ \textbf{true} \qquad \Gamma, x : \{y : \textbf{nat} \mid y < i\} \vdash T \equiv U \qquad y \notin \mathrm{fv}(i)}{\Gamma \vdash \forall\ x{<}i.T \equiv \forall\ x{<}j.U}$$

$$\textsc{Eq-AllUnfold}\ \frac{\Gamma \vdash i > 0\ \textbf{true} \qquad \Gamma, x : \{y : \textbf{nat} \mid y < i\} \vdash T \equiv U \qquad y \notin \mathrm{fv}(i)}{\Gamma \vdash \forall\ x{<}i.T \equiv U\{i-1/x\}; \forall\ x{<}i-1.U}$$

$$\textsc{Eq-AllSkip}\ \frac{\Gamma \vdash i = 0\ \textbf{true} \qquad (\Gamma, x : \{y : \textbf{nat} \mid y < i\} \vdash T\ \textbf{type}) \qquad (y \notin \mathrm{fv}(i))}{\Gamma \vdash \forall\ x{<}i.T \equiv \textbf{skip}}$$

$$\textsc{Eq-NeutralR}\ \Gamma \vdash T; \textbf{skip} \equiv T \qquad \textsc{Eq-NeutralL}\ \Gamma \vdash \textbf{skip}; T \equiv T \qquad \textsc{Eq-Assoc}\ \Gamma \vdash (T; U); V \equiv T; (U; V)$$

$$\textsc{Eq-IfcL}\ \frac{\Gamma \vdash p\ \textbf{true} \qquad (\Gamma \vdash U\ \textbf{type})}{\Gamma \vdash \textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U \equiv T} \qquad \textsc{Eq-IfcR}\ \frac{\Gamma \vdash \neg p\ \textbf{true} \qquad (\Gamma \vdash T\ \textbf{type})}{\Gamma \vdash \textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U \equiv U}$$

$$\textsc{Eq-Proj}\ \frac{\Gamma \vdash i \neq \mathit{myrank}\ \textbf{true} \qquad \Gamma \vdash j \neq \mathit{myrank}\ \textbf{true} \qquad (\Gamma \vdash i : \textbf{rank}) \qquad (\Gamma \vdash j : \textbf{rank}) \qquad (\Gamma \vdash i \neq j\ \textbf{true})}{\Gamma \vdash \textbf{message}\ i\ j \equiv \textbf{skip}}$$

$$\textsc{Eq-Sym}\ \frac{\Gamma \vdash T \equiv U}{\Gamma \vdash U \equiv T} \qquad \textsc{Eq-Trans}\ \frac{\Gamma \vdash T \equiv U \qquad \Gamma \vdash U \equiv V}{\Gamma \vdash T \equiv V}$$

Fig. 7. Type equivalence.

Primitive Recursion Rules. Allows, for example, to show the following equalities:

$$\mathit{size} \geq 5 \vdash \forall\ x{<}5.\textbf{reduce}\ x \equiv \textbf{reduce}\ 4; \textbf{reduce}\ 3; \forall\ x{<}3.\textbf{reduce}\ x.$$

$$\mathit{size} = 3 \vdash \forall\ x{<}\mathit{size}.\textbf{message}\ x\ (x+1)\%\mathit{size} \equiv \textbf{message}\ 2\ 3; \textbf{message}\ 1\ 2; \textbf{message}\ 0\ 1; \textbf{skip}$$

There are two rules that define the semantics of primitive recursion. The former unfolds the type, making use of sequential composition; the latter rewrites primitive recursion into **skip** then the

limit, i, is 0. Primitive recursion unrolls loops at the left. Unless loops can be fully unrolled, right unroll is not valid. Examples:

$$n = 5, size > n \vdash \forall\, x{<}n.\mathbf{reduce}\ x \equiv (\forall\, x{<}n-1.\mathbf{reduce}\ x+1); \mathbf{reduce}\ 0$$
$$n\colon \mathbf{int}, size > n \nvdash \forall\, x{<}n.\mathbf{reduce}\ x \equiv (\forall\, x{<}n-1.\mathbf{reduce}\ x+1); \mathbf{reduce}\ 0$$

Collective Conditional Rules. Allow simplifying a conditional type when its condition can be proved or disproved. For example, type

$$\forall\, i{<}size-2.(\mathbf{ifc}\ i\%2 = 0\ \mathbf{then}\ \mathbf{message}\ i\ (size-1)\ \mathbf{else}\ \mathbf{skip})$$

can be equated to a sequence of two messages $\mathbf{message}\ 2\ 4; \mathbf{message}\ 0\ 4$ under a context that entails $size = 5$.

Projection Rule. Implements *projection* in the sense of multiparty session types [39]. This is where the distinguished variable *myrank* comes into play. The projection rule rewrites a message type into **skip** if both the target and the source can be proved to be different from *myrank*. For example:

$$size \geq 3, myrank = 1 \vdash \mathbf{message}\ 2\ 3 \equiv \mathbf{skip}$$

Preorder Rules. Include the symmetry and transitivity rules. Reflexivity is attained via the congruence rules. Notice that transitivity is not derivable from the other rules. We have $\Gamma \vdash \forall\, x{<}i.\mathbf{skip} \equiv \mathbf{skip}$ and $\Gamma \vdash \mathbf{skip} \equiv \mathbf{message}\ 0\ 1$ under an appropriate Γ, but cannot show $\Gamma \vdash \forall\, x{<}i.\mathbf{skip} \equiv \mathbf{message}\ 0\ 1$ without the rule for transitivity.

We can easily show that what we call datatype and type equivalence are indeed equivalence relations.

Lemma 3.1 (Subtyping and Equivalence).

(1) $\Gamma \vdash D <: E$ *is a preorder.*
(2) $\Gamma \vdash D \equiv E$ *is an equivalence relation.*
(3) $\Gamma \vdash T \equiv U$ *is an equivalence relation.*

Proof. *Item 1.* Reflexivity follows by rule induction on datatype formation, using the tautology assumption on deducibility (Section 3.2). Transitivity holds by nested rule induction on datatype formation; there are quite a few cases, but they all follow the pattern: read premises, apply induction, conclude with formation rule. *Item 2.* From item 1 and the definition of $\Gamma \vdash D \equiv E$. *Item 3.* The first seven rules establish reflexivity by rule induction. We highlight the case of Eq-Bcast. Agreement for type formation (Lemma 4.3) yields $\Gamma \vdash i : \mathbf{rank}$ (from which we get $\Gamma \vdash i = i\ \mathbf{true}$), $\Gamma, x\colon D \vdash T\ \mathbf{type}$ (and, by induction, $\Gamma, x\colon D \vdash T \equiv T$), and $\Gamma \vdash D <: \mathbf{int}$ (and by agreement followed by item 2, $\Gamma \vdash D \equiv D$). The result follows by rule Eq-Bcast. Symmetry and transitivity are directly embedded in the typing rules. □

One might wonder why we have not lifted subtyping from datatypes to types, and decided instead to define an equality relation on types. Languages with subtyping and some form of input/output always behave co/contravariantly with respect to subtyping. This is the case with functions, where the $T \to U$ type constructor is contravariant in the input position T and covariant in the output position U [62], and with binary session types [33, 37], where the message types $?T.S$ and $!U.S$ are covariant in the input position T and contravariant in output position U [23]. Our types do not talk about input/output. Rather than talking about message-sending and message-reception (as is the case with binary session types), a type $\mathbf{message}\ i\ j$ describes a message *exchanged* between two processes i and j, taking no side (as is the case with global session types [38, 39]). Any attempt to subtype a message type would violate safe substitutability either on the sending or on the receiving side.

Store-related index term formation $\boxed{\Gamma \vdash_s i : D}$

$$\textsc{IS-Var}\ \frac{\Gamma\ \mathbf{context} \qquad x : D \in \Gamma}{\Gamma \vdash_s x : D} \qquad \textsc{IS-Int}\ \frac{\Gamma\ \mathbf{context}}{\Gamma \vdash_s n : \mathbf{int}} \qquad \textsc{IS-Op}\ \frac{\Gamma \vdash_s i : \mathbf{int} \qquad \Gamma \vdash_s j : \mathbf{int}}{\Gamma \vdash_s i \oplus j : \mathbf{int}}$$

$$\textsc{IS-Mkref}\ \frac{\Gamma \vdash_s i : D}{\Gamma \vdash_s \mathsf{mkref}\ i : D\ \mathbf{ref}} \qquad \textsc{IS-Deref}\ \frac{\Gamma \vdash_s i : D\ \mathbf{ref}}{\Gamma \vdash_s !i : D} \qquad \textsc{IS-Assign}\ \frac{\Gamma \vdash_s i : D\ \mathbf{ref} \qquad \Gamma \vdash_s j : D}{\Gamma \vdash_s i := j : D}$$

Store-related proposition formation $\boxed{\Gamma \vdash_s p\ \mathbf{prop}}$

$$\textsc{PS-Rel}\ \frac{\Gamma \vdash_s i : \mathbf{int} \qquad \Gamma \vdash_s j : \mathbf{int}}{\Gamma \vdash_s i \circledcirc j\ \mathbf{prop}} \qquad \textsc{PS-Op}\ \frac{\Gamma \vdash_s p\ \mathbf{prop} \qquad \Gamma \vdash_s q\ \mathbf{prop}}{\Gamma \vdash_s p \circledwedge q\ \mathbf{prop}}$$

Datatype formation and subtyping $\boxed{\Gamma \vdash D\ \mathbf{dtype}}$ $\boxed{\Gamma \vdash D <: D}$

$$\textsc{D-Ref}\ \frac{\Gamma \vdash D\ \mathbf{dtype}}{\Gamma \vdash D\ \mathbf{ref}\ \mathbf{dtype}} \qquad \textsc{S-Ref}\ \frac{\Gamma \vdash D \equiv E}{\Gamma \vdash D\ \mathbf{ref} <: E\ \mathbf{ref}}$$

Fig. 8. Store-related index term and proposition formation; datatype formation and subtyping.

Flexible as it is, type equality is still quite intensional. We have seen above that right loop-unroll does not yield an equivalent type. Another instance is the example at the end of Section 3.2, where the two alternative types (two loops in a row and a loop with a conditional) are not equivalent, yet they describe the same pattern of interaction among processes. A third example are the two types **message** 0 1;**message** 2 3 and **message** 2 3;**message** 1 0 that denote two independent messages exchanges (and that may happen in any order or even simultaneously), yet they are not equivalent.

3.4 Program Formation

Expression formation is in Figure 9. Syntax $\{p\}$ abbreviates datatype $\{\_ : \mathbf{int} \mid p\}$ [26], and is used in the rule for the collective conditional to push a proposition into the context. The send and the receive operators both expect two index terms: the first represents the target or the source rank, respectively; the second the value to be sent or the reference where to store the value received, again respectively. The target and the source ranks must be different from *myrank* as witnessed by premise $\Gamma \vdash i \neq myrank\ \mathbf{true}$ where, we recall, *myrank* is a distinguished variable that denotes the rank of the process. The formation of the reduce operation follows the same principles.

The broadcast operation requires two index terms: the root process i and the value j to be broadcast. Variable x denotes the value broadcasted and can be further used both in the continuation expression e and in type T, hence it is added to the context when checking the formation of e.

In the broadcast operation, the value of index term j is shared among all processes, so that these may use it (through variable x) in the continuation. The same does not apply receive or reduce, where the value exchanged is known only to a subset of the processes, and therefore cannot be used in types. We must make sure that values exchanged among processes are not references; we use judgement $\Gamma \vdash i : D$ for this purpose.

The rules for the collective conditional E-Ifc, copy p or $\neg p$ to the context before checking the then and the else branch. The rule for the conventional conditional, E-If, is standard. Rules E-All and E-Let introduce in the context an entry for the bound variable x with the appropriate refinement type in each case. Rule E-LetS simply introduces type D (which may contain **ref**-types) in

Expression formation $\boxed{\Gamma \vdash e : T}$

$$\textsc{E-Skip}\ \frac{\Gamma\ \textbf{context}}{\Gamma \vdash \mathsf{skip} : \textbf{skip}}$$

$$\textsc{E-Send}\ \frac{\Gamma \vdash \mathit{myrank} : \textbf{rank} \quad \Gamma \vdash i : \textbf{rank} \quad \Gamma \vdash i \neq \mathit{myrank}\ \textbf{true} \quad \Gamma \vdash_{\mathsf{s}} j : \textbf{int}}{\Gamma \vdash \mathsf{send}\ i\ j : \textbf{message}\ \mathit{myrank}\ i}$$

$$\textsc{E-recv}\ \frac{\Gamma \vdash i : \textbf{rank} \quad \Gamma \vdash \mathit{myrank} : \textbf{rank} \quad \Gamma \vdash i \neq \mathit{myrank}\ \textbf{true} \quad \Gamma \vdash_{\mathsf{s}} j : \textbf{int ref}}{\Gamma \vdash \mathsf{receive}\ i\ j : \textbf{message}\ i\ \mathit{myrank}}$$

$$\textsc{E-Red}\ \frac{\Gamma \vdash i : \textbf{rank} \quad \Gamma \vdash_{\mathsf{s}} j : \textbf{int} \quad \Gamma \vdash_{\mathsf{s}} k : \textbf{int ref}}{\Gamma \vdash \mathsf{reduce}\ i\ j\ k : \textbf{reduce}\ i}$$

$$\textsc{E-Bcast}\ \frac{\Gamma \vdash i : \textbf{rank} \quad \Gamma \vdash_{\mathsf{s}} j : D \quad \Gamma, x\colon D \vdash e : T}{\Gamma \vdash \mathsf{let}\ x\colon D = \mathsf{broadcast}\ i\ j\ \mathsf{in}\ e : \textbf{broadcast}\ i\ x\colon D.T}$$

$$\textsc{E-Ifc}\ \frac{\Gamma, \_\colon \{p\} \vdash e : T \quad \Gamma, \_\colon \{\neg p\} \vdash f : U}{\Gamma \vdash \mathsf{ifc}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f : \textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U} \qquad \textsc{E-Semi}\ \frac{\Gamma \vdash e : T \quad \Gamma \vdash f : U}{\Gamma \vdash e; f : T; U}$$

$$\textsc{E-All}\ \frac{\Gamma, x\colon \{y\colon \textbf{nat} \mid y < i\} \vdash e : T \quad y \notin \mathrm{fv}(i)}{\Gamma \vdash \mathsf{for}\ x < i\ \mathsf{do}\ e : \forall\, x{<}i.T} \qquad \textsc{E-If}\ \frac{\Gamma \vdash_{\mathsf{s}} p\ \textbf{prop} \quad \Gamma \vdash e : T \quad \Gamma \vdash f : T}{\Gamma \vdash \mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f : T}$$

$$\textsc{E-Let}\ \frac{\Gamma \vdash i : D \quad \Gamma, x\colon \{y\colon D \mid y = i\} \vdash e : T \quad x \notin \mathrm{fv}(T) \quad y \notin \mathrm{fv}(i)}{\Gamma \vdash \mathsf{let}\ x\colon D = i\ \mathsf{in}\ e : T}$$

$$\textsc{E-LetS}\ \frac{\Gamma \vdash_{\mathsf{s}} i : D \quad \Gamma, x\colon D \vdash e : T \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash \mathsf{lets}\ x\colon D = i\ \mathsf{in}\ e : T} \qquad \textsc{E-Eq}\ \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \equiv U}{\Gamma \vdash e : U}$$

Fig. 9. Expression formation.

the context. In rule E-All, variable x may appear in T because $\forall\, x{<}i.T$ introduces a binding for it. This is not the case for the two let-rules and so we require $x \notin \mathrm{fv}(T)$. Finally, rule E-Eq includes type equivalence in expression formation.

The following example shows a typical usage of rule E-Eq. Consider an expression e of the form $\mathsf{let}\ x\colon \textbf{rank} = i\ \mathsf{in}\ \mathsf{reduce}\ x\ 5\ r$ and take a context Γ such that $\Gamma \vdash i : \textbf{rank}$ and $\Gamma \vdash_{\mathsf{s}} r : \textbf{int ref}$. Then, using rules E-Red and E-Let, we know that $\Gamma, x\colon \{y\colon \textbf{rank} \mid y = i\} \vdash \mathsf{reduce}\ x\ 5\ r : \textbf{reduce}\ x$, but we cannot conclude that e has type $\textbf{reduce}\ x$ for x is bound in the let-expression (and not present in the final context). Instead, between the two rules, we use rule E-Eq to conclude $\Gamma, x\colon \{y\colon \textbf{rank} \mid y = i\} \vdash \mathsf{reduce}\ x\ 5\ r : \textbf{reduce}\ i$, and then use rule E-Let to conclude $\Gamma \vdash e : \textbf{reduce}\ i$.

In Section 4.3, we show that all well-formed programs terminate. By using a for-loop in conjunction with references, we may trying writing a program that diverges

$$\mathsf{for}\ x < !r\ \mathsf{do}\ \mathsf{let}\ \_\colon \textbf{int} = (r := !r + 1)\ \mathsf{in}\ \mathsf{skip}.$$

Store formation $\boxed{\Gamma \vdash \rho \ \textbf{store}}$

$$\varepsilon \vdash \varepsilon \ \textbf{store} \qquad \frac{\Gamma \vdash \rho \ \textbf{store} \qquad \Gamma \vdash v : D}{\Gamma, r : D \ \textbf{ref} \vdash \rho, r := v \ \textbf{store}}$$

Process formation $\boxed{\Gamma \vdash P : T}$

$$\frac{\Delta \vdash \rho \ \textbf{store} \qquad \Gamma, \Delta \vdash e : T \qquad \Gamma \vdash T \ \textbf{type}}{\Gamma, \Delta \vdash \langle \rho, e \rangle : T}$$

Program formation $\boxed{\Gamma \vdash S : T}$

$$\frac{\Gamma_0^n, \Delta_0 \vdash P_0 : T \quad \ldots \quad \Gamma_{n-1}^n, \Delta_{n-1} \vdash P_{n-1} : T \qquad \Delta_0, \ldots, \Delta_{n-1} \ \textbf{context} \qquad \Gamma^n \vdash T \ \textbf{type}}{\Gamma^n \vdash P_0 \mid \cdots \mid P_{n-1} : T}$$

Fig. 10. Store, process, and program formation.

Any attempt to write a derivation for this program will necessarily fail. Consider a context r: **int ref** binding r, the only free variable in the expression. We first apply rule E-All to obtain a premise featuring the following candidate context r: **int ref**, x: $\{y : \textbf{nat} \mid y < !r\}$. There is now one path in the candidate derivation leading to a judgement of the form

$$r : \textbf{int ref}, y : \textbf{nat} \vdash !r : \textbf{int}.$$

which is unsatisfiable. The path is composed of rules E-Let, IS-Assign, I-Var, C-Bind, D-Refin, and P-Rel (possibly with occurrences of rule E-Eq). The crucial observation is that a refinement type for variable x is added to the context and refinement types may not contain operations on references.

Despite bearing a similar syntax, expressions if p then e else f and ifc p then e else f are quite different. The former is evaluated within a process (Figure 3), the latter is a collective operation evaluated at the level of programs (Figure 4). The conventional conditional expression cannot be seen as particular case of the collective conditional (of type **ifc** p **then** T **else** T) for it would require a global synchronisation among all processes which is not what a local conditional suggests.

We now turn our attention to programs. Store, process, and program formation are in Figure 10. A separate, yet conventional, typing context Δ is used to type the store. The expression in a process is typed under a pair of contexts: Δ for the references and Γ for the variables. The latter context typically contains bindings for variables *size* and *myrank*. The variables in context Δ—reference identifiers—cannot show up in the type T of the expression, an assumption captured by premise $\Gamma \vdash T$ **type**.

Programs, ordered lists of *size* processes, are typed under a context that introduces n for the number of processes, that is, *size*: $\{x : \textbf{int} \mid x = n\}$, denoted by Γ^n. Each process adds to this context an entry for variable *myrank* with the corresponding process number. Notation Γ_m^n denotes the context Γ^n, *myrank*: $\{y : \textbf{int} \mid y = m\}$. All processes in a program share the same type T; this is the type of the program itself.

We complete this section by discussing the two last premises to the rule for programs. Premise $\Delta_0, \ldots, \Delta_{n-1}$ **context** forces store locality by not allowing the same reference name to appear in two distinct processes. Premise $\Gamma^n \vdash T$ **type** makes sure variable *myrank* does *not* appear in type T. Suppose that we allow variable *myrank* in the type of a program. The following program

$$\langle \varepsilon, \text{send } ((myrank + 1)\%size)\ 5 \rangle \mid \langle \varepsilon, \text{send } ((myrank + 1)\%size)\ 7 \rangle$$

would be typable at type **message** *myrank* $((myrank + 1)\%size)$, yet it would deadlock, for there is no receiving process for any of the two send operations.

4 PROGRESS AND STRONG NORMALISATION

This section presents the main results of ParTypes: progress, preservation, and strong normalisation. In the process, we introduce a notion of normal forms for types and an algorithm to convert types to normal forms.

4.1 Administrative Results for Programs and Types

We start by addressing weakening, strengthening and agreement for the relevant syntactic constructions introduced in Figures 5 to 10.

Lemma 4.1 (Weakening). *Let* $\Gamma \vdash D$ **dtype**.

(1) *If* $\Gamma \vdash i : E$, *then* $\Gamma, x\colon D \vdash i : E$.
(2) *If* $\Gamma \vdash_{\mathsf{s}} i : E$, *then* $\Gamma, x\colon D \vdash_{\mathsf{s}} i : E$.
(3) *If* $\Gamma \vdash p$ **prop**, *then* $\Gamma, x\colon D \vdash p$ **prop**.
(4) *If* $\Gamma \vdash_{\mathsf{s}} p$ **prop**, *then* $\Gamma, x\colon D \vdash_{\mathsf{s}} p$ **prop**.
(5) *If* $\Gamma \vdash E$ **dtype**, *then* $\Gamma, x\colon D \vdash E$ **dtype**.
(6) *If* Γ **context**, *then* $\Gamma, x\colon D$ **context**.
(7) *If* $\Gamma \vdash p$ **true**, *then* $\Gamma, x\colon D \vdash p$ **true**.
(8) *If* $\Gamma \vdash E <: F$, *then* $\Gamma, x\colon D \vdash E <: F$.
(9) *If* $\Gamma \vdash E \equiv F$, *then* $\Gamma, x\colon D \vdash E \equiv F$.
(10) *If* $\Gamma \vdash T$ **type**, *then* $\Gamma, x\colon D \vdash T$ **type**.
(11) *If* $\Gamma \vdash e : T$, *then* $\Gamma, x\colon D \vdash e : T$.

Proof. By simultaneous rule induction on the hypothesis in each case. □

Strengthening allows removing an entry $x\colon D$ from a context $\Gamma, x\colon D, \Delta$ under the conditions that x does not occur in the types in Δ and that D is closed (so that information in refinement types in D cannot be used by semantic entailment).

Lemma 4.2 (Strengthening). *Assume that* $x \notin \mathrm{fv}(\Delta)$ *and* $\vdash D$ **dtype**.

(1) *If* $\Gamma, x\colon D, \Delta \vdash i : E$ *and* $x \notin \mathrm{fv}(E)$, *then* $\Gamma, \Delta \vdash i : E$.
(2) *If* $\Gamma, x\colon D, \Delta \vdash_{\mathsf{s}} i : E$ *and* $x \notin \mathrm{fv}(E)$, *then* $\Gamma, \Delta \vdash_{\mathsf{s}} i : E$.
(3) *If* $\Gamma, x\colon D, \Delta \vdash p$ **prop** *and* $x \notin \mathrm{fv}(p)$, *then* $\Gamma, \Delta \vdash p$ **prop**.
(4) *If* $\Gamma, x\colon D, \Delta \vdash_{\mathsf{s}} p$ **prop** *and* $x \notin \mathrm{fv}(p)$, *then* $\Gamma, \Delta \vdash_{\mathsf{s}} p$ **prop**.
(5) *If* $\Gamma, x\colon D, \Delta \vdash E$ **dtype** *and* $x \notin \mathrm{fv}(E)$, *then* $\Gamma, \Delta \vdash E$ **dtype**.
(6) *If* $\Gamma, x\colon D, \Delta$ **context**, *then* Γ, Δ **context**.
(7) *If* $\Gamma, x\colon D, \Delta \vdash p$ **true** *and* $x \notin \mathrm{fv}(p)$ *and* $\mathrm{fv}(D) = \emptyset$, *then* $\Gamma, \Delta \vdash p$ **true**.
(8) *If* $\Gamma, x\colon D, \Delta \vdash E <: F$ *and* $x \notin \mathrm{fv}(E, F)$, *then* $\Gamma, \Delta \vdash E <: F$
(9) *If* $\Gamma, x\colon D, \Delta \vdash T \equiv U$ *and* $x \notin \mathrm{fv}(T, U)$, *then* $\Gamma \vdash T \equiv U$.
(10) *If* $\Gamma, x\colon D, \Delta \vdash T$ **type** *and* $x \notin \mathrm{fv}(T)$, *then* $\Gamma, \Delta \vdash T$ **type**.
(11) *If* $\Gamma, x\colon D, \Delta \vdash e : T$ *and* $x \notin \mathrm{fv}(e)$, *then* $\Gamma \vdash e : T$.

Proof. By simultaneous rule induction on the first hypothesis in each case. □

Lemma 4.3 (Agreement).

(1) *If* $\Gamma \vdash i : D$, *then* $\Gamma \vdash D$ **dtype**.
(2) *If* $\Gamma \vdash_{\mathsf{s}} i : D$, *then* $\Gamma \vdash D$ **dtype**.
(3) *If* $\Gamma \vdash p$ **prop**, *then* Γ **context**.
(4) *If* $\Gamma \vdash_{\mathsf{s}} p$ **prop**, *then* Γ **context**.
(5) *If* $\Gamma \vdash D$ **dtype**, *then* Γ **context**.

(6) *If* $\Gamma \vdash p$ **true**, *then* $\Gamma \vdash p$ **prop**.
(7) *If* $\Gamma \vdash D <: E$, *then* $\Gamma \vdash D$ **dtype** *and* $\Gamma \vdash E$ **dtype**.
(8) *If* $\Gamma \vdash D \equiv E$, *then* $\Gamma \vdash D$ **dtype** *and* $\Gamma \vdash E$ **dtype**.
(9) *If* $\Gamma \vdash T$ **type**, *then* Γ **context**.
(10) *If* $\Gamma \vdash e : T$, *then* $\Gamma \vdash T$ **type**.
(11) *If* $\Gamma \vdash \rho$ **store**, *then* Γ **context**.
(12) *If* $\Gamma \vdash P : T$, *then* $\Gamma \vdash T$ **type**.
(13) *If* $\Gamma \vdash S : T$, *then* $\Gamma \vdash T$ **type**.

Proof. By simultaneous rule induction on the hypothesis in each case. For expressions, using strengthening (Lemma 4.2), we detail one representative case: the collective conditional expression. We show that $\Gamma \vdash p$ **prop** and $\Gamma, \_ : \{p\} \vdash T$ **type** and $\Gamma, \_ : \{\neg p\} \vdash U$ **type** to conclude that $\Gamma \vdash$ **ifc** p **then** T **else** U **type** using rule T-Ifc in Figure 5. From premise $\Gamma, \_ : \{p\} \vdash e : T$, by induction, we have $\Gamma, \_ : \{p\} \vdash T$ **type**. A similarly reasoning allows concluding that $\Gamma, \_ : \{\neg p\} \vdash U$ **type**. By item 9 we have $\Gamma, \_ : \{p\}$ **context**, and from the rules C-Bind and D-Refin in Figure 5 we have $\Gamma, \_ :$ **int** $\vdash p$ **prop**, hence, by strengthening, we have $\Gamma \vdash p$ **prop**. □

The substitution lemma is a cornerstone in the proof of preservation (Theorem 4.17).

Lemma 4.4 (Substitution). *Let* $\Gamma \vdash i : D$.

(1) *If* $\Gamma, x : D, \Delta \vdash j : E$, *then* $\Gamma, \Delta\{i/x\} \vdash j\{i/x\} : E\{i/x\}$.
(2) *If* $\Gamma, x : D, \Delta \vdash_s j : E$, *then* $\Gamma, \Delta\{i/x\} \vdash_s j\{i/x\} : E\{i/x\}$.
(3) *If* $\Gamma, x : D, \Delta \vdash p$ **prop**, *then* $\Gamma, \Delta\{i/x\} \vdash p\{i/x\}$ **prop**.
(4) *If* $\Gamma, x : D, \Delta \vdash_s p$ **prop**, *then* $\Gamma, \Delta\{i/x\} \vdash_s p\{i/x\}$ **prop**.
(5) *If* $\Gamma, x : D, \Delta \vdash E$ **dtype**, *then* $\Gamma, \Delta\{i/x\} \vdash E\{i/x\}$ **dtype**.
(6) *If* $\Gamma, x : D, \Delta$ **context**, *then* $\Gamma, \Delta\{i/x\}$ **context**.
(7) *If* $\Gamma, x : D, \Delta \vdash p$ **true**, *then* $\Gamma, \Delta\{i/x\} \vdash p\{i/x\}$ **true**.
(8) *If* $\Gamma, x : D, \Delta \vdash E <: F$, *then* $\Gamma, \Delta\{i/x\} \vdash E\{i/x\} <: F\{i/x\}$.
(9) *If* $\Gamma, x : D, \Delta \vdash E \equiv F$, *then* $\Gamma, \Delta\{i/x\} \vdash E\{i/x\} \equiv F\{i/x\}$.
(10) *If* $\Gamma, x : D, \Delta \vdash T$ **type**, *then* $\Gamma, \Delta\{i/x\} \vdash T\{i/x\}$ **type**.
(11) *If* $\Gamma, x : D, \Delta \vdash e : T$, *then* $\Gamma, \Delta\{i/x\} \vdash e\{i/x\} : T\{i/x\}$.

The same under hypothesis $\Gamma \vdash_s i : D$.

Proof. By simultaneous rule induction on the hypothesis in each case. In item 1, when the derivation ends with the rule I-Var, we use weakening (Lemma 4.1). □

Lemma 4.5 (Store Update). *If* $\Gamma \vdash \rho$ **store** *and* $\Gamma \vdash_s r : D$ ***ref*** *and* $\Gamma \vdash_s v : D$, *then* $\Gamma \vdash \rho[r := v]$ **store**.

Proof. A straightforward rule induction on the first hypothesis. □

The proofs of normalisation for evaluation, progress and preservation both for evaluation and reduction (Lemmas 4.12 and 4.13 and Theorems 4.14 and 4.17) require extracting the general form of the type for a given expression.

Lemma 4.6 (Inversion for Expression Formation). *Let* $\Gamma \vdash e : T$.

(1) *If* $e =$ skip, *then* $\Gamma \vdash T \equiv$ ***skip***.
(2) *If* $e =$ send $i\ j$, *then* $\Gamma \vdash T \equiv$ ***message*** *myrank* i *and* $\Gamma \vdash$ *myrank* $:$ ***rank*** *and* $\Gamma \vdash i :$ ***rank*** *and* $\Gamma \vdash i \neq$ *myrank* **true** *and* $\Gamma \vdash_s j :$ ***int***.
(3) *If* $e =$ receive $i\ j$, *then* $\Gamma \vdash T \equiv$ ***message*** i *myrank* *and* $\Gamma \vdash i :$ ***rank*** *and* $\Gamma \vdash$ *myrank* $:$ ***rank*** *and* $\Gamma \vdash i \neq$ *myrank* **true** *and* $\Gamma \vdash_s j :$ ***int ref***.

(4) *If* $e = \mathsf{reduce}\ i\ j\ k$, *then* $\Gamma \vdash T \equiv \mathbf{reduce}\ i$ *and* $\Gamma \vdash i : \mathbf{rank}$ *and* $\Gamma \vdash_{\mathsf{s}} j : \mathbf{int}$ *and* $\Gamma \vdash_{\mathsf{s}} k : \mathbf{int\ ref}$.
(5) *If* $e = \mathsf{let}\ x\colon D = \mathsf{broadcast}\ i\ j\ \mathsf{in}\ e$, *then* $\Gamma \vdash T \equiv \mathbf{broadcast}\ i\ x\colon D.U$ *and* $\Gamma \vdash i : \mathbf{rank}$ *and* $\Gamma \vdash_{\mathsf{s}} j : D$ *and* $\Gamma \vdash D <: \mathbf{int}$ *and* $\Gamma, x\colon D \vdash e : U$.
(6) *If* $e = \mathsf{ifc}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f$, *then* $\Gamma \vdash T \equiv \mathbf{ifc}\ p\ \mathbf{then}\ U\ \mathbf{else}\ V$ *and* $\Gamma, \_\colon \{p\} \vdash e : U$ *and* $\Gamma, \_\colon \{\neg p\} \vdash f : V$.
(7) *If* $e = e; f$, *then* $\Gamma \vdash T \equiv U; V$ *and* $\Gamma \vdash e : U$ *and* $\Gamma \vdash f : V$.
(8) *If* $e = \mathsf{for}\ x < i\ \mathsf{do}\ e$, *then* $\Gamma \vdash T \equiv \forall\, x{<}i.U$ *and* $\Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i\} \vdash e : U$ *and* $y \notin \mathrm{fv}(i)$.
(9) *If* $e = \mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f$, *then* $\Gamma \vdash T \equiv U$ *and* $\Gamma \vdash_{\mathsf{s}} p\ \mathbf{prop}$ *and* $\Gamma \vdash e : U$ *and* $\Gamma \vdash f : U$.
(10) *If* $e = \mathsf{lets}\ x\colon D = i\ \mathsf{in}\ e$, *then* $\Gamma \vdash T \equiv U$ *and* $\Gamma \vdash_{\mathsf{s}} i : D$ *and* $\Gamma, x\colon D \vdash e : U$ *and* $x \notin \mathrm{fv}(T)$.
(11) *If* $e = \mathsf{let}\ x\colon D = i\ \mathsf{in}\ e$, *then* $\Gamma \vdash T \equiv U$ *and* $\Gamma \vdash i : D$ *and* $\Gamma, x\colon \{y\colon D \mid y = i\} \vdash e : T$ *and* $x \notin \mathrm{fv}(T)$ *and* $y \notin \mathrm{fv}(i)$.

Proof. Because ≡ is an equivalence relation (Lemma 3.1) we may assume that each derivation ends with exactly one occurrence of rule E-Eq. Each case is then a simple analysis of the corresponding rule in Figure 9. □

The last result in this section tells us the possible shapes of expressions that belong to a certain type.

Lemma 4.7 (Canonical Forms). *Let* $\Gamma \vdash e : T$.

(1) *If* $T = \mathbf{message}\ i\ j; U$, *then*
  (a) $e = e'; \mathsf{send}\ k\ l; e''$ *and* $\Gamma \vdash \mathit{myrank} = i$ **true**, *or*
  (b) $e = e'; \mathsf{receive}\ k\ l; e''$ *and* $\Gamma \vdash \mathit{myrank} = j$ **true**, *or else*
  (c) $e = e'; e''$ *and* $\Gamma \vdash i \neq \mathit{myrank}$ **true** *and* $\Gamma \vdash j \neq \mathit{myrank}$ **true**.
(2) *If* $T = \mathbf{reduce}\ i; U$, *then* $e = e'; \mathsf{reduce}\ j\ k\ l; e''$ *and* $\Gamma \vdash i = j$ **true**.
(3) *If* $T = \mathbf{broadcast}\ i\ x\colon D.V; U$, *then* $e = e'; \mathsf{let}\ x\colon D = \mathsf{broadcast}\ j\ k\ \mathsf{in}\ f; e''$ *and* $\Gamma \vdash i = j$ **true**.
(4) *If* $T = \mathbf{ifc}\ p\ \mathbf{then}\ V\ \mathbf{else}\ W; U$, *then* $e = e'; \mathsf{ifc}\ q\ \mathsf{then}\ f\ \mathsf{else}\ f'; e''$ *and* $\Gamma \vdash p \Leftrightarrow q$ **true**.
(5) *If* $T = \forall\, x{<}i.V; U$, *then* $e = e'; \mathsf{for}\ x < j\ \mathsf{do}\ f; e''$ *and* $\Gamma \vdash i = j$ **true**.

Furthermore, in all cases, $\Gamma \vdash e' : \mathbf{skip}$ *and* $\Gamma \vdash e'' : U$ *and one or both of* e' *and* e'' *may be absent (in the latter case* $U = \mathbf{skip}$*).*

Proof. The proof is by inspection of the typing derivation for the expressions, together with the inspection of the typing rules for type equivalence. The structure of the proof is similar for all cases, so we only report that for $\mathbf{message}\ i\ j; T$. Assume that the derivation ends with exactly one occurrence of rule E-Eq (cf. proof of Lemma 4.6). We have $\Gamma \vdash e : T'$ and $\Gamma \vdash T' \equiv \mathbf{message}\ i\ j; T$. Inspecting the rules for type equivalence we conclude that the general form of T' is $T_1; \mathbf{message}\ k\ l; T_2$, with $\Gamma \vdash T_1 \equiv \mathbf{skip}$ and $\Gamma \vdash T_2 \equiv T$. Inspecting the rules for expression formation we conclude that e is of the form $e'; f; e''$. Rule E-Eq gives $\Gamma \vdash e' : \mathbf{skip}$ and $\Gamma \vdash e'' : T$. Now f can be send or receive of skip. Items 1a and 1b follow by rules E-Send and E-recv, and item 1c by rules E-Eq, E-Skip and Eq-Proj. □

4.2 Type Conversion and List Normal Forms

The notion of type equivalence introduced in Section 3.3 induces equivalent types that can be quite different, syntactically. Fortunately, types can be converted into a normal form that allows for quick verification of type equivalence and for a quick assessment of the size of a type. The former notion is used in algorithmic type equivalence (Section 5.1); the latter in strong normalisation (Section 4.4).

The normal forms we are interested in resemble lists, hence the name *list normal form*. The inductive definition is in Figure 11 and comprises one rule for each of the type constructors

List normal form $\boxed{\Gamma \vdash T\ \mathbf{nf}}$

$$\frac{(\Gamma\ \mathbf{context})}{\Gamma \vdash \mathbf{skip}\ \mathbf{nf}} \qquad \frac{\Gamma \vdash T\ \mathbf{nf} \quad (\Gamma \vdash i : \mathbf{rank})}{\Gamma \vdash \mathbf{reduce}\ i; T\ \mathbf{nf}} \qquad \frac{\Gamma, x\colon D \vdash T\ \mathbf{nf} \quad \Gamma \vdash U\ \mathbf{nf} \quad (\Gamma \vdash i : \mathbf{rank})}{\Gamma \vdash \mathbf{broadcast}\ i\ x\colon D.T; U\ \mathbf{nf}}$$

$$\frac{\Gamma \nvdash i, j \neq \mathit{myrank} \quad \Gamma \vdash T\ \mathbf{nf} \quad (\Gamma \vdash \mathbf{message}\ i\ j\ \mathbf{type})}{\Gamma \vdash \mathbf{message}\ i\ j; T\ \mathbf{nf}}$$

$$\frac{\Gamma \nvdash p \quad \Gamma \nvdash \neg p \quad \Gamma \vdash T, U, V\ \mathbf{nf} \quad (\Gamma \vdash p\ \mathbf{prop})}{\Gamma \vdash \mathbf{ifc}\ p\ \mathbf{then}\ T\ \mathbf{else}\ U; V\ \mathbf{nf}}$$

$$\frac{\Gamma \nvdash i = 0 \quad \Gamma \nvdash i > 0 \quad \Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i\} \vdash T\ \mathbf{nf} \quad \Gamma \vdash U\ \mathbf{nf}}{\Gamma \vdash \forall\ x{<}i.T; U\ \mathbf{nf}}$$

Fig. 11. List normal form.

introduced so far. We can easily see that objects in list normal form are types (Lemma 4.8); the premises in parentheses work towards this end. Intuitively, a type is in list normal form, if it is of the form $T_1; T_2; \ldots; T_n; \mathbf{skip}$, for $n \geq 0$, and each type T_i is neither **skip** nor of the form $\_;\_$. In addition, types that occur in T_i are themselves in list normal form. There are three more conditions for a type to be a list normal form: message types must be genuine (not equivalent to **skip**), conditional types cannot be simplified, and primitive recursion cannot be unfolded or eliminated. For messages $\mathbf{message}\ i\ j$ we make sure that neither the sender i nor the recipient j can be proved equal to *myrank*; for conditional types $\mathbf{ifc}\ p\ \mathbf{then}\ T\ \mathbf{else}\ U$ we require that neither p nor $\neg p$ can be proved; and for primitive recursion $\forall\ x{<}i.T$ we require that the upper bound i cannot be proved neither equal nor greater than 0.

Conversion to normal form is performed by the algorithm in Figure 12. The first rule builds an empty list (**skip**) from the terminated type (**skip**). As before, premises in parentheses account for agreement (Lemma 4.8). The first rule for messages builds a singleton list. The second is for non-genuine messages, that is, messages that are equivalent to **skip** because both the sender and the recipient of the message are different from *myrank*. In this case the message is converted to **skip**. The rules for **reduce** and **broadcast** build singleton lists from types. The rule for sequential composition calls recursively the conversion rule and concatenates the results. Type concatenation, T ++ U, is a standard list concatenation function. There are three rules for converting primitive recursion. When the bound cannot be proved neither equal nor greater than 0, the function builds a singleton list. Otherwise, when the upper bound can be proved to be 0, the type is converted into **skip**, and when the upper bound can be proved to be larger than 0 the type is expanded into a sequential composition, as in the corresponding rules in type equivalence (Figure 7). Finally, the rules for the conditional type follow an approach similar to primitive recursion, featuring three cases: when both p and $\neg p$ cannot be proved, when p is true, and when p is false.

For example, let T be the type $\mathbf{message}\ 0\ 1$ and recall that Γ_1^3 is the context stating that *size* is 3 and *myrank* is 1. Then,

$$\Gamma_1^3 \vdash T \Downarrow T; \mathbf{skip}$$

because $\Gamma_1^3 \vDash 1 = \mathit{myrank}$, but

$$\Gamma_2^3 \vdash T \Downarrow \mathbf{skip}$$

because $\Gamma_2^3 \nvDash 0 = \mathit{myrank}$ and $\Gamma_2^3 \nvDash 1 = \mathit{myrank}$. For an example with primitive recursion, let U be the type $\forall\ x{<}y.\mathbf{reduce}\ x$. Then, we have

$$\Gamma_1^2, y\colon \{z\colon \mathbf{int} \mid z = 2\} \vdash U \Downarrow \mathbf{reduce}\ 1; \mathbf{reduce}\ 0; \mathbf{skip}$$

Type conversion $\boxed{\Gamma \vdash T \Downarrow T}$

$$\frac{(\Gamma \textbf{ context})}{\Gamma \vdash \textbf{skip} \Downarrow \textbf{skip}} \qquad \frac{\Gamma \vDash i = myrank \quad (\Gamma \vdash \textbf{message } i\ j \textbf{ type})}{\Gamma \vdash \textbf{message } i\ j \Downarrow \textbf{message } i\ j; \textbf{skip}}$$

$$\frac{\Gamma \vDash j = myrank \quad (\Gamma \vdash \textbf{message } i\ j \textbf{ type})}{\Gamma \vdash \textbf{message } i\ j \Downarrow \textbf{message } i\ j; \textbf{skip}}$$

$$\frac{\Gamma \nvDash i = myrank \quad \Gamma \nvDash j = myrank \quad (\Gamma \vdash \textbf{message } i\ j \textbf{ type})}{\Gamma \vdash \textbf{message } i\ j \Downarrow \textbf{skip}}$$

$$\frac{(\Gamma \vdash i : \textbf{rank})}{\Gamma \vdash \textbf{reduce } i \Downarrow \textbf{reduce } i; \textbf{skip}} \qquad \frac{\Gamma, x\colon D \vdash T \Downarrow U \quad (\Gamma \vdash i : \textbf{rank})}{\Gamma \vdash \textbf{broadcast } i\ x\colon D.T \Downarrow \textbf{broadcast } i\ x\colon D.U; \textbf{skip}}$$

$$\frac{\Gamma \vdash T \Downarrow T' \quad \Gamma \vdash U \Downarrow U' \quad \Gamma \vdash T' +\!\!+ U' = V}{\Gamma \vdash T; U \Downarrow V}$$

$$\frac{\Gamma \nvDash i = 0 \quad \Gamma \nvDash i > 0 \quad \Gamma, x\colon \{y\colon \textbf{nat} \mid y < i\} \vdash T \Downarrow U}{\Gamma \vdash \forall\, x{<}i.T \Downarrow \forall\, x{<}i.U; \textbf{skip}}$$

$$\frac{\Gamma \vDash i = 0 \quad (\Gamma, x\colon \{y\colon \textbf{nat} \mid y < i\} \vdash T \textbf{ type})}{\Gamma \vdash \forall\, x{<}i.T \Downarrow \textbf{skip}}$$

$$\frac{\begin{array}{c}\Gamma \vDash i > 0 \quad \Gamma \vdash T\{i-1/x\} \Downarrow U \quad \Gamma \vdash \forall\, x{<}i-1.T \Downarrow V \\ \Gamma \vdash U +\!\!+ V = W \quad (\Gamma, x\colon \{y\colon \textbf{nat} \mid y < i\} \vdash T \textbf{ type})\end{array}}{\Gamma \vdash \forall\, x{<}i.T \Downarrow W}$$

$$\frac{\Gamma \nvDash p \quad \Gamma \nvDash \neg p \quad \Gamma \vdash T \Downarrow T' \quad \Gamma \vdash U \Downarrow U' \quad (\Gamma \vdash p \textbf{ prop})}{\Gamma \vdash \textbf{ifc } p \textbf{ then } T \textbf{ else } U \Downarrow \textbf{ifc } p \textbf{ then } T' \textbf{ else } U'; \textbf{skip}}$$

$$\frac{\Gamma \vdash p \textbf{ true} \quad \Gamma \vdash T \Downarrow T' \quad (\Gamma \vdash U \textbf{ type})}{\Gamma \vdash \textbf{ifc } p \textbf{ then } T \textbf{ else } U \Downarrow T'} \qquad \frac{\Gamma \vdash \neg p \textbf{ true} \quad \Gamma \vdash U \Downarrow U' \quad (\Gamma \vdash T \textbf{ type})}{\Gamma \vdash \textbf{ifc } p \textbf{ then } T \textbf{ else } U \Downarrow U'}$$

List concatenation $\boxed{\Gamma \vdash T +\!\!+ T = T}$

$$\frac{(\Gamma \vdash T \textbf{ type})}{\Gamma \vdash \textbf{skip} +\!\!+ T = T} \qquad \frac{\Gamma \vdash U +\!\!+ V = W \quad (\Gamma \vdash T \textbf{ type})}{\Gamma \vdash T; U +\!\!+ V = T; W}$$

Structural type equivalence $\boxed{\Gamma \vdash T \equiv_s T}$

Defined by the first seven rules of type equivalence in fig. 7, one for each type constructor

Fig. 12. Type conversion.

because $\Gamma \vDash y > 0$ and $\Gamma \vDash y - 1 > 0$ and $\Gamma \vDash y - 1 - 1 = 0$, but

$$\Gamma_1^2, y\colon \textbf{int} \vdash U \Downarrow U; \textbf{skip}$$

because $\Gamma \nvDash y > 0$ and $\Gamma \nvDash y = 0$.

The following lemma states that the judgements introduced in Figures 11 and 12 respect agreement.

Lemma 4.8 (Agreement for Type Conversion).

(1) *If* $\Gamma \vdash T$ **nf**, *then* $\Gamma \vdash T$ **type**.
(2) *If* $\Gamma \vdash T \Downarrow U$, *then* $\Gamma \vdash T, U$ **type**.
(3) *If* $\Gamma \vdash T +\!\!+ U = V$, *then* $\Gamma \vdash T, U, V$ **type**.
(4) *If* $\Gamma \vdash T \equiv_s U$, *then* $\Gamma \vdash T, U$ **type**.

Proof. By rule induction on each hypothesis, using the rule premises in parentheses. □

We now look into properties of concatenation: type concatenation is defined for types in normal form; the result of concatenation is type equivalent to the sequential composition of the input; and given two types in list normal form, the result of the concatenation is in list normal form.

Lemma 4.9 (Properties of Concatenation).

(1) *If* $\Gamma \vdash T$ **nf** *and* $\Gamma \vdash U$ **nf**, *then* $\Gamma \vdash T +\!\!+ U = V$, *for some type* V.
(2) *If* $\Gamma \vdash T +\!\!+ U = V$, *then* $\Gamma \vdash T; U \equiv V$.
(3) *If* $\Gamma \vdash T +\!\!+ U = V$ *and* $\Gamma \vdash T$ **nf** *and* $\Gamma \vdash U$ **nf**, *then* $\Gamma \vdash V$ **nf**.

Proof. By rule induction on the first hypothesis, in all cases. We detail some cases of items 2 and 3.

Item 2. Case the derivation ends with $\Gamma \vdash \mathbf{skip} +\!\!+ U = U$. The premise reads $\Gamma \vdash U$ **type**. Conclude with rule Eq-NeutralL. Case the derivation ends with $\Gamma \vdash T'; T'' +\!\!+ U = T'; W$. The premises read $\Gamma \vdash T'' +\!\!+ U = W$ and $\Gamma \vdash T'$ **type**. By induction $\Gamma \vdash T''; U \equiv W$. Because $\Gamma \vdash T'$ **type**, we have $\Gamma \vdash T'; (T''; U) \equiv T'; W$ and $\Gamma \vdash (T'; T''); U \equiv T'; W$. Hence $\Gamma \vdash T; U \equiv T'; W$.

Item 3. Case the derivation ends with $\Gamma \vdash \mathbf{skip} +\!\!+ U = U$. Then, $\Gamma \vdash U$ **nf** by hypothesis. Case the derivation ends with $\Gamma \vdash T'; T'' +\!\!+ U = T'; W$. The premise reads $\Gamma \vdash T'' +\!\!+ U = W$. Proceed by rule induction on $\Gamma \vdash T'; T''$ **nf**. There are five cases to consider—**reduce**, **broadcast**, **message**, (**ifc**), and (∀)—they are all similar. Take **broadcast** as an example. The premises of the rule read $\Gamma, x : D \vdash T$ **nf** and $\Gamma \vdash T''$ **nf** and $\Gamma \vdash i : \mathbf{rank}$. The induction hypothesis yields $\Gamma \vdash W$ **nf**. Conclude with this information and $\Gamma, x : D \vdash T$ **nf** and $\Gamma \vdash i : \mathbf{rank}$ and the rule for **reduce**. □

We now show that type conversion is strongly normalising. We also show that type conversion yields a type in normal form that is equivalent to the original type. We finally show that type conversion is defined on all well-formed types.

Lemma 4.10 (Properties of Type Conversion).

(1) *Type conversion is strongly normalizing.*
(2) *Type conversion is confluent.*
(3) *If* $\Gamma \vdash T \Downarrow U$, *then* $\Gamma \vdash U$ **nf** *and* $\Gamma \vdash T \equiv U$.
(4) *If* $\Gamma \vdash T$ **type**, *then* $\Gamma \vdash T \Downarrow U$, *for some type* U.

Proof. *Item 1.* The size of a type is defined as follows: size(**skip**) = size(**message** $i\ j$) = size(**reduce** i) = 1 and size(**broadcast** $i\ x : D.T$) = 1 + size(T) and size(**ifc** p **then** T **else** U) = size($T; U$) = 1 + size(T) + size(U) and size($\forall\ x{<}i.T$) = i + size(T). By agreement (Lemma 4.8) we know that $\Gamma \vdash T$ **type**. We can easily show that size(T) is a natural number. It is a straightforward exercise to check each type conversion rule strictly decreases the size of the rule premises.

Item 2. For each type other than **message**, collective conditional and recursor exactly one rule applies. In the three remaining cases, the various $\vDash$ premises are disjoint (for example, in the case of messages we have $\Gamma \vDash i, j \neq myrank$ and $\Gamma \nvDash i, j \neq myrank$).

Item 3. By rule induction on the hypothesis. We detail a few representative cases. For space constraints, we decided not to label the rules in Figure 12. Below we refer to the three rules that mention **message** as message/1, message/2 and message/3, according to the order by which they appear in the figure. Same for the remaining type constructors.

When the last rule is message/1, the premises to the rule include $\Gamma \nvDash i, j \neq myrank$ and $\Gamma \vdash \mathbf{message}\ i\ j$ **type**. By agreement, we have Γ **context**, hence $\Gamma \vdash \mathbf{skip}$ **nf**. This last fact and the two premises allow us to conclude that $\Gamma \vdash \mathbf{message}\ i\ j; \mathbf{skip}$ **nf**. That $\Gamma \vdash \mathbf{message}\ i\ j \equiv \mathbf{message}\ i\ j; \mathbf{skip}$ follows from rules Eq-Sym and Eq-NeutralR.

When the last rule is message/2, the premises to the rule read $\Gamma \vdash i, j \neq \mathit{myrank}$ **true** and $\Gamma \vdash$ **message** $i\ j$ **type**. By agreement, we have Γ **context**, hence $\Gamma \vdash$ **skip** **nf**. That $\Gamma \vdash$ **message** $i\ j \equiv$ **skip** follows from rule Eq-Proj.

When the last rule is recursor/1, the premises to the rule read $\Gamma \not\vdash i = 0$ and $\Gamma \not\vdash i > 0$ and $\Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i\} \vdash T \Downarrow U$. By induction, we have $\Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i\} \vdash U$ **nf** and also $\Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i\} \vdash T \equiv U$. To show that $\Gamma \vdash \forall\, x{<}i.U; \mathbf{skip}$ **nf**, we use premises $\Gamma \not\vdash i = 0$ and $\Gamma \not\vdash i > 0$, the first induction hypothesis and that $\Gamma \vdash$ **skip** **nf**. To show that $\Gamma \vdash \forall\, x{<}i.T \equiv \forall\, x{<}i.U; \mathbf{skip}$ we use transitivity and symmetry (Lemma 3.1) from the following two facts: (a) $\Gamma \vdash \forall\, x{<}i.T \equiv \forall\, x{<}i.U$, obtained from the appropriate rule for type equivalence, the second induction hypothesis, and $\Gamma \vdash i = i$ **true**; and (b) $\Gamma \vdash \forall\, x{<}i.U; \mathbf{skip} \equiv \forall\, x{<}i.U$, obtained from the second induction hypothesis, agreement (Lemma 4.3), the type formation rule for the recursor and rule Eq-NeutralR.

When the last rule is recursor/3, the premises to the rule read $\Gamma \vdash T\{i-1/x\} \Downarrow U$ and $\Gamma \vdash \forall\, x{<}i-1.T \Downarrow V$ and $\Gamma \vdash U +\!\!+ V = W$. To show that $\Gamma \vdash \forall\, x{<}i.T \equiv W$, we notice that $\Gamma \vdash (\forall\, x{<}i.T) \equiv (T\{i-1/x\}; \forall\, x{<}i-1.T)$. To show that $\Gamma \vdash W$ **nf**, we use induction on $\Gamma \vdash T\{i-1/x\} \Downarrow U$ and on $\Gamma \vdash \forall\, x{<}i-1.T \Downarrow V$, and properties of concatenation (Lemma 4.9, item 3) on $\Gamma \vdash U +\!\!+ V = W$. To show that $\Gamma \vdash (T\{i-1/x\}; \forall\, x{<}i-1.T) \equiv W$ holds, we use the induction hypothesis on $\Gamma \vdash T\{i-1/x\} \Downarrow U$ and again on $\Gamma \vdash \forall\, x{<}i-1.T \Downarrow V$, as well properties of concatenation (Lemma 4.9, item 2) on $\Gamma \vdash U +\!\!+ V = W$.

Item 4. By rule induction on the hypothesis. The cases of **skip** and **reduce** are direct; **message** requires the analysis of two cases ($\Gamma \vDash i, j \neq \mathit{myrank}$ and $\Gamma \not\vDash i, j \neq \mathit{myrank}$); **broadcast** and sequential composition follow by induction; collective choice requires the analysis of three cases, namely (a) $\Gamma \not\vDash p$ and $\Gamma \not\vDash \neg p$, (b) $\Gamma \vDash p$, and (c) $\Gamma \vDash \neg p$; they all follow by a simple induction. The interesting case is the recursor where we must distinguish again three subcases: (a) $\Gamma \not\vDash i = 0$ and $\Gamma \not\vDash i > 0$, (b) $\Gamma \vDash i = 0$, and (c) $\Gamma \vDash i > 0$. We detail the last case. One of premises to the rule says that $\Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i\} \vdash T$ **type**, from which we can conclude that $\Gamma \vdash i - 1 : \{y\colon \mathbf{nat} \mid y < i\}$. Applying substitution (Lemma 4.4), we get $\Gamma \vdash T\{i-1/x\}$ **type**, and the induction hypothesis gives $\Gamma \vdash T\{i-1/x\} \Downarrow U'$, for some type U'. Using the same premise, we can show that $\Gamma, x\colon \{y\colon \mathbf{nat} \mid y < i-1\} \vdash T$ **type**. Using this fact and premise $y \notin \mathrm{fv}(i)$, we know that $\Gamma \vdash \forall\, x{<}i-1.T$ **type**, and by induction $\Gamma \vdash \forall\, x{<}i-1.T \Downarrow V'$, for some type V'. Now, from item 3 of this lemma we know that $\Gamma \vdash V'$ **nf** and $\Gamma \vdash U'$ **nf**. By properties of concatenation (Lemma 4.9, item 1), we know that there is a W such that $\Gamma \vdash U' +\!\!+ V' = W$. Conclude with the $\Gamma \vdash i > 0$ **true** rule for the recursor. □

The notion of structural equivalence (Figure 12) equates types that are syntactically equal except possibly on their index terms and propositions, which must be equal or have the same truth-value. For example, $\Gamma_2^3 \vdash$ **reduce** $2 \equiv_s$ **reduce** $1 + 1$. From the definition of structural equivalence, one easily concludes that structurally equivalent types are equivalent. Further, if two types are equivalent and one is in normal form, then the two types are structurally equivalent.

Lemma 4.11 (Properties of Structural Equivalence).

(1) *Structural equivalence is an equivalence relation.*
(2) *If* $\Gamma \vdash T \equiv_s U$, *then* $\Gamma \vdash T \equiv U$.
(3) *If* $\Gamma \vdash T \equiv U$ *and* $\Gamma \vdash T$ **nf** *and* $\Gamma \vdash U$ **nf**, *then* $\Gamma \vdash T \equiv_s U$.

Proof. *Item 1.* In $\equiv_s$, messages are only equal to messages, and likewise for the remaining six constructors. The result then follows from the fact that $\Gamma \vdash D \equiv E$ and $\Gamma \vdash i = j$ **true** and $\Gamma \vdash p \Leftrightarrow q$ **true** are all equivalence relations. For **reduce** and **broadcast**, we must further establish

that $\Gamma \vdash j : \mathbf{rank}$. We can easily show that this follows from $\Gamma \vdash i : \mathbf{rank}$ and $\Gamma \vdash i = j$ **true** (and similarly for **message** and indices k and l.)

Item 2. $\equiv_s$ is built solely from rules taken from $\equiv$.

Item 3. By rule induction on the first hypothesis. In the first seven cases, the relations coincide. Except for symmetry and transitivity, no other rule applies for either T or U are not (or cannot be) in list normal form. For symmetry, we have $\Gamma \vdash U \equiv T$, and by induction we get $\Gamma \vdash T \equiv_s U$. The result follows from the fact that $\equiv_s$ is symmetric (item 1). Transitivity is similar, following from the fact that $\equiv_s$ is transitive. □

4.3 Progress

Local reduction on processes is captured by evaluation. The next lemma states that evaluation preserves the good formation of stores and processes, as well as the datatypes of the index terms under evaluation. Notice that the lemma requires contexts with entries only for *size* and *myrank* (that is, Γ_m^n) and for references (Δ), for these are the only variables that may evaluate to a value.

Lemma 4.12 (Preservation for Evaluation). *Let* $\Delta \vdash \rho$ **store**.

(1) *If* $\langle \rho, i\rangle \Downarrow_m^n \langle \rho, v\rangle$ *and* $\Gamma_m^n, \Delta \vdash i : D$, *then* $\Delta \vdash i = v$ **true**.
(2) *If* $\langle \rho, i\rangle \Downarrow_m^n \langle \sigma, v\rangle$ *and* $\Gamma_m^n, \Delta \vdash_{\mathbf{s}} i : D$, *then* $\Sigma \vdash \sigma$ **store** *and* $\Sigma \vdash_{\mathbf{s}} v : D$.
(3) *If* $\langle \rho, p\rangle \Downarrow_m^n \langle \rho, b\rangle$ *and* $\Gamma_m^n, \Delta \vdash p$ **prop**, *then* $\Delta \vdash p \Leftrightarrow b$ **true**.
(4) *If* $\langle \rho, p\rangle \Downarrow_m^n \langle \sigma, q\rangle$ *and* $\Gamma_m^n, \Delta \vdash_{\mathbf{s}} p$ **prop**, *then* $\Sigma \vdash \sigma$ **store** *and* $\Gamma_m^n, \Sigma \vdash_{\mathbf{s}} q$ **prop**.
(5) *If* $P \Downarrow_m^n Q$ *and* $\Gamma_m^n, \Delta \vdash P : T$, *then* $\Gamma_m^n, \Sigma \vdash Q : T$.

In items 2, 4 and 5, $\mathrm{dom}(\rho) \subseteq \mathrm{dom}(\sigma)$ *and* $\Delta \subseteq \Sigma$ *and the references in* Σ *but not in* Δ *are fresh.*

Proof. By rule induction of the evaluation judgements using weakening, agreement, inversion for expression formation, substitution (Lemmas 4.1, 4.3, 4.4 and 4.6).

Item 1. A straightforward induction.

Item 2. We detail the case for rule IS-Deref. In this case, i is $!j$ and the premises to the rule read $\langle \rho, j\rangle \Downarrow_m^n \langle \sigma, r\rangle$ and $r := v \in \sigma$. The premises to the only rule for $\Gamma_m^n, \Delta \vdash_{\mathbf{s}} !j : D$ read $\Gamma_m^n, \Delta \vdash_{\mathbf{s}} j : D$ **ref**. By induction, $\Sigma \vdash \sigma$ **store** and $\Sigma \vdash_{\mathbf{s}} r : D$ **ref**. Finally, the rules for store formation and weakening give $\Sigma \vdash_{\mathbf{s}} v : E$.

Items 3 and 4. A straightforward induction.

Item 5. Let P be the process $\langle \rho, e'\rangle$. Premises to the only rule for processes read $\Delta \vdash \rho$ **store** and $\Gamma_m^n, \Delta \vdash e' : T$. We proceed by rule induction on the last judgement and analyse a couple of cases.

Case PE-Let. Inversion for e' gives $\Gamma_m^n, \Delta \vdash T \equiv U$ and $\Gamma_m^n, \Delta \vdash_{\mathbf{s}} i : D$ and $\Gamma_m^n, \Delta, x : D \vdash e : U$ and $x \notin \mathrm{fv}(T)$. Item 2 gives $\Delta \subseteq \Sigma \vdash \sigma$ **store** and $\Sigma \vdash_{\mathbf{s}} v : D$. Weakening followed by substitution gives $\Gamma_m^n, \Sigma \vdash e\{v/x\} : U\{v/x\}$ and rule E-Eq gives $\Gamma_m^n, \Sigma \vdash e\{v/x\} : T\{v/x\}$. But $T\{v/x\} = T$ because $x \notin \mathrm{fv}(T)$. Finally, induction gives $\Gamma_m^n, \Theta \vdash \langle \sigma, f\rangle : T$ with $\Sigma \subseteq \Theta$. Hence, $\Delta \subseteq \Theta$ and we are done.

Case PE-ForStep. Inversion for e' gives $\Gamma_m^n, \Delta \vdash T \equiv \forall\, x{<}i.U$ and $\Gamma_m^n, \Delta, x : \{y : \mathbf{nat} \mid y < i\} \vdash e : U$. Agreement gives $\Gamma_m^n, \Delta \vdash i : \mathbf{nat}$ and item 1 yields $\Gamma_m^n, \Delta \vdash i = k$ **true**. Rule PE-ForStep applies only when $k > 0$, hence $\Gamma_m^n, \Delta \vdash k - 1 : \{y : \mathbf{nat} \mid y < i\}$. Then we may apply substitution to obtain $\Gamma_m^n, \Delta \vdash e\{k-1/x\} : U\{k-1/x\}$. Because $i > 0$, we also have $\Gamma_m^n, \Delta, x : \{y : \mathbf{nat} \mid y < i - 1\} \vdash e : U$. Rule E-All gives $\Gamma_m^n, \Delta \vdash \text{for } x < i - 1 \text{ do } e : \forall\, x{<}i-1.U$. Rule E-Semi gives $\Gamma_m^n, \Delta \vdash (e\{k-1/x\}; \text{for } x < i - 1 \text{ do } e) : (U\{k-1/x\}; \forall\, x{<}i-1.U)$ and rule Eq-AllUnfold followed by Eq-All yields $\Gamma_m^n, \Delta \vdash (e\{k-1/x\}; \text{for } x < i - 1 \text{ do } e) : (\forall\, x{<}i.T)$. We also know that $k > 0$ because $\Gamma_m^n, \Delta \vdash i = k$ **true**. We may now apply the induction hypothesis to complete the proof. □

The next lemma states that evaluation for processes terminates and that datatypes and types are preserved during the operation.

Lemma 4.13 (Normalisation for Evaluation). *Let* $\Delta \vdash \rho$ **store** *with* $\mathrm{dom}(\Delta) = \mathrm{dom}(\rho)$.

(1) *If* $\Gamma_m^n, \Delta \vdash i : D$, *then* $\langle \rho, i\rangle \Downarrow_m^n \langle \rho, k\rangle$ *and* $\Gamma_m^n, \Delta \vdash i = k$ **true**.
(2) *If* $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} i : D$, *then* $\langle \rho, i\rangle \Downarrow_m^n \langle \sigma, v\rangle$ *and* $\Sigma \vdash \sigma$ **store**. *Furthermore, if* $D = E$ **ref**, *then* v *is a variable and* $v \in \mathrm{dom}(\rho)$.
(3) *If* $\Gamma_m^n, \Delta \vdash p$ **prop**, *then* $\langle \rho, p\rangle \Downarrow_m^n \langle \rho, b\rangle$ *and* $\Gamma_m^n, \Delta \vdash p \Leftrightarrow b$ **true**.
(4) *If* $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} p$ **prop**, *then* $\langle \rho, p\rangle \Downarrow_m^n \langle \sigma, b\rangle$ *and* $\Sigma \vdash \sigma$ **store**.
(5) *If* $\Gamma_m^n, \Delta \vdash P : T$, *then* $P \Downarrow_m^n Q$ *and* $\Gamma_m^n, \Sigma \vdash Q : T$ *and the expression in process* Q *is either* skip *or of the form* $c; e$.

In items 2, 4 and 5, $\mathrm{dom}(\rho) \subseteq \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma)$ *and* $\Delta \subseteq \Sigma$ *and the references in* σ *but not in* ρ *are fresh.*

Proof. Uses weakening, substitution, store update, inversion of expression formation and preservation for evaluation (Lemmas 4.1, 4.4 to 4.6 and 4.12). For items 1 to 4, the proof is by mutual rule induction on the first hypothesis in each item. We sketch a few representative cases.

Item 1. Case I-Var. Then i is a variable. It can be *size* or *myrank* (the entries in Γ_m^n) in which case rules IE-Size or IE-Myrank apply (and the value produced are integer values). Rule IE-Ref does not apply because $\Delta \vdash \rho$ **store** implies that i is of a **ref** type, but $\Gamma_m^n, \Delta \vdash i : D$ implies that D is not a **ref** type. Case I-Int. Directly by axiom IE-Int. Case I-Op. By induction and rule IE-Op. Case I-Refin and I-Sub. Directly by induction.

Item 2. Case IS-Op. The premises to the rule are $D =$ **int** and $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} i :$ **int** and $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} j :$ **int**. Induction gives $\langle \rho, i\rangle \Downarrow_m^n \langle \sigma, u\rangle$ and $\Sigma \vdash \sigma$ **store** and $\Gamma_m^n, \Sigma \vdash_{\mathsf{s}} v :$ **int** and $\Delta \subseteq \Sigma$. By weakening, we have $\Gamma_m^n, \Sigma \vdash_{\mathsf{s}} j :$ **int** and by induction again we reach the result.

Case IS-Assign. The premises to the rule are $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} i : D$ **ref** and $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} j : D$. Induction gives $\langle \rho, i\rangle \Downarrow_m^n \langle \sigma, x\rangle$ and $\Sigma \vdash \sigma$ **store** and $\Delta \subseteq \Sigma$. By weakening, we have $\Gamma_m^n, \Sigma \vdash_{\mathsf{s}} j : D$ and, by induction again, we get $\langle \sigma, j\rangle \Downarrow_m^n \langle \theta, v\rangle$ and $\Sigma' \vdash \theta$ **store** and $\Sigma \subseteq \Sigma'$. By rule IE-Assign we get $\langle \rho, i := j\rangle \Downarrow_m^n \langle \theta[x{:=}v], v\rangle$. To prove $\Sigma' \vdash \theta[x{:=}v]$ **store** we first apply preservation for evaluation using $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} i : D$ **ref** and $\langle \rho, i\rangle \Downarrow_m^n \langle \sigma, x\rangle$ and $\Delta \vdash \rho$ **store**, to obtain $\Delta \vdash_{\mathsf{s}} x : D$ **ref**. We then apply again preservation for evaluation to obtain $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} j : D$ and $\langle \sigma, j\rangle \Downarrow_m^n \langle \theta, v\rangle$ and $\Sigma \vdash \sigma$ **store**, to get $\Sigma' \vdash_{\mathsf{s}} v : D$. To conclude, apply store update using $\Delta \vdash_{\mathsf{s}} x : D$ **ref** and $\Sigma' \vdash_{\mathsf{s}} v : D$.

Case IS-Mkref. The premises to the rule are $D = E$ **ref** and $\Gamma \vdash i : E$. Induction gives $\langle \rho, i\rangle \Downarrow_m^n \langle \sigma, v\rangle$ and $\Sigma \vdash \sigma$ **store** and $\Gamma_m^n, \Sigma \vdash v : E$ and $\Delta \subseteq \Sigma$. We are then in the conditions of the evaluation rule IE-Mkref and obtain r. Conclude by taking $\Sigma = \Delta, r : D$ **ref** and $\sigma = (\rho, r := v)$.

Items 3 and 4. A straightforward induction, followed by the appropriate evaluation rule in Figure 3.

Item 5. Let $P = \langle \rho, e\rangle$. The only rule for process formation gives $\Delta \vdash \rho$ **store** and $\Gamma_m^n, \Delta \vdash e : T$. We proceed by induction on the structure of e.

Case $e = e_1; e_2$. If $e_1 = c$, then the result follows by rule PE-SemiCol. Otherwise, inversion gives $\Gamma_m^n, \Delta \vdash T \equiv T_1; T_2$ and $\Gamma_m^n, \Delta \vdash e_1 : T_1$ and $\Gamma_m^n, \Delta \vdash e_2 : T_2$. Induction gives either $\langle \rho, e_1\rangle \Downarrow_m^n \langle \sigma, c; e_1'\rangle$ or $\langle \rho, e_1\rangle \Downarrow_m^n \langle \sigma, \mathsf{skip}\rangle$. In the former case, $e_1; e_2$ reduces by rule PE-SemiSemi. For the latter case, induction also gives $\Gamma_m^n, \Sigma \vdash \langle \sigma, \mathsf{skip}\rangle : T_1$, from which we get $\Sigma \vdash \sigma$ **store** and $T_1 =$ **skip**. Weakening and rule E-Eq give $\Gamma_m^n, \Sigma \vdash e_2 : T$ and induction yields $\langle \sigma, e_2\rangle \Downarrow_m^n \langle \theta, e_2'\rangle$. The result follows by rule PE-SemiSkip.

Case $e =$ if p then e_1 else e_2. Inversion gives $\Gamma_m^n, \Delta \vdash T \equiv U$ and $\Gamma_m^n, \Delta \vdash_{\mathsf{s}} p$ **prop** and $\Gamma_m^n, \Delta \vdash e_1 : U$ and and $\Gamma_m^n, \Delta \vdash e_2 : U$. By item 4, we have $\langle \rho, p\rangle \Downarrow_m^n \langle \sigma, b\rangle$ and $\Sigma \vdash \sigma$ **store** and $\Delta \subseteq \Sigma$. Weakening and rule E-Eq give $\Gamma_m^n, \Sigma \vdash \langle \sigma, e_1\rangle : T$. Induction gives $\langle \sigma, e_1\rangle \Downarrow_m^n \langle \theta, e_1'\rangle$ and the result follows by rule PE-IfThen.

Case $e =$ for $x < i$ do e. Inversion gives $\Gamma_m^n, \Delta \vdash T \equiv \forall\, x{<}i.U$ and $\Gamma_m^n, \Delta, x : \{y : \mathbf{nat} \mid y < i\} \vdash e : U$ and $y \notin \mathrm{fv}(i)$. Agreement gives $\Gamma_m^n, \Delta \vdash i :$ **nat** and item 1 gives $\langle \rho, i\rangle \Downarrow_m^n \langle \rho, k\rangle$ and

$\Gamma^n_m, \Delta \vdash i = k$ **true**. When $k = 0$, the process reduces by rule PE-ForBase. In this case, we also know that $\Gamma^n_m, \Delta \vdash \mathsf{skip} : T \equiv \forall\, x{<}i.U$ from rule Eq-AllSkip given that $\Gamma^n_m, \Delta \vdash i = k = 0$ **true**. Otherwise, $k > 0$; hence , $\Gamma^n_m, \Delta \vdash k - 1 : \{y\colon \mathbf{nat} \mid y < i\}$. Then we may apply substitution to obtain $\Gamma^n_m, \Delta \vdash e\{k-1/x\} : U\{k-1/x\}$. Because $i > 0$, we also have $\Gamma^n_m, \Delta, x\colon \{y\colon \mathbf{nat} \mid y < i-1\} \vdash e : U$. Rule E-All gives $\Gamma^n_m, \Delta \vdash \mathsf{for}\ x < i-1\ \mathsf{do}\ e : \forall\, x{<}i-1.U$. Then, rule E-Semi yields $\Gamma^n_m, \Delta \vdash (e\{k-1/x\}; \mathsf{for}\ x < i-1\ \mathsf{do}\ e) : (U\{k-1/x\}; \forall\, x{<}i-1.U)$ and rule Eq-AllUnfold followed by Eq-All gives $\Gamma^n_m, \Delta \vdash (e\{k-1/x\}; \mathsf{for}\ x < i-1\ \mathsf{do}\ e) : (\forall\, x{<}i.T)$. We may now apply the induction hypothesis to obtain $\langle \rho, (e\{k-1/x\}; \mathsf{for}\ x < i-1\ \mathsf{do}\ e)\rangle \Downarrow^n_m \langle \sigma, f\rangle$. The result follows by rule PE-ForStep.

Case $e = \mathsf{lets}\ x\colon D = i\ \mathsf{in}\ e'$. Inversion gives $\Gamma \vdash T \equiv U$ and $\Gamma \vdash_{\mathsf{s}} i : D$ and $\Gamma, x\colon D \vdash e' : U$ and $x \notin \mathrm{fv}(T)$. Item 2 gives $\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, v\rangle$ and $\Sigma \vdash \sigma$ **store**. Substitution gives $\Gamma, x\colon D \vdash e'\{v/x\} : U\{v/x\}{=}T{\equiv}U$. Then induction gives $\langle \sigma, e'\{v/x\}\rangle \Downarrow^n_m \langle \theta, f\rangle$ and we are in the conditions of rule PE-Let. □

The following axiom determines the meaning of assertions of the form S **halted**.

$$\langle \rho_0, \mathsf{skip}\rangle \mid \cdots \mid \langle \rho_{n-1}, \mathsf{skip}\rangle\ \mathbf{halted}.$$

Our first main result ensures that well-typed programs are either halted or can be further reduced.

Theorem 4.14 (Progress). *If* $\Gamma \vdash R : T$, *then either* R **halted** *or* $R \longrightarrow S$.

Proof. If the expressions of all processes in R are skip, then R **halted** and we are done. Otherwise, there is one process in R whose expression is different from skip; let it be m. From the only rule for programs and for processes, we know that (1) $\Delta_k \vdash \rho_k$ **store** and (2) $\Gamma^n_k, \Delta_k \vdash e_k : T$, for all $0 \le k < n$. The proof proceeds by a case analysis on T; we sketch a few cases. By properties of type conversion (Lemma 4.10), we know that $\Gamma^n_k, \Delta_k \vdash e_k : U$ for U a type in normal form. We only need to analyse the six cases in Figure 11; we sketch a few cases.

Case $T = \mathbf{skip}$. Then $e \neq c; e'$ because all collective operations have types different from **skip**. Form (1), (2), and normalisation for evaluation (Lemma 4.13), we have $\langle \rho, \rho_m\rangle e_m \Downarrow^n_m Q$, which places us in the condition for rule PR-Proc to fire.

Case T is $\mathbf{reduce}\ i; U$. The canonical form of T is $e_k = e'_k; \mathsf{reduce}\ j_k\ j'_k\ j''_k; e''_k$ with $\Gamma^n_k, \Delta_k \vdash j_k = i$ **true**, where one or both e'_k and e''_k may be absent. If e'_k is present, then its type is **skip**. Now consider the subcase where all e'_k are absent and all e''_k are present. Applying inversion (Lemma 4.6) to (1) (first to semicolon and then to reduce) yields (3) $\Gamma^n_k, \Delta_k \vdash j_k : \mathbf{rank}$, (4) $\Gamma^n_k, \Delta_k \vdash_{\mathsf{s}} j'_k : \mathbf{int}$ and (5) $\Gamma^n_k, \Delta_k \vdash_{\mathsf{s}} j''_k : \mathbf{int\ ref}$. From (3) and normalisation for evaluation (Lemma 4.13), we know that (6) $\langle \rho_k, j_k\rangle \Downarrow^n_m \langle \rho_k, u_k\rangle$ and (7) $\Gamma^n_k, \Delta_k \vdash j_k = u_k$ **true**. From (3) and (7), we know that (8) u_k is a valid rank. From (4) and normalisation, we know that (9) $\langle \rho_k, j'_k\rangle \Downarrow^n_k \langle \theta_k, v_k\rangle$. From (5) and normalisation, we know that (10) $\langle \theta_k, j''_k\rangle \Downarrow^n_k \langle \sigma_k, w_k\rangle$ and (11) w_k is a reference. Now, (6), (8), (9), (10), and (11) place us in the conditions for rule PR-Red to fire. In all other subcases, let m be a process such that e'_m is present. In these cases, (1), (2), and normalisation yield $P_m \Downarrow^n_m Q$ and thus the program reduces by rule PR-Proc.

Case T is $\mathbf{message}\ i\ j; U$ The canonical forms of T say that there is a rank (call it l) such that (a) $e_l = e'_l; \mathsf{send}\ i_l\ j_l; e''_l$ with $\Gamma^n_m, \Delta_m \vdash i_l = m$ **true** and e'_l, e''_l optional, as well as that there is a rank (call it m) such that (b) $e_m = e'_m; \mathsf{receive}\ i_m\ j_m; e''_m$ with $\Gamma^n_l, \Delta_l \vdash i_m = l$ **true** and e'_m, e''_m optional. We thus see that there is a rank l whose send expression targets rank m, and there is a rank m whose receive expression reads from rank l. We proceed as for reduce and consider the subcase where e'_l and e'_m are absent and e''_l and e''_m are present. Inversion for send gives $\Gamma^n_l, \Delta_l \vdash i_l : \mathbf{rank}$ from which normalisation gives $\langle \rho_l, i_l\rangle \Downarrow^n_l \langle \rho_l, m\rangle$. Inversion also gives $\Gamma^n_l, \Delta_l \vdash_{\mathsf{s}} j_l : \mathbf{int}$ from

The size of a type $\boxed{\|T\| = n}$

$$\frac{}{\|\mathbf{skip}\| = 0} \qquad \frac{}{\|\mathbf{message}\ i\ j\| = 1} \qquad \frac{}{\|\mathbf{reduce}\ i\| = 1} \qquad \frac{\|T\| = n}{\|\forall\ x{<}i.T\| = i \cdot n}$$

$$\frac{\|T\| = n \quad \|U\| = m}{\|T;U\| = n + m} \qquad \frac{\|T\| = n}{\|\mathbf{broadcast}\ i\ x\colon D.T\| = 1 + n} \qquad \frac{\|T\| = n \quad \|U\| = m}{\|\mathbf{ifc}\ p\ \mathbf{then}\ T\ \mathbf{else}\ U\| = 1 + \max(n, m)}$$

The first non-skip operation $\boxed{\mathrm{fst}(e) = f}$

$$\frac{\mathrm{fst}(e) = \mathrm{skip}}{\mathrm{fst}(e;f) = \mathrm{fst}(f)} \qquad \frac{\mathrm{fst}(e) \neq \mathrm{skip}}{\mathrm{fst}(e;f) = \mathrm{fst}(e)} \qquad \frac{e \neq f;g}{\mathrm{fst}(e) = \mathrm{fst}(e)}$$

The size of a process $\boxed{\|P\| = n}$

$$\frac{\mathrm{fst}(e) = c}{\|\langle \rho, e\rangle\| = 0} \qquad \frac{\mathrm{fst}(e) = \mathrm{skip}}{\|\langle \rho, e\rangle\| = 0} \qquad \frac{\mathrm{fst}(e) \neq c, \mathrm{skip}}{\|\langle \rho, e\rangle\| = 1}$$

The size of a typed program $\boxed{\|\Gamma \vdash S : T\| = n}$

$$\frac{\Gamma \vdash T \Downarrow U}{\|\Gamma \vdash (P_1 \mid \cdots \mid P_n) : T\| = \|P_1\| + \cdots + \|P_n\| + (n+1) \cdot \|U\|}$$

Fig. 13. The size of a typed program.

which normalisation gives $\langle \rho_l, j_l\rangle \Downarrow_l^n \langle \theta_l, v\rangle$ with v an integer constant. In the case of receive, we obtain $\langle \rho_m, i_m\rangle \Downarrow_m^n \langle \rho_m, l\rangle$ and $\langle \rho_m, j_m\rangle \Downarrow_m^n \langle \theta_l, r\rangle$ with r a reference. Inversion also gives $\Gamma_l^n, \Delta_l \vdash i_l \neq l$ **prop**, from which we conclude $l \neq m$. We have thus established the premises to rule PR-Msg (or its dual). Again, similarly to reduce, in all other subcases reduction happens by rule PR-Proc. □

4.4 Strong Normalisation

The second main result states that each program reduction step preserves typing. In preparation for strong normalisation we also show that the size of a program strictly decreases with reduction. We define the size of program by taking into consideration both its type and the expressions in each process; the details are in Figure 13. The size of a program denotes an upper bound on the number of collective operations the program will perform when run. The size of a type may or may not decrease with reduction. In the below example, the size of the type reduces from 1 to 0,

$$(\Gamma^1 \vdash \langle r := 7, \mathrm{reduce}\ 0\ 23\ r\rangle : \mathbf{reduce}\ 0) \longrightarrow (\Gamma^1 \vdash \langle r := 23, \mathrm{skip}\rangle : \mathbf{skip})$$

but this is not necessarily the case, as witnessed by the next example, where the size of the type alone remains 0.

$$(\Gamma^1 \vdash \langle \varepsilon, \mathrm{if\ true\ then\ skip\ else\ skip}\rangle : \mathbf{skip}) \longrightarrow (\Gamma^1 \vdash \langle \varepsilon, \mathrm{skip}\rangle : \mathbf{skip})$$

For programs that may reduce locally, we take into consideration the size of expressions in processes. Processes whose first operation is a collective operation or skip have size 0, for they are ready to engage in a collective operation or are halted. All other programs have size 1 because they will eventually engage in a local reduction.

In general, type equivalence does not preserve the size of a type. Consider context Γ_0^3 for a program of *size* 3 and a process rank 0. Then, $\Gamma \vdash 1, 2 \neq \mathit{myrank}$ **true**, hence $\Gamma \vdash \mathbf{message}\ 1\ 2 \equiv \mathbf{skip}$ but $\|\mathbf{message}\ 1\ 2\| = 1 \neq 0 = \|\mathbf{skip}\|$. However, one can show that size is preserved for equivalent types in normal form.

Lemma 4.15 (Type Equivalence for Normal Forms Preserves Size). *If* $\Gamma \vdash T \equiv U$ *and* $\Gamma \vdash T$ **nf** *and* $\Gamma \vdash U$ **nf**, *then* $\|T\| = \|U\|$.

Proof. Properties of structural congruence (Lemma 4.11) give $\Gamma \vdash T \equiv_s U$. Complete the proof with a straightforward rule induction on this judgement. □

The following technical result is used in preservation for reduction (Theorem 4.17).

Lemma 4.16. *If* $\Gamma^n \vdash T$ **type**, Δ *contains only* **ref** *datatypes and* $\Gamma^n_m, \Delta \vdash T \equiv U$, *then* $\Gamma^n \vdash U$ **type**.

Proof. Noticing that variables of **ref** datatypes do not go into types, strengthening (Lemma 4.2) yields $\Gamma^n_k \vdash T \equiv U$. Recalling the definition of Γ^n_m (that is, $\Gamma^n, \mathit{myrank}\colon \{y\colon \mathbf{int} \mid y = m\}$), we can replace all occurrences of *myrank* in the derivation of $\Gamma^n_k \vdash T \equiv U$ by m to obtain a valid deviation (the actual proof is by induction). □

Equipped with the notion of the size of a program, we are in a position to state and prove the first main result of this section.

Theorem 4.17 (Preservation for Reduction). *If* $R \longrightarrow S$ *and* $\Gamma \vdash R : T$, *then* $\Gamma \vdash S : U$ *and* $\|\Gamma \vdash S : U\| < \|\Gamma \vdash R : T\|$.

Proof. By analysing the different cases for the $R \longrightarrow S$ hypothesis, using weakening, inversion of expression formation, store update, size preservation for type normalisation and preservation for evaluation (Lemmas 4.1, 4.5, 4.6, 4.10, 4.12 and 4.15). By inverting the only rules for programs and processes, we know that (1) $\Delta_k \vdash \rho_k$ **store**, (2) $\Gamma^n_k, \Delta_k \vdash e_k : T$, (3) $\Delta_0, \ldots, \Delta_{n-1}$ **context**, and (4) $\Gamma^n \vdash T$ **type**. We sketch a few representative cases.

Case PR-Red. In this case, $e_k = \mathsf{reduce}\ i_k\ j_k\ j'_k; e'_k$ and the premises to the rule are $\langle \rho_k, i_k\rangle \Downarrow^n_k \langle \rho_k, l\rangle$, (5) $\langle \rho_k, j_k\rangle \Downarrow^n_k \langle \sigma_k, v_k\rangle$, (6) $\langle \sigma_l, j'_l\rangle \Downarrow^n_l \langle \theta, r\rangle$ and $S = \langle \sigma_k, e'_k\rangle^{l-1}_{k=0} \mid \langle \theta[r{:=}v_0 + \cdots + v_{n-1}], e_l\rangle \mid \langle \sigma_k, e'_k\rangle^{n-1}_{k=l+1}$ and $\Gamma = \Gamma^n$. Inversion for sequential composition followed by inversion for reduce applied to (2) yields: (7) $\Gamma^n_k, \Delta_k \vdash T \equiv U; V$, (8) $\Gamma^n_k, \Delta_k \vdash_{\mathbf{s}} j_k : \mathbf{int}$, (9) $\Gamma^n_k, \Delta_k \vdash_{\mathbf{s}} j'_l : \mathbf{int\ ref}$, and (10) $\Gamma^n_k, \Delta_k \vdash e'_k : V$. From (1), (5), (8) and preservation for evaluation we have: (11) $\Sigma_k \vdash \sigma_k$ **store**, (12) $\Sigma_k \vdash v_k : \mathbf{int}$, (13) $\Delta_k \subseteq \Sigma_k$, and (14) the references in Σ but not in Δ are fresh. From (10), (13), and weakening, we get (15) $\Gamma^n_k, \Sigma_k \vdash e'_k : V$. From (11), (15), and the rule for process creation, we get (16) $\Gamma^n_k, \Sigma_k \vdash \langle \sigma_k, e'_k\rangle : V$ and this the typing for all process in S except l.

From (9) and weakening, we get (17) $\Gamma^n_k, \Sigma_k \vdash_{\mathbf{s}} j'_l : \mathbf{int\ ref}$. From (6), (17), and preservation for evaluation, we have: (18) $\Theta \vdash \theta$ **store**, (19) $\Theta \vdash r : \mathbf{int\ ref}$, (20) $\Sigma_l \subseteq \Theta$, and (21) the references in Θ but not in Σ_l are fresh. From (12), (13), (20), and weakening and rule IS-Op, we obtain: (22) $\Theta \vdash v_0 + \cdots + v_{n-1} : \mathbf{int}$. From (18), (19), (22), and store update, we get: (23) $\Theta \vdash \theta[r{:=}v_0 + \cdots + v_{n-1}]$ **store**. From (15), (20), and weakening, we have (24) $\Gamma^n_k, \Theta \vdash e'_k : V$. From (23), (24), and the rule for process creation, we get (25) $\Gamma^n_k, \Theta \vdash \langle \theta[r{:=}v_0 + \cdots + v_{n-1}], e_l\rangle : V$, the typing for process l in S.

From (3), (14), (21), we have (26) $\Sigma_0, \ldots, \Sigma_{l-1}, \Theta, \Sigma_{l+1}, \ldots, \Sigma_{n-1}$ **context**. From (4), (7), and Lemma 4.16, we obtain (27) $\Gamma^n \vdash V$ **type**. Finally, from (16), (25)–(27), and the rule for program reduction we have $\Gamma^n \vdash S : V$.

We still need to show that $\|\Gamma^n \vdash \langle \theta_k, e_k\rangle^{l-1}_{k=0} \mid \langle \theta'_l[r{:=}v_0 + \cdots + v_{n-1}], e_l\rangle \mid \langle \theta_k, e_k\rangle^{n-1}_{k=l+1} : V\| < \|\Gamma^n \vdash \langle \rho_k, \mathsf{reduce}\ i_k\ j_k\ j'_k; e_k\rangle^{n-1}_{k=0} : T\|$. By definition, we have $\|\langle \rho_k, \mathsf{reduce}\ i_k\ j_k\ j'_k; e_k\rangle^{n-1}_{k=0}\| = n \times \|\langle \rho_k, \mathsf{reduce}\ i_k\ j_k\ j'_k; e_k\rangle^{n-1}_{k=0}\| + (n+1) \cdot \|T_{\mathrm{nf}}\|$. Since $\mathrm{fst}(\mathsf{reduce}\ i_k\ j_k\ j'_k; e_k)$ is a collective operation, we know that $\|\langle \rho_k, \mathsf{reduce}\ i_k\ j_k\ j'_k; e_k\rangle\| = 0$ for $0 \le k < n-1$. So, the size of the right member of the inequality is $(n+1) \cdot \|T_{\mathrm{nf}}\|$. We can easily show that structurally equivalent types have the same size, therefore $(n+1) \cdot \|T_{\mathrm{nf}}\| = (n+1) + (n+1) \cdot \|V_{\mathrm{nf}}\|$, since $\|\mathbf{reduce}\ i\| = 1$, by Lemma 4.15. On the other hand, $\|\Gamma^n \vdash \langle \theta_k, e_k\rangle^{l-1}_{k=0} \mid \langle \theta'_l[r{:=}v_0 + \cdots + v_{n-1}], e_l\rangle \mid \langle \theta_k, e_k\rangle^{n-1}_{k=l+1} : V\| \le n + (n+1) \cdot \|V\|$,

considering the worst-case scenario in which all n terms of the process are of size 1. Nevertheless, the size is still smaller than $(n+1)+(n+1)\cdot\|V\|$, and we conclude the case.

Case PR-Msg. In this case, S is $S_1 \mid \langle \sigma_l, e_l\rangle \mid S_2 \mid \langle \sigma_m[r{:=}k], e_m\rangle \mid S_3$ and relevant e_k expressions are $\mathsf{send}\ i_l\ j_l; e_l$ and $\mathsf{receive}\ i_m\ j_m; e_l$. The relevant premises to the rule are (5) $\langle \rho_l, j_l\rangle \Downarrow^n_l \langle \sigma_l, k\rangle$ and (6) $\langle \rho_m, j_m\rangle \Downarrow^n_m \langle \sigma_m, r\rangle$. Inversion for semicolon followed by inversion for send applied to (2) yields: (7) $\Gamma^n_k, \Delta_k \vdash T \equiv U; V$, (8) $\Gamma^n_k, \Delta_k \vdash_{\mathsf{s}} j_l : \mathbf{int}$ and (9) $\Gamma^n_k, \Delta_k \vdash e_l : V$. From (1), (6), and preservation for evaluation, we have (10) $\Delta_l \vdash \sigma_l$ **store**. From (9), (10), and the rule for processes, we have (11) $\Gamma^n_l, \Sigma_l \vdash \langle \sigma_l, e_l\rangle : V$, one of the processes in S.

Inversion for semicolon followed by inversion for receive yields (12) $\Gamma^n_k, \Delta_k \vdash_{\mathsf{s}} j_m : \mathbf{int\ ref}$ and (13) $\Gamma^n_k, \Delta_k \vdash e_m : V$. From (1), (6), (12), and preservation for evaluation, we obtain (14) $\Delta_m \vdash \sigma_m$ **store** and (15) $\Delta_m \vdash_{\mathsf{s}} r : \mathbf{int\ ref}$. From (1), (5), (8), and again preservation for evaluation, we obtain $\Delta_l \vdash_{\mathsf{s}} k : \mathbf{int}$. But k is a constant, hence (16) $\Delta_m \vdash_{\mathsf{s}} k : \mathbf{int}$. From (14)–(16) and store update, we have (17) $\Delta_m \vdash [\sigma_m{:=}r]k$ **store**. From (12), (17), and the rule for processes, we have (18) $\Gamma^n_l, \Sigma_l \vdash \langle [\sigma_m{:=}r]k, e_m\rangle : V$, another process in S.

For the remaining processes, S_1, S_2, S_3, we further have $k \neq l$ and $k \neq m$. The ranks of these processes place us in the conditions to apply rule Eq-Proj to obtain $\Gamma^n_k, \Delta_k \vdash \mathbf{message}\ l\ m \equiv \mathbf{skip}$ which, from (7) and the rules of type equivalence, yields (19) $\Gamma^n_k, \Sigma_k \vdash P_k : V$, for all P_k in S_1, S_2, S_3. Finally, we apply Lemma 4.16 to (11), (18), and (19) obtain $\Gamma^n \vdash S : V$.

Finally, we show the following inequality $\|\Gamma^n \vdash S_1 \mid \langle \theta_l, e_l\rangle \mid S_2 \mid \langle \theta_m[r{:=}v], e_m\rangle \mid S_3 : V\| < \|\Gamma^n \vdash S_1 \mid \langle \rho_l, \mathsf{send}\ i_l\ j_l; e_l\rangle \mid S_2 \mid \langle \rho_m, \mathsf{receive}\ i_m\ j_m; e_m\rangle \mid S_3 : V\|$. Towards this end, we show that $\|\langle \theta_l, e_l\rangle\| + \|\langle \theta_m[r{:=}v], e_m\rangle\| + (n+1)\|V_{\mathrm{nf}}\| < 2 + (n+1)\|T_{\mathrm{nf}}\|$, where $\Gamma \vdash V \Downarrow V_{\mathrm{nf}}$, $\Gamma \vdash T \Downarrow T_{\mathrm{nf}}$ and $\|\langle \rho_l, \mathsf{send}\ i_l\ j_l; e_l\rangle\| = 1$ and $\|\langle \rho_m, \mathsf{receive}\ i_m\ j_m; e_m\rangle\| = 1$, since the remaining terms are common in both expressions. Given that expressions e_l and e_m are unknown, we assume the worst-case scenario for their sizes and take $\|\langle \theta_l, e_l\rangle\| = 1$ and $\|\langle \theta_m[r{:=}v], e_m\rangle\| = 1$. So, we have to show that $\|V_{\mathrm{nf}}\| < \|T_{\mathrm{nf}}\|$. But, $\Gamma^n \vdash U; V \equiv T$ and $\Gamma^n \vdash U \equiv \mathbf{message}\ m\ l$. The rules for type conversion ensure that the normal form of $U; V$ is $U; V_{\mathrm{nf}}$. Conclude the case by recalling that type equivalence for normal forms preserves size and the fact that $\|\mathbf{message}\ m\ l\| = 1$.

Case PR-Proc. The premises to the rule say that there is a process P in S of, say, rank m such that $P \Downarrow^n_m Q$. The premises to the only rule for programs include $\Gamma^n_m, \Delta_m \vdash P : T$. Then, preservation for evaluation yields $\Gamma^n_m, \Sigma \vdash Q : T$ and Lemma 4.16 ensures that $\Gamma^n \vdash Q : T$. Conclude with the rule for programs.

For $\|\Gamma^n \vdash S_1 \mid P \mid S_2 : T\| < \|\Gamma^n \vdash S_1 \mid Q \mid S_2 : T\|$, we show that $\|Q\| < \|P\|$. From the premises of rule PR-Proc, we know that Q is either skip or of the from $c; f$. So, $\|Q\| = 0$ follows directly from the definition. On the other hand, $\|P\| = 1$, since, for the reduction rule to apply, it must be the case that $P = \langle \rho, e\rangle$ and $\mathrm{fst}(e) \neq c$ and $e \neq \mathsf{skip}$ and that concludes the proof. □

Process evaluation is deterministic; that should be apparent from the rules in Figure 3. Program reduction, on the other hand, is not deterministic. For example, processes that do not synchronise may evolve locally; program reduction for such processes encompasses all possible interleavings. In the case of message exchange, two processes that reach a rendez-vous (processes l and m in rule PR-Msg, Figure 4) may reduce immediately or else be delayed by message exchanges involving different processes or by local reductions in other processes (via rule PR-Proc in the same figure). In any case, if a program is capable of two distinct reductions, then there exists a common program that is reachable from both results.

Theorem 4.18 (Confluence).

(1) *If* $\langle \rho, i\rangle \Downarrow^n_m \langle \tau, j\rangle$ *and* $\langle \rho, i\rangle \Downarrow^n_m \langle \sigma, k\rangle$, *then* $\tau = \sigma$ *and* $j = k$.
(2) *If* $\langle \rho, p\rangle \Downarrow^n_m \langle \tau, q\rangle$ *and* $\langle \rho, p\rangle \Downarrow^n_m \langle \sigma, r\rangle$, *then* $\tau = \sigma$ *and* $q = r$.

Algorithmic type equivalence $\boxed{\Gamma \vdash_a T \equiv T}$

$$\frac{\Gamma \vdash T \Downarrow V \quad \Gamma \vdash U \Downarrow W \quad \Gamma \vdash V \equiv_s W}{\Gamma \vdash_a T \equiv U}$$

Fig. 14. Algorithmic type equivalence.

(3) *If* $\langle \rho, e \rangle \Downarrow_m^n \langle \tau, f \rangle$, *then* e *is* skip *or of the form* $c; g$.
(4) *If* $P \Downarrow_m^n Q$ *and* $P \Downarrow_m^n Q'$, *then* $Q = Q'$.
(5) *If* $S_1 \longrightarrow S_2$ *and* $S_1 \longrightarrow S_3$, *then either* $S_2 = S_3$ *or* $S_2 \longrightarrow S_4$ *and* $S_3 \longrightarrow S_4$.

Proof. *Items 1 and 2.* By induction on the index term and the proposition, noting that there is at most one evaluation rule per index term and proposition.

Item 3. A straightforward rule induction on process evaluation.

Item 4. By induction on the expression in the process. Processes featuring expressions skip, let, lets, and c have a single evaluation rule. Processes with if and for have two evaluation rules each with mutual exclusive premises (from items 1 and 2). Sequential composition comprises three rules. One is for expressions $c; f$, the other two for expression $e; f$ with $e \neq c$ (one of the rule features an explicit premise $e \neq c$, the other requires that e evaluate to skip which implies $e \neq c$) and the premises are mutual exclusive by item 3.

Item 5. Rules PR-Red, PR-BCast and PR-IfcT include all processes and thus $S_2 = S_3$. It remains to analyse rules PR-Msg and PR-Proc. Given item 4, in each of the four combinations (PR-Msg-PR-Msg, PR-Msg-PR-Proc, PR-Proc-PR-Msg, and PR-Proc-PR-Proc) the two reductions involve distinct processes, so that the premises to the reduction ending in program S_3 remain untouched, allowing this reduction to proceed from state S_2. □

Corollary 4.19 (Strong Normalisation). *If* $\Gamma \vdash R : T$, *then every reduction path starting from program* R *ends in a program* S *such that* S **halted**.

Proof. From progress, preservation and confluence (Theorems 4.14, 4.17 and 4.18) and the fact that all programs with size 0 are of the form $\langle \rho_0, \text{skip} \rangle \mid \cdots \mid \langle \rho_{n-1}, \text{skip} \rangle$, hence are halted. □

5 ALGORITHMIC TYPE CHECKING

This section introduces sound and complete algorithms for checking type equivalence and expression formation.

5.1 Checking Type Equivalence

Algorithmic type equivalence, in Figure 14, works by converting types to normal forms. Given two types, the algorithm converts each to a normal form and then compares the thus obtained types for structural type equivalence. Equipped with the previous results, we are in a position to prove the main result of this section.

Theorem 5.1 (Correctness of Algorithmic Type Equivalence). $\Gamma \vdash_a T \equiv U$ *if and only if* $\Gamma \vdash T \equiv U$.

Proof. Soundness. The premises to the only rule that establishes the hypothesis are $\Gamma \vdash T \Downarrow T'$ and $\Gamma \vdash U \Downarrow U'$ and $\Gamma \vdash T' \equiv_s U'$. Using properties of type conversion and structural equivalence (Lemma 4.10-item 3 and Lemma 4.11-item 2), we have $\Gamma \vdash T \equiv T'$, $\Gamma \vdash U \equiv U'$, and $\Gamma \vdash T' \equiv U'$. The result follows by symmetry and transitivity of type equivalence (Lemma 3.1).

Completeness. From the hypothesis and agreement (Lemma 4.8), we know that $\Gamma \vdash T$ **type** and $\Gamma \vdash U$ **type**. Using properties of type conversion (Lemma 4.10-item 4) we get $\Gamma \vdash T \Downarrow T'$ and

$\Gamma \vdash U \Downarrow U'$ and $\Gamma \vdash T'$ **nf** and $\Gamma \vdash U'$ **nf** and $\Gamma \vdash T \equiv T'$ and $\Gamma \vdash U \equiv U'$. Now we use the symmetry and transitivity of type equivalence (Lemma 3.1) to obtain $\Gamma \vdash T' \equiv U'$. Using properties of structural equivalence (Lemma 4.11-item 3), we get $\Gamma \vdash T' \equiv_s U'$. We have established the three premises to the only rule deducing $\Gamma \vdash_a T \equiv U$. □

5.2 Checking Expression Formation

In a concrete implementation of our system, programmers write a series of expressions of length *size*, not necessarily distinct. Then, a program is assembled from a series of processes, each composed of an expression and an empty store. We must then provide for algorithmic checking for expressions only, an easy task once equipped with algorithmic type equivalence.

We first notice that if proposition entailment is decidable, then datatype formation $\Gamma \vdash D$ **dtype**, context formation Γ **context**, datatype subtyping $\Gamma \vdash D <: E$ and equivalence $\Gamma \vdash D \equiv E$, type formation $\Gamma \vdash T$ **type** and store-related proposition formation $\Gamma \vdash_s p$ **prop** index term formation $\Gamma \vdash_s i : D$ (in Figures 5, 6 and 8) are all decidable relations, given that the rules in each of these judgements are directed by the syntax. For these relations, we shall not require algorithmic counterparts. Figure 15 introduces the algorithmic counterparts of the remaining relations.

Due to the presence of subtyping in datatype formation (rule I-Sub in Figure 5) and that of type equivalence in expression formation (rule E-Eq in Figure 9), algorithmic index term and algorithmic expression formation are bidirectional systems [13, 14, 60]. Index term formation is split into a *synthesise* function ($\Gamma \vdash i \Rightarrow D$) and a *check against* function ($\Gamma \vdash i \Leftarrow D$), and similarly for expression formation ($\Gamma \vdash e \Leftarrow T$ and $\Gamma \vdash e \Rightarrow T$). In general, all objects in these judgements are considered as input, except those at right of a right arrow (that is, the D and T in the synthesis functions) which are considered output.

The rules in the datatype synthesis function are obtained from the syntax-directed counterparts (I-Var, I-Int, I-Op in Figure 5) by replacing the colons in the premises by $\Leftarrow$ or $\Rightarrow$ as appropriate. Notice that the rules for integer values and arithmetic operations yield refinement types that describe subtypes of all possible datatypes for the index term, thus compensating for the absence of rule I-Refin. There is one only rule for the check-against datatype function: it synthesises a datatype E for the given index term i and checks that E is a subtype of the given D.

Algorithmic expression formation is obtained similarly. The only rule for check-against calls expression synthesis to obtain a type U and checks whether the given type T and U are equivalent, using algorithmic type equivalence (Figure 14). The rules for type synthesis are obtained from the syntax-directed rules in Figure 10 by replacing the semicolons in the premises by $\Leftarrow$ or $\Rightarrow$ as appropriate.

Proposition entailment, $\Gamma \vdash p$ **true**, calls logical deducibility, $\Gamma \vDash p$. Deducibility is usually transformed into a verification condition and passed to an SMT solver. Decidability of type checking ultimately depends on the decidability of logical deducibility, and this depends on the concrete language of propositions (Figure 5). The inclusion of multiplicative integer operators in the language of propositions (as instances of the generic binary operator $\oplus$ in Figure 5) renders logical deducibility undecidable in general. We sometimes use modular arithmetic in examples and that can pose problems to SMT solvers when the value of *size* cannot be determined.

We can now show that algorithmic type checking is sound and complete with respect to its declarative counterpart.

Theorem 5.2 (Correctness of Algorithmic Type Checking).

(1) $\Gamma \vdash i \Leftarrow D$ *if and only if* $\Gamma \vdash i : D$.
(2) *If* $\Gamma \vdash i \Rightarrow D$, *then* $\Gamma \vdash i : D$.
(3) *If* $\Gamma \vdash i : D$, *then* $\Gamma \vdash i \Rightarrow E$ *and* $\Gamma \vdash E <: D$.

Datatype synthesis $\boxed{\Gamma \vdash i \Rightarrow D}$

$$\frac{\Gamma\ \textbf{context} \quad x : D \in \Gamma}{\Gamma \vdash x \Rightarrow D} \qquad \frac{\Gamma\ \textbf{context}}{\Gamma \vdash n \Rightarrow \{x : \textbf{int} \mid x = n\}} \qquad \frac{\Gamma \vdash i \Leftarrow \textbf{int} \quad \Gamma \vdash j \Leftarrow \textbf{int}}{\Gamma \vdash i \oplus j \Rightarrow \{x : \textbf{int} \mid x = i \oplus j\}}$$

Check against a datatype $\boxed{\Gamma \vdash i \Leftarrow D}$

$$\frac{\Gamma \vdash i \Rightarrow E \quad \Gamma \vdash E <: D}{\Gamma \vdash i \Leftarrow D}$$

Type synthesis $\boxed{\Gamma \vdash e \Rightarrow T}$

$$\frac{\Gamma\ \textbf{context}}{\Gamma \vdash \mathsf{skip} \Rightarrow \textbf{skip}} \qquad \frac{\Gamma \vdash \mathit{myrank} \Leftarrow \textbf{rank} \quad \Gamma \vdash i \Leftarrow \textbf{rank} \quad \Gamma \vdash i \neq \mathit{myrank}\ \textbf{true} \quad \Gamma \vdash_{\mathbf{s}} j : \textbf{int}}{\Gamma \vdash \mathsf{send}\ i\ j \Rightarrow \textbf{message}\ \mathit{myrank}\ i}$$

$$\frac{\Gamma \vdash i \Leftarrow \textbf{rank} \quad \Gamma \vdash \mathit{myrank} \Leftarrow \textbf{rank} \quad \Gamma \vdash i \neq \mathit{myrank}\ \textbf{true} \quad \Gamma \vdash_{\mathbf{s}} j : \textbf{int ref}}{\Gamma \vdash \mathsf{receive}\ i\ j \Rightarrow \textbf{message}\ i\ \mathit{myrank}}$$

$$\frac{\Gamma \vdash i \Leftarrow \textbf{rank} \quad \Gamma \vdash_{\mathbf{s}} j : \textbf{int} \quad \Gamma \vdash_{\mathbf{s}} k : \textbf{int ref}}{\Gamma \vdash \mathsf{reduce}\ i\ j\ k \Rightarrow \textbf{reduce}\ i}$$

$$\frac{\Gamma \vdash i \Leftarrow \textbf{rank} \quad \Gamma \vdash_{\mathbf{s}} j : D \quad \Gamma, x : D \vdash e \Rightarrow T}{\Gamma \vdash \mathsf{let}\ x : D = \mathsf{broadcast}\ i\ j\ \mathsf{in}\ e \Rightarrow \textbf{broadcast}\ i\ x : D.T}$$

$$\frac{\Gamma, \_ : \{p\} \vdash e \Rightarrow T \quad \Gamma, \_ : \{\neg p\} \vdash f \Rightarrow U}{\Gamma \vdash \mathsf{ifc}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f \Rightarrow \textbf{ifc}\ p\ \textbf{then}\ T\ \textbf{else}\ U}$$

$$\frac{\Gamma \vdash e \Rightarrow T \quad \Gamma \vdash f \Rightarrow U}{\Gamma \vdash e; f \Rightarrow T; U} \qquad \frac{\Gamma, x : \{y : \textbf{nat} \mid y < i\} \vdash e \Rightarrow T \quad y \notin \mathrm{fv}(i)}{\Gamma \vdash \mathsf{for}\ x < i\ \mathsf{do}\ e \Rightarrow \forall\ x{<}i.T}$$

$$\frac{\Gamma \vdash_{\mathbf{s}} p\ \textbf{prop} \quad \Gamma \vdash e \Rightarrow T \quad \Gamma \vdash f \Leftarrow T}{\Gamma \vdash \mathsf{if}\ p\ \mathsf{then}\ e\ \mathsf{else}\ f \Rightarrow T} \qquad \frac{\Gamma \vdash_{\mathbf{s}} i : D \quad \Gamma \vdash e \Rightarrow T \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash \mathsf{lets}\ x : D = i\ \mathsf{in}\ e \Rightarrow T}$$

$$\frac{\Gamma \vdash i \Rightarrow D \quad \Gamma, x : \{y : D \mid y = i\} \vdash e \Rightarrow T \quad x \notin \mathrm{fv}(T) \quad y \notin \mathrm{fv}(i)}{\Gamma \vdash \mathsf{let}\ x : D = i\ \mathsf{in}\ e \Rightarrow T}$$

Check against a type $\boxed{\Gamma \vdash e \Leftarrow T}$

$$\frac{\Gamma \vdash e \Rightarrow U \quad \Gamma \vdash_a U \equiv T}{\Gamma \vdash e \Leftarrow T}$$

Fig. 15. Algorithmic type checking.

(4) $\Gamma \vdash e \Leftarrow T$ *if and only if* $\Gamma \vdash e : T$.
(5) *If* $\Gamma \vdash e \Rightarrow T$, *then* $\Gamma \vdash e : T$.
(6) *If* $\Gamma \vdash e : T$, *then* $\Gamma \vdash e \Rightarrow U$ *and* $\Gamma \vdash T \equiv U$.

Proof. A simple, yet long, proof by mutual rule induction on the hypotheses, using transitivity of datatype subtyping and type equivalence, weakening and agreement (Lemmas 3.1, 4.1 and 4.3). We sketch a few cases.

Item 2, rule I-Int. Then $i = n$ and $D = \textbf{int}$. Agreement gives Γ **context** and the rule for integer values give $\Gamma \vdash n \Rightarrow \{x : \textbf{int} \mid x = n\}$. With rule S-Int we have $\Gamma \vdash \textbf{int} <: \textbf{int}$ and with rule S-RefinL we have $\Gamma \vdash \{x : \textbf{int} \mid x = n\} <: \textbf{int}$. Conclude with the only rule for checking against a datatype.

Item 3, rule I-Op. Then $i = i_1 \oplus i_2$ and $D = \textbf{int}$. Induction gives $\Gamma \vdash i_k : E_k$ and $\Gamma \vdash E_k <: \textbf{int}$, for $k = 1, 2$. Rule I-Sub gives $\Gamma \vdash i_k : \textbf{int}$. Weakening and rule P-Op gives $\Gamma, x : \textbf{int} \vdash i_1 \oplus i_2\ \textbf{prop}$. Agreement and rule S-Int give $\Gamma \vdash \textbf{int} <: \textbf{int}$. Conclude with rule S-RefinR.

Item 6, rule E-Bcast. Then $e = \mathsf{let}\ x\colon D = \mathsf{broadcast}\ i\ j\ \mathsf{in}\ f$ and $T = \mathbf{broadcast}\ i\ x\colon D.U$. Premises are $\Gamma \vdash i : \mathbf{rank}$ and $\Gamma \vdash_s j : D$ and $\Gamma, x\colon D \vdash f : U$. Induction gives $\Gamma \vdash i \Leftarrow \mathbf{rank}$ and $\Gamma \vdash_s j : D$ and $\Gamma, x\colon D \vdash f \Rightarrow V$ and $\Gamma \vdash U \equiv V$. The algorithmic rule for broadcast gives $\Gamma \vdash \mathsf{let}\ x\colon D = \mathsf{broadcast}\ i\ j\ \mathsf{in}\ e : \mathbf{broadcast}\ i\ x\colon D.V$ and Eq-Bcast yields $\Gamma \vdash \mathbf{broadcast}\ i\ x\colon D.V \equiv T$. □

We complete this section with a brief discussion on the running time of type checking. The procedures for checking the formation of index terms, propositions, datatypes and expressions make one call to each subterm. On what concerns type equivalence ($\Gamma \vdash_a T \equiv U$), structural type equivalence checking ($\Gamma \vdash T \equiv_s U$) is linear on the size of the types. Type conversion ($\Gamma \vdash T \Downarrow U$) makes one call to each subterm of the input type, excepted for primitive recursion. In this case, the number of calls if constant, for the number of goals $\Gamma \vDash i$, $\Gamma \vDash i - 1$, ... that succeed is finite (and fixed by Γ). There remains logical deducibility, $\Gamma \vDash p$. Even though the logical deducibility for equality, uninterpreted functions and linear arithmetics is NP-hard, state-of-the-art solvers for this theory can be quite efficient for the simple queries that arise in the context of type checking ParTypes expressions.

6 EXTENSIONS

This section sketches the introduction of a number of extensions to expressions and types. The extensions allow modelling many protocols usually found in the literature of parallel programming. We illustrate the new constructions with examples.

6.1 Scattering and Gathering Arrays

This section introduces arrays and operations for communicating arrays. The scatter operator distributes arrays among processes. A root process proposes an array. The array is divided in size equal parts, each delivered to a different process. Each process (including the root) puts forward a reference (to an array) where to receive its part. The gather operator performs the inverse operation: each process proposes an array and the root process collects all parts into a large array. In both cases, the typing rules enforce that the length of the large array divides size , the number of processes.

We start with a few examples. The first is the *parallel scalar product* (or dot product) of two arrays. The code is as follows:

```
1 let n:{z:nat|z%size==0} = broadcast 0 read_problem_size()
2     x:int[n/size] ref = mkref (mkarray (n/size) 0)
3     y:int[n/size] ref = mkref (mkarray (n/size) 0) in
4 scatter 0 x read_array(n);
5 scatter 0 y read_array(n);
6 let result:int ref = mkref 0 in
7 reduce 0 serial_dot(x, y) result;
8 if myrank == 0 then print(!result)
```

The root process (0 in our example) starts by reading the problem size (the length of the arrays). It then broadcasts this value to all processes, so that each process may allocate space for their local arrays. Recall that the semantics of broadcast ensures that read_problem_size () is evaluated by rank 0 alone (rule PR-BCast in Figure 4). The root process then reads and scatters the two arrays among all processes including itself (lines 4–5). Similarly to broadcast , the read_array (n) expression is only evaluated by the root process. Each process stores its sub-array locally, in x and y. Finally, each process computes a sequential version of the dot product

on their local arrays, and the root collects the result via a reduce operation (line 7). The length of the arrays must evenly divide the number of processes, so that each process receives parts of the original arrays of equal size. We capture this requirement with the refined type for n in line 1.

The following type describes the program's protocol.

```
1 broadcast 0 n:{z:nat|z%size==0}.
2 scatter 0 int[n];
3 scatter 0 int[n];
4 reduce 0
```

We can easily notice the alignment between the code and the type. For example, the broadcast operation in line 1 of the code corresponds to the **broadcast** type in line 1 of the type, and similarly for scatter and reduce. The type of the value broadcast, {z : **nat** | z% size ==0} is the same in both the code and the type. Then, the array declarations in lines 2–3 make sure that the scatter operations type check against the **scatter** types: function read_array returns an int array of size n that is scattered as n/ size arrays allocated on each process.

Our next example takes advantage of dependent types to implement *topology passing* algorithms. The topology underlying the protocol for the finite differences is that of a ring: a linear array with a wraparound link. If a different mapping of ranks to processes is to be used, a new protocol must be derived. It turns out that the language of datatypes is flexible enough to encode topologies in integer arrays. Such a topology may then be made known to all processes in such a way that processes exchange messages as per the particular topology.

A datatype of the form

$$\{a : \mathbf{nat}[size] \mid \forall i.0 \leq i < size \rightarrow a[i] < size \wedge a[i] \neq i\}$$

encodes a one-dimensional network topology, where $j = a[i]$ means j is a *direct neighbor* of i: each node has exactly one direct neighbor (a number between 0 and $size - 1$) that is different from itself. At this point, we assume that the language of propositions is extended with quantifiers. Call such a datatype D and consider the type below. The topology is first distributed among all processes by the root process (0). Thereafter, each process can exchange a message with its neighbour.

```
1 broadcast 0 topology: D.
2 forall i<len(topology).
3   message i topology[i]
```

For example, a right-to-left ring topology of length five can be encoded as [4, 0, 1, 2, 3].

The encoding above requires all processes to have exactly one direct neighbour. How can one encode a topology when this is not the case? Think of a star or a line. One possibility is to weaken the above condition on the elements of the array, while strengthening the subsequent message passing loop. We could, for example, drop the restriction that $a[i] \neq i$ and encode a right-to-left line of length five as [0, 0, 1, 2, 3], a 0-centred star as [0, 0, 0, 0, 0], and a full binary 0-rooted tree of depth 3 as [0, 0, 0, 1, 1, 2, 2]. In all cases, the topology maps 0 to 0. And this causes a problem if we try to send a message from i to topology [i], as in the above example. Here's the example rewritten with a conditional type that guards against ranks without neighbours:

```
1 broadcast 0 topology: nat[size].
2 forall i<len(topology).
3   ifc 0≤topology[i] && topology[i]<size && i!=topology[i] then
4     message i topology[i]
```

$$
\begin{array}{rrcl}
\text{Index terms} & i, j & ::= & \ldots \mid [i, \ldots, i] \mid \mathsf{len}(i) \mid i[i] \mid \mathsf{mkarray}\ i\ i \mid i[i] := i \\
\text{Datatypes} & D, E & ::= & \ldots \mid D[] \\
\text{Types} & T, U & ::= & \ldots \mid \mathbf{scatter}\ i\ D \mid \mathbf{gather}\ i\ D \\
\text{Expressions} & e, f & ::= & \ldots \mid \mathsf{scatter}\ i\ i\ i \mid \mathsf{gather}\ i\ i\ i
\end{array}
$$

Index term formation $\boxed{\Gamma \vdash i : D} \quad \boxed{\Gamma \vdash_{\mathsf{s}} i : D}$

$$
\frac{\Gamma \vdash i_1 : D \quad \cdots \quad \Gamma \vdash i_n : D}{\Gamma \vdash [i_1, \ldots, i_n] : D[n]} \qquad \frac{\Gamma \vdash i : D[]}{\Gamma \vdash \mathsf{len}(i) : \mathbf{nat}} \qquad \frac{\Gamma \vdash i : D[n] \quad \Gamma \vdash 0 \le j < n\ \mathbf{true}}{\Gamma \vdash i[j] : D}
$$

$$
\frac{\Gamma \vdash i : \mathbf{nat} \quad \Gamma \vdash_{\mathsf{s}} j : D}{\Gamma \vdash_{\mathsf{s}} \mathsf{mkarray}\ i\ j : D[i]} \qquad \frac{\Gamma \vdash_{\mathsf{s}} i : D[n]\ \mathbf{ref} \quad \Gamma \vdash 0 \le j < n\ \mathbf{true} \quad \Gamma \vdash_{\mathsf{s}} k : D}{\Gamma \vdash_{\mathsf{s}} i[j] := k : D}
$$

$$
\frac{\Gamma \vdash_{\mathsf{s}} i : D[]}{\Gamma \vdash_{\mathsf{s}} \mathsf{len}(i) : \mathbf{nat}} \qquad \frac{\Gamma \vdash_{\mathsf{s}} i : D[n] \quad \Gamma \vdash 0 \le j < n\ \mathbf{true}}{\Gamma \vdash_{\mathsf{s}} i[j] : D}
$$

Datatype formation and subtyping $\boxed{\Gamma \vdash D\ \mathbf{dtype}} \quad \boxed{\Gamma \vdash D <: D}$

$$
\frac{\Gamma \vdash D\ \mathbf{dtype}}{\Gamma \vdash D[]\ \mathbf{dtype}} \qquad \frac{\Gamma \vdash D \equiv E}{\Gamma \vdash \Gamma \vdash D[] <: E[]}
$$

Type formation $\boxed{\Gamma \vdash T\ \mathbf{type}}$

$$
\frac{\Gamma \vdash i : \mathbf{rank} \quad \Gamma \vdash D <: \{x : E[n] \mid n\%size = 0\}}{\Gamma \vdash \mathbf{scatter}\ i\ D\ \mathbf{type}} \qquad \frac{\Gamma \vdash i : \mathbf{rank} \quad \Gamma \vdash D <: \{x : E[n] \mid n\%size = 0\}}{\Gamma \vdash \mathbf{gather}\ i\ D\ \mathbf{type}}
$$

Expression formation $\boxed{\Gamma \vdash e : T}$

$$
\frac{\Gamma \vdash i : \mathbf{rank} \quad \Gamma \vdash_{\mathsf{s}} j : D[n]\ \mathbf{ref} \quad \Gamma \vdash_{\mathsf{s}} k : D[n * size]}{\Gamma \vdash \mathsf{scatter}\ i\ j\ k : \mathbf{scatter}\ i\ D[n * size]}
$$

$$
\frac{\Gamma \vdash i : \mathbf{rank} \quad \Gamma \vdash_{\mathsf{s}} j : D[n] \quad \Gamma \vdash_{\mathsf{s}} k : D[n * size]\ \mathbf{ref}}{\Gamma \vdash \mathsf{gather}\ i\ j\ k : \mathbf{gather}\ i\ D[n * size]}
$$

Fig. 16. Formation rules for scattering and gathering arrays.

Arbitrary topologies can be encoded, e.g., using an adjacency matrix. For example, a partially connected mesh topology can be encoded with two arrays, one for the sources, and another for the targets. We leave to the reader adapting the above protocols to this situation.

The formation rules for arrays and array operations are in Figure 16. Index terms are extended to include array constants, $[i_1, \ldots, i_n]$, array length, $\mathsf{len}(i)$, array access, $i_1[i_2]$, array creation, $\mathsf{mkarray}\ i\ j$, and array update, $i[j] := k$. To datatypes, we add $D[]$ to describe arrays of a given datatype D. The shorthand notation $D[i]$ stands for the datatype $\{a : D[] \mid \mathsf{len}(a) = i\}$. As before, we distinguish index terms that may be present in types ($\Gamma \vdash i : D$) from those that may manipulate the store ($\Gamma \vdash_{\mathsf{s}} i : D$). Operators that update the store (mkarray and array update) are present in the latter only. Other operators are present in both relations.

The formation rules for the new index terms and datatypes should be straightforward; notice that the rule for array access formation performs array-bound checking. The type formation rules for **scatter** and **gather** require the root process, i, to be a valid rank and the values to scattered/-gathered, D, to be arrays of a length that divides the number of processes, *size*. This restriction ensures that each process receives/sends sub-arrays of equal length, so that they may share the same code.

On what concerns expressions, $\mathsf{scatter}\ i\ j\ k$ denotes a collective operator where i is the root process (the process holding the array), j is the array to be scattered (held by the root), and k is the

reference, local to each process, that will receive a part of the array. The expression formation rule for scatter, ensures that the initial array evenly divides the number of processes (*size*), so that each gets an array of equal length, as requested by type **scatter**. The type for expression gather *i j k* sees the roles of index terms *j* and *k* reversed: the former denotes the local arrays proposed by each process, and the latter the reference, at the root process, that will collect the result. In both expressions, scatter *i j k* and gather *i j k*, the root process *i* goes into the type, so we use pure index term formation ($\Gamma \vdash i : \mathbf{rank}$); the two other indices, *i* and *j*, are arbitrary index terms that may manipulate the store and hence we use the store-related relation ($\Gamma \vdash_{\mathbf{s}} j : D[n]\ \mathbf{ref}$ and $\Gamma \vdash_{\mathbf{s}} k : D[n * size]$ in the case of **scatter**).

Notice that array subtyping is invariant on the type of the elements in the array. This is justified by the fact that array elements are both read and updated as usual in imperative languages [59] (cf. rule S-Ref in Figure 5). We can easily check that agreement (Lemma 4.3) holds for the extensions and that subtyping is still a preorder (Lemma 3.1). Theories of arrays are presently built into many SMT solvers.

6.2 More Collective Operations

This section presents a few extra useful collective operations. Operator allreduce behaves very much like reduce , except that the value obtained from the reduction process is disseminated to all processes (including the root, as usual). The allgather collective operator is similar to gather, only that all processes get the whole array (hence, we allow protocol dependency on this value). We could alternatively define allreduce and allgather as operators derived from reduce or gather (respectively) followed by a broadcast. MPI provides them as primitives to allow performance optimisations by implementors.

As an example let us consider a program that computes the projection of a vector $\vec{y}$ onto a vector $\vec{x}$, notation $\mathrm{proj}_{\vec{x}}\ \vec{y}$. The formula uses the dot product introduced in the previous section as follows:

$$\mathrm{proj}_{\vec{x}}\ \vec{y} = \frac{\vec{x} \cdot \vec{y}}{\vec{x} \cdot \vec{x}} \vec{x}$$

A simple algorithm to compute the $\mathrm{proj}_{\vec{y}}\ \vec{x}$ starts by scattering both vectors as in the previous section, but now processes call the allreduce operation (twice) so that $\vec{x} \cdot \vec{y}$ and $\vec{x} \cdot \vec{x}$ become known to all processes. After that, each process computes its part of the projection vector and, eventually, the root process gathers the resulting vector. The program is as follows:

```
1  let n:{z:nat|z%size==0} = broadcast 0 read_problem_size()
2      x:int[n/size] ref = mkref (mkarray (n/size) 0)
3      y:int[n/size] ref = mkref (mkarray (n/size) 0) in
4  scatter 0 !x read_array(n);
5  scatter 0 !y read_array(n);
6  let x_dot_y:int = allreduce serial_dot(x, y)
7      x_dot_x:int = allreduce serial_dot(x, x)
8      proj_y_x:int[n] ref = mkref (mkarray n 0) in
9  gather 0 project(x_dot_y, x_dot_x, x, y) proj_y_x
```

The first five lines coincide with the first lines of example in the previous section. The remaining code instructs processes to compute the dot product of x by y and of x by itself, both locally and using function serial_dot , and then to engage in two allreduce operations where each process contributes with its local portion of the dot products, the sum of which is then provided to each process (lines 6–7). Because all processes now know both $\vec{x} \cdot \vec{y}$ and $\vec{x} \cdot \vec{x}$, they can proceed with

$$\text{Types} \quad T, U \;::=\; \ldots \mid \textbf{allreduce}\ x\colon D.T \mid \textbf{allgather}\ x\colon D.T$$

$$\text{Expressions} \quad e, f \;::=\; \ldots \mid \text{let } x\colon D = \text{allreduce } i \text{ in } e \mid \text{let } x\colon D = \text{allgather } i \text{ in } e$$

Type formation $\boxed{\Gamma \vdash T\ \textbf{type}}$

$$\frac{\Gamma, x\colon D \vdash T\ \textbf{type}}{\Gamma \vdash \textbf{allreduce}\ x\colon D.T\ \textbf{type}} \qquad \frac{\Gamma \vdash D <\colon \{x\colon E[n] \mid n\%size = 0\}}{\Gamma \vdash \textbf{allgather}\ x\colon D.T\ \textbf{type}}$$

Expression formation $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash_{\mathsf{s}} i : \textbf{int} \quad \Gamma, x\colon \textbf{int} \vdash e : T \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash \text{let } x\colon \textbf{int} = \text{allreduce } i \text{ in } e : \textbf{allreduce}\ x\colon \textbf{int}.T}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} i : E[n] \quad \Gamma \vdash D <\colon E[n * size] \quad \Gamma, x\colon D \vdash e : T \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash \text{let } x\colon D = \text{allgather } i \text{ in } e : \textbf{allgather}\ x\colon D.T}$$

Fig. 17. Formation rules for further collective operations.

the computation of the local projection of y over x, using function project. In the end, process rank 0 gathers the contribution of all processes and stores the result in reference proj_y_x (line 9).

The type for this code is as follows:

```
1 broadcast 0 n:{z:nat|z%size==0}.
2 scatter 0 int[n];
3 scatter 0 int[n];
4 allreduce x_dot_y:int.
5 allreduce x_dot_x:int.
6 gather 0 int[n]
```

We can easily check that the code type checks against the type. The first three lines coincide with the type of the previous section and we already explained how they match against the first five lines of the program. As for the remaining code, the allreduce operations in lines 6 and 7 yield the **allreduce** types in lines 4 and 5. In both cases, variables x_dot_y and x_dot_x of type **int** are introduced. Then, line 9 in the source code correspond to line 6 in the type. The project function provides an **int**[n/ size] array that is gathered into an **int**[n] array as prescribed by the type. The restrictions on array lengths imposed by the typing rule for reduce are met due to the return type of the project function (which we assume being **int**[n/ size]) and by the type of array proj_y_x .

The extensions introduced in this section are summarised in Figure 17. It is instructive to compare the type formation rule of **allreduce** against that of **reduce** (Figure 6): the former allows the continuation type T to use the value obtained by reduction (via variable x), whereas the latter does not even mention the continuation type. The difference is that, in **allreduce**, all processes share a common value, whereas in **reduce** this value is held by the root process alone and therefore cannot constitute a dependency for the global protocol.

More collective operations in the spirit of MPI [17] can easily be added. Examples include a simple barrier where processes synchronise and progress only when all have arrived at the barrier, sendreceive where two processes exchange a pair of bidirectional messages, and alltoall where each process proposes a value to each other process and then collects values from all processes.

6.3 Pairs and allreducemaxloc

This section introduces pairs as a new datatype as well as a new collective communication primitive. Operator allreducemaxloc behaves very much like allreduce (Figure 17), only that each

process contributes a value-rank pair and the value obtained from the reduction process, a pair with the max value and its rank, is disseminated to all processes (including the root, as usual).

As an example we investigate the parallel implementation of the Gaussian elimination algorithm based on a data distribution of a matrix A and a sequence of matrices A^k, $k = 1, \ldots, n-1$, organised in a row-cyclic distribution and a column-oriented pivoting [61]. If n is the dimension of the matrix, each process q owns rows $q, q + \mathit{size}, q + 2 \cdot \mathit{size}, \ldots$, that is, it owns all rows i with $i\%\mathit{size} = q$ for $0 \leq i < n$. The algorithm works as follows: for the forward elimination phase each process takes its elements of column k ($0 \leq k < n - 1$) and determines the local pivot by computing the element with the largest absolute value. The global pivot is the largest absolute value of the local pivots. In case the pivot is owned by a process other than the one holding row, the line is exchanged between these two processes. In order for elimination to happen locally, the pivot is broadcast to all processes. For the backward substitution, process $n - 1$ computes locally and informs all other processes of the value computed; then, process $n - 2$ does its part, and so on, until process 0 completes the computation.

A possible implementation of the Gauss elimination algorithm is as follows:

```
1  // Problem size and data for Ax = b
2  let n:nat = broadcast 0 read_problem_size()
3      a:int[n][n] ref = read_A_row_cyclic_distribution(n, size)
4      b:int[n] ref = read_b_row_cyclic_distribution(n, size)
5      x:int[n] ref = mkref (mkarray n 0) in
6  // Forward elimination
7  for k < n-1 do
8    let (local_pivot,line):int * {x:nat|x<n} = pivot(a, n, n-2-k)
9        (pivot,node):int * rank = allreducemaxloc(local_pivot,myrank)
10       buf:int[k+2] ref = mkref (mkarray (k+2) 0) in
11   ifc (n-2-k)%size != node then // rows pivot and k are on different
         processes
12     if (n-2-k)%size == myrank then
13       copy_row(a, b, n, n-2-k, buf);
14       send node !buf
15     else
16       receive (n-2-k)%size buf
17   else // rows pivot and k are on the same process
18     if (n-2-k)%size == myrank then
19       copy_or_exchange_row(a, b, n, n-2-k, pivot, buf);
20   let pivot_row:int[k+2] = broadcast node !buf in
21   gauss_row_elimination(a, b, n, pivot_row, n-2-k);
22 // Backward substitution
23 for k < n do
24   let local_x:int = broadcast k%size solve_k(a, b, x, n, k) in
25   x[k] := local_x
```

Since our loops unroll downwards, we use expression n-2-k to force column indices to move upwards. The program starts by informing all processes of the problem size n and then allocates and initialises the arrays to hold the coefficient matrix A, vector b, and the unknowns vector x. The program focus on the communication between processes during forward elimination

and backward substitution, so we assume that row cyclic distribution has taken place and is stored at every process. Functions read_A_row_cyclic_distribution and read_b_row_cyclic_distribution load the data appropriately to arrays a and b, respectively (lines 2–5).

The forward elimination operates on a column at a time and is anchored around the selection of a pivot. For that, each process computes the (local) pivot for the lines it holds (function pivot) and then engages in a reduction operation (allreducemaxloc) to determine which process owns the (largest local) pivot (lines 8–10). This operation communicates to all processes a pair containing the pivot and the rank of the process holding it.

To eliminate row $n-2-k$, two situations may happen: either the row to eliminate and the pivot row are on the same process, in which case the elimination can proceed without any communication (lines 18–19); or rows are on different processes. In this case, the process that owns the pivot row must send the significant part of the row (the $k+2$ elements that are not 0) to the process that owns row $n-2-k$ where the elimination is going to take place (lines 12–16). The collective conditional in line 11 ensures that all processes decide equally. To conclude the elimination, the significant portion of the pivot row is distributed among all processes so they can apply the elimination procedure to the rows they own (lines 20–21).

The backward substitution phase, starting at $n-1$, computes the unknowns. The program relies on function solve_k to compute the kth unknown. The process that owns the unknown broadcast it to all processes so they update its x vector.

A possible type for the Gauss elimination algorithm is as follows:

```
1 broadcast 0 n:nat.
2 forall k < n-1.
3   allreducemaxloc (_, node):int*rank.
4   ifc (n-2-k)%size != node then
5     message (n-2-k)%size node;
6   broadcast node int[k+2];
7 forall k < n.
8   broadcast k%size int
```

where we have omitted the variables (_) as well as the continuations (**skip**) of the two **broadcast** operators. We can easily see that the expression above conforms to the type. The broadcast expressions in line 1 corresponds to the **broadcast** in line 1 of the type; they both introduce variable n of datatype **nat**. The allreducemaxloc, the for and the ifc in the code correspond to their counterparts in the type. The send to node and the receive from (n-2-k) the code (lines 14 and 16) correspond the \lstinline message (n-2-k) in the code. The rest of the expression conforms to the rest of the type. For the **broadcast** in line 6, notice that buf is an integer array of size k+2. For that in line 8, we assume that function solve_k returns an **int ref**.

Figure 18 describes the extensions introduced in this section. The pair constructor, (i, j), and its two deconstructors, fst(i) and snd(i), are standard, and so is the datatype $D \times E$ for pairs. The formation and subtyping rules are also standard. The figure introduces a new type constructor, **allreducemaxloc**, similar to **allreduce** (Figure 17), except that each process provides a pair with the value and its rank (rather than just the value), and a pair is collected, composed of the maximum (rather than the sum) element and the rank of the process that proposed the element. In the examples, and for the sake of readability, we use pattern matching in the **allreducemaxloc** expression to deconstruct pairs, obviating the use of fst and snd deconstructors in the subsequent code (and similarly for the type).

$$
\begin{array}{rrcl}
\text{Index terms} & i, j & ::= & \ldots \mid (i, i) \mid \mathsf{fst}(i) \mid \mathsf{snd}(i) \\
\text{Datatypes} & D, E & ::= & \ldots \mid D \times D \\
\text{Types} & T, U & ::= & \ldots \mid \textbf{allreducemaxloc}\ x\colon D.T \\
\text{Expressions} & e, f & ::= & \ldots \mid \mathsf{let}\ x\colon D = \mathsf{allreducemaxloc}\ i\ \mathsf{in}\ e
\end{array}
$$

Index term formation $\boxed{\Gamma \vdash i : D}$ $\boxed{\Gamma \vdash_{\mathsf{s}} i : D}$

$$
\frac{\Gamma \vdash i : D \quad \Gamma \vdash j : E}{\Gamma \vdash (i, j) : D \times E}
\qquad
\frac{\Gamma \vdash i : D \times E}{\Gamma \vdash \mathsf{fst}(i) : D}
\qquad
\frac{\Gamma \vdash i : D \times E}{\Gamma \vdash \mathsf{snd}(i) : E}
$$

$$
\frac{\Gamma \vdash_{\mathsf{s}} i : D \quad \Gamma \vdash_{\mathsf{s}} j : E}{\Gamma \vdash_{\mathsf{s}} (i, j) : D \times E}
\qquad
\frac{\Gamma \vdash_{\mathsf{s}} i : D \times E}{\Gamma \vdash_{\mathsf{s}} \mathsf{fst}(i) : D}
\qquad
\frac{\Gamma \vdash_{\mathsf{s}} i : D \times E}{\Gamma \vdash_{\mathsf{s}} \mathsf{snd}(i) : E}
$$

Datatype formation $\boxed{\Gamma \vdash D\ \textbf{dtype}}$

$$
\frac{\Gamma \vdash D\ \textbf{dtype} \quad \Gamma \vdash E\ \textbf{dtype}}{\Gamma \vdash D \times E\ \textbf{dtype}}
$$

Datatype subtyping $\boxed{\Gamma \vdash D <: D}$

$$
\frac{\Gamma \vdash D_1 <: D_2 \quad \Gamma \vdash E_1 <: E_2}{\Gamma \vdash D_1 \times E_1 <: D_2 \times E_2}
$$

Type formation $\boxed{\Gamma \vdash T\ \textbf{type}}$

$$
\frac{\Gamma, x\colon D \vdash T\ \textbf{type} \quad \Gamma \vdash D <: \textbf{int} \times \textbf{rank}}{\Gamma \vdash \textbf{allreducemaxloc}\ x\colon D.T\ \textbf{type}}
$$

Expression formation $\boxed{\Gamma \vdash e : T}$

$$
\frac{\Gamma \vdash_{\mathsf{s}} i : D \quad \Gamma, x\colon D \vdash e : T \quad \Gamma \vdash D <: \textbf{int} \times \textbf{rank} \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash \mathsf{let}\ x\colon D = \mathsf{allreducemaxloc}\ i\ \mathsf{in}\ e : \textbf{allreducemaxloc}\ x\colon D.T}
$$

Fig. 18. Formation rules for pairs and allreducemaxloc.

In order to simplify the type theory we decided that messages carry integer values only (Figure 9). The send and receive expressions in the example exchange arrays (of datatype **int** [k+2]). In order to type check the example we need to generalise rules E-Send and E-recv so as to allow exchanging arrays as well.

We can easily check that agreement (Lemma 4.3) holds for the extensions and that subtyping is still a preorder (Lemma 3.1). A theory for pairs is not difficult to develop, if not present as primitive in SMT solvers.

7 RELATED WORK

This section compares ParTypes to related approaches found in theliterature.

Session Type Theories. Among all theoretical works on session types, the closest to ours is probably that of Deniélou et al. [11], introducing dependent types and a form of primitive recursion into session types. ParTypes aims at verifying real parallel programming languages, so that it provides for various communication primitives (in contrast to message passing only in [11]) and incorporates dependent collective choices. On the other hand, we do not allow session delegation which is not needed in parallel programming languages such as MPI. At the term level, we work with an imperative language, as opposed to a variant of the the π-calculus used by Deniélou et al. [11]. Kouzapas et al. [41] introduce a notion of broadcast in the setting of session types. A new

operational semantics system provides for the description of 1-to-n and n-to-1 message passing, where n is not fixed *a priori*, meaning that a non-deterministic number of processes may join the message-passing operation, the others being left waiting. Types, however, do not distinguish point-to-point from broadcast operations. We work on a deterministic setting and provide a much richer choice of type operators.

Toninho and Yoshida [74] propose a *value-dependent* multiparty session calculus and study several conditions for ensuring the well-formedness of global types. The base syntax is a variant of the (multiparty session) π-calculus; parameterisation over the number of participants (as in ParTypes and others [8, 53, 54]) is not studied. See the following paragraph for comparison with multiparty session-based approaches.

Toninho and Yoshida [75] propose a dependent typed calculus that combines functions and linear logic-based session-typed processes, allowing full value dependency (i.e., higher-order functional dependency). The base syntax is the linear logic based π-calculus and its expressiveness is limited to a strong normalising (linear-logic based) fragment of the π-calculus.

Scribble. Based on the theory of multiparty session types by Honda et al. [38, 39], Scribble [34, 36, 66, 82] is a language to describe protocols for message-passing programs. Protocols written in Scribble include explicit senders and receivers, thus ensuring that all senders have a matching receiver and vice versa. Global protocols are projected into each participant, yielding one local protocol for each participant present in the global protocol. Developers can then implement programs based on the local protocols and using standard message-passing libraries, as in Multiparty Session C [56].

Pabble [54] is a parametric extension of Scribble, which adds indices to participants and represents Scribble protocols in a compact and concise notation for parallel programming. Pabble protocols can represent interaction patterns of MPI programs where the number of participants in a protocol is decided at runtime. Pabble was applied to generate communication safe-by-construction MPI programs [53, 55], leveraging the close affinity between Pabble protocols and MPI programs. These works show how protocol languages can be used for verifying or constructing MPI programs. Recently an indexed dependent-type theory for Scribble which allows more flexible projection was proposed by Castro-Perez et al. [8] and applied to generating APIs for Go programming language.

In ParTypes, we depart from multiparty session types along two distinct dimensions: (a) our protocol language is specifically built for MPI primitives, and (b) we do *not* explicitly project a protocol nor generate the MPI code but else check the conformance of code against a global protocol. In contrast to ParTypes, works on parameterised session types [8, 53, 54] cannot deal with:

- Protocols where a given communication (say the source or the target) depends on the contents of previously exchanged data;
- Protocols whose behaviour does not depend directly on message passing, but else on a data-dependent common agreement among all processes (what we call *collective operations*); and
- Most of the collective operations (broadcast, gather, scatter, reduce) primitives, as well as general and array passing.

Recent work by Zhou et al. [84] studies *refinement types* in multiparty session types and proposes a toolchain for F$\star$ [73]. This work does not treat parameterisation of participants like ours and [8], [53], and [54], but handle a general value dependency which can be updated when unrolling the recursions.

Choreographic and Multitier Programming. Choreographic programming [6, 7] is another formalism rooted in the theory of the π-calculus, which can express multiparty communication protocols by providing primitives for programming global interactions. One of the early examples

is the W3C's Web Services Choreography Description Language [78], targeting mainly web services and distributed systems. Multitier programming [79], on the other hand, are rooted in the theory of the λ-calculus and enable programming computations that spans across several tiers of a distributed system, by supporting primitives for changing the location of program execution. The work by Giallorenzo et al. [24] explains similarities between these two paradigms. The main difference between our work and choreographic programming can be seen from the following perspective: ParTypes start with code for each participant and looks for a type that makes each code typable, whereas choreographic programming starts with a single code that is then projected into the different participants.

Dependent Type Systems. Following Martin-Löf's work on constructive type theory [48], a number of programming languages have made use of dependent type systems. Rather than taking advantage of the power of full dependent type systems (that brings undecidability to type checking), Freeman and Pfenning [21] and Xi and Pfenning [81] introduce a restricted form of dependent types, where types may refer to values of a restricted domain, as opposed to values of the term language. The type checking problem is then reduced to constraint satisfiability, for which different tools are nowadays available. ParTypes follows this approach. Xanadu [80] incorporates these ideas in a imperative C-like language. F$\star$ [73], Omega [67], Liquid Types [63], and Liquid Haskell [76] are further examples of pure functional languages that resort to theorem provers. All these languages share refinement types with ParTypes, but are functional, so that their type systems cannot abstract program's communication patterns.

The ParTypes *Family.* This work constitutes the result of several attempts at verifying C+MPI programs using a type-based approach. Starting with Honda et al. [35], key session type abstractions to capture the general traits of MPI programs were identified, for instance: rank-based communication, collective operations, typical communication patterns (e.g., ring, mesh), and the possible choice/coexistence between blocking and nonblocking operations. Subsequent work proposed a preliminary evaluation of the approach and experiments [47], where we did not make use of a protocol language, verification did not scale and also required an *a priori* defined number of processes. We also considered the type-based verification of WhyML parallel programs [64] and the synthesis of correct-by-construction C+MPI programs from protocol specifications [43].

López et al. [46] first presented ParTypes in its maturity, introducing a type-based methodology for imperative message-passing parallel programs, the implementation of a protocol verification toolchain, and an experimental evaluation of type-based verification against model checking and runtime verification.

ParTypes have influenced different research works. Martins et al. [49] extend ParTypes with type inference: from the source code of one individual process, it attempts to generate its type. Then it gradually merges the type of each individual process into a single type. In case the type can be generated, then the program is deadlock-free and assured to behave as prescribed by the type. The verification method by Martins et al. [49] depends on the number of processes, which limits its extensibility. An additional difference is the omission of parametric types. Finally, ParTypes influenced further extensions of session type theories in other application fields. For example, López et al. [45] extend the primitives for collective communication taking into consideration the availability of nodes in a network, which served to model coordination protocols used in electrical distribution networks [44].

Static Analysis for MPI-like Programs. Fu et al. [22] present MPISE, a symbolic checker for MPI programs written in C. The analysis extracts a subset of the MPI interfaces and checks whether there is a path that deadlocks. The set of MPI primitives include synchronous, point-to-point

communication, collectives, as well as wildcard receives, but no scatter-gather operations. The symbolic execution takes the state of the program and uses a scheduler to analyse the result of possible continuations. If one of the continuations reports a deadlock, the process stops. MPISE offers deadlock-freedom guarantees, but no protocol conformance for endpoints have been studied. Droste et al. [12] present an extension of the Clang static analyzer for MPI programs. It performs a combination of abstract-syntax tree analysis and symbolic execution to check for properties including type mismatch, buffer referencing, protocol well-formedness and deadlock freedom. In comparison to our work, MPI-checker does not consider errors coming from the use of collective communications. Botbol et al. [2] apply abstract interpretation techniques to MPI-communication. Their technique represents sets of program states as lattice automata, and the program semantics is encoded as symbolic transducers. Safety properties are encoded as lattice automaton, and the verification of properties computes whether the intersection of the languages generated by the program and a safety property is empty. The technique was tested in a subset of the MPI language considered by ParTypes, including point-to-point message passing, broadcast. and reduce. In comparison, the method proposed by Botbol et al. [2] allows the verification of properties regarding the state of the MPI program, and the method can be used to check whether reachable program states are not matched by any transition, thus detecting deadlocks.

Model Checking. Verification techniques based on model checking require tracking the state of each of the processes involved, which easily becomes a non-scalable solution. Some symbolic model checkers, such as TASS [71] and CIVL [70] employ symbolic model checking techniques in order to verify safety properties such as deadlock detection, buffer overflows and memory leaks. TASS checks for the functional equivalence between MPI programs and sequential counterparts [71]. Böhm et al. [1] check for deadlocks by implementing partial-order reduction techniques to construct the state space directly. Yu [83] presents a symbolic model checking technique for detecting deadlocks in MPI programs with non-deterministic sends and non-blocking receives: it first collects all the paths where processes block or terminate, and for each of the non-deterministic communication actions, it generates symbolic states representing different message matchings. The path-level models are used to generate a CSP [32] model representing possible interleavings, which is then used for model checking against a global reachability property representing deadlock freedom.

Tools for the Verification of MPI Programs. The MPI standard has more than 20 years of existence, and it is not surprising that it has generated a large number of tools for validating and verifying protocols. Gopalakrishnan et al. [25] provide an initial perspective on the type of formal techniques used ensuring the reliability of message-passing programs with the MPI standard.

Tools such as the **in-situ partial order (ISP)** [58], the **distributed analyser for MPI (DAMPI)** [77] and Marmot [3, 30, 31, 42] are runtime verifiers that aim at detecting deadlocks, hence are dependent on the quality of the tests. These tools typically require an overhead at runtime. The overhead can be manifested in additional wrappers that intercept any MPI call issued by the application (e.g., Marmot [42]), or by additional threads per each MPI process (e.g., Umpire [3]). Even with architectural changes that decrease the runtime overhead (cf. MUST [31]), the performance of runtime verifiers depends on the number of processes and the number of loop iterations that monitored programs execute. In contrast, our type checker running time does not depend on the number of loop iterations, yet reliance on an SMT solver renders type checking exponential on the number of variables in verification conditions, as opposed to the number of processes.

The Hermes tool [40] implements a hybrid verification technique to limit the growth in the state space when verifying multipath MPI programs, for instance, those using wildcard operations. The tool combines explicit-state model checking and symbolic analysis: the explicit part encodes the

program in the form of rules to constraint rules, that are fed to an SMT solver. In case of unsatisfiability, the symbolic analysis modifies the scheduler to infer the existence of another executable control flow path. It is difficult to compare the approach taken by Hermes with respect to ours, since most of the work has been placed with the assumption that data is received using wildcard receives, and no collective communications are supported.

The MOPPER tool [15, 16] is a verifier that detects deadlocks by analyzing execution traces of MPI programs. It executes an MPI program, collects the trace, builds a formula from the trace that determines whether there exists a valid MPI run that respects the matches-before order and yields a deadlock, and checks for satisfiability. This technique differs from our solution, that performs verification without requiring running the program.

Errors and Exceptions. Large-scale parallel applications can suffer from different kinds of errors that cannot be overlooked. Exceptions handling at the protocol level has been studied in the context of session and behavioural types [4, 5, 20, 51]. These works allows for communicating peers to escape from one point of a protocol to another in a coordinated fashion, so that processes are guaranteed to follow a protocol even in the presence of errors. On the MPI side, little work has been done on error/exception verification, apart from DeFreez et al. [10] that study error code propagation on MPI library implementations, notably MPICH. The current version of ParTypes does not address errors or exceptions; Section 8 discusses ideas for future work.

8 CONCLUSION AND DISCUSSION

This article presents ParTypes, a theory of types for parallel programming whereby programs that can be given a type are guaranteed to be strongly normalising, hence exempt from deadlocks. Furthermore, in ParTypes, checking program formation against a type is a decidable property, and we present an algorithm for the effect. The ParTypes language is quite rich, allowing to describe a large number of algorithms usually found in textbooks on parallel programming. Type-based approaches have clear advantages against competing solutions for the verification of the sort of functional properties that can be captured by types. With respect to testing, not only they forego the writing (and running) of test suits, but they give a far greater confidence. When compared to model checking, they do not suffer from scalability problems on what concerns, e.g., the problem size or the number of loops iterations in a program. There are however limitations to the current version of ParTypes, which we discuss in turn.

Type Equivalence Is Too Intensional. Type equivalence, as introduced in this article, is at times not flexible enough, requiring source code to be "too aligned" with the type. This restrains developers' options when writing code for a given type. An example features one type with a "fat" loop inside which a conditional avoids sending a message to a fixed process, and another type with two consecutive loops achieving the same effect (see Section 3.3). The types are not equivalent but programs exhibit the same interactive behaviour, namely they both send the same number of messages, in the same order. A bisimulation-based notion of type equivalence could alleviate this problem.

Non-Blocking Message Passing. The sort of point-to-point messages that we consider are synchronous, blocking, and non-buffered: the sending and the receiving processes wait for each other, then exchange a value, and then continue their works. Asynchronous, or non-blocking, or buffered messages allow the superposition of computation and communication: the sender process writes to the buffer and continues; the receiver only blocks if the buffer is empty. Many efficient algorithms use buffered message-passing. Modelling at type level buffered messages poses some subtle difficulties we did not manage to overcome. We present two of these hurdles: keeping track of the message

ordering between pairs of sender-receiver processes and accounting for the effective transmission of non-blocking messages. The former may cause deadlocks, while the latter may originate data races.

For the former, the order of the messages exchanged between the same pairs of processes must be maintained, regardless of the blocking/non-blocking semantics. On the other hand, messages not related to the same sender-receiver pair can be interleaved. Capturing such message intertwining in a satisfactory manner in a type system turns out be quite a challenge.

As for the latter, a sender or receiver process may only use the data in messages after its effective transmission. Otherwise, the process may enter a data race with the underlying middleware responsible for the actual movement of the data between processes. Therefore, every non-blocking send/receive operation needs to be matched by a wait operation, which marks the point in the program after which data may be safely used. Finding an adequate representation, in the type system, for uniquely identifying such matches revealed a challenge, in particular, in the presence of primitive recursion.

Receive from Any. MPI includes a receive-from-any primitive (wildcard receive), which leads directly to *race conditions*, and race conditions may easily lead to deadlocks. Just think of process 0 receiving from any and processes 1 and 2 sending to 0. One of the sending processes will deadlock, a clearly undesirable situation. We analysed some instances of this primitive in textbooks and found many instances where the wildcard can be easily identified. A notable example is the solution to the parallel Gaussian elimination algorithm by Rauber and Rünger [61] (discussed in Section 6.3) where the sending process can be identified, making the wildcard a convenience for not having to explicitly calculate the sender. Genuine races not leading to deadlocks—as in process 0 receives twice from any and processes 1 and 2 send to 0—constitute a challenge we leave for further work.

Process Topologies. Topologies restrict collective communications to subgroups of processes, imposing a logical organisation of processes that mimics the structure of particular problems, for instance, by renumbering ranks so that communicating processes become physically closer to each other. MPI offers n-dimension Cartesian topologies, with primitives to create a new topologies, to obtain the rank of a process in the new topology, and to obtain neighbours to a process by specifying an offset in one of the dimensions. Given the discrete nature of Cartesian topologies, we could think of adding a datatype to describe a topology and the length of each dimension (as a **nat[]**) together with (expression and type) primitives to encode the topology-specific operators. We leave this for future work.

Further Datatypes. A single base datatype is enough to exhibit the versatility of ParTypes. The propositions that refine datatypes are those of the programming language and deal with integer values, references, pairs and arrays. Decidability of type checking depends on the availability (and decidability) of theories for the different datatypes. Realistic programming languages require further base types, including booleans, characters, strings, and floating point numbers. Floating point values are particularly important for scientific computing, a notable application of parallel programming. They can be easily added to store-related indices, together with the relevant operators (Figure 8).

General Recursion. We have deliberately eschewed while loops from our term language. The addition of such a loop construct at the term level would undermine strong normalisation (just think of a process running a while true loop). In a full-fledged programming language, one would probably accept such a tradeoff. General recursion at the type level is a completely different story: type equivalence would become undecidable undermining the whole ParTypes metatheory.

Errors and Exceptions. MPI v4 [18] makes available three run-time error handlers with different semantics (abort all executing processes, abort processes within a communicator, return an error code without aborting). In most of the cases, error handling allows for the application to take appropriate actions based upon the error code. Typically, before terminating, applications flush buffers and save internal state that allows the algorithm to resume later. Ideas put forward on session and behavioural types [4, 5, 20, 51] could be used to enrich ParTypes. We anticipate challenges due to the rudimentary support for error handling made available by the MPI library. ParTypes could explore the new concept of session introduced in MPI v4 that allows for processes to rejoin a session after a failure and resume the computation, thus ensuring that it happens according to the protocol.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the detailed comments and valuable suggestions that greatly improved the quality of the article.

REFERENCES

[1] Stanislav Böhm, Ondřej Meca, and Petr Jančar. 2016. State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in MPI. In *FM 2016: Formal Methods, Lecture Notes in Computer Science (LNCS)*. Springer, 102–118.

[2] Vincent Botbol, Emmanuel Chailloux, and Tristan Le Gall. 2017. Static analysis of communicating processes using symbolic transducers. In *Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science (LNCS)*. Springer, 73–90.

[3] D. Bridges and S. Mostashfi. 2009. Universal monitoring platform for interactive real-time expansive networks (UMPIRE). In *CTS*. IEEE, 571.

[4] Luís Caires and Jorge A. Pérez. 2017. Linearity, control effects, and behavioral types. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer, Berlin, 229–259.

[5] Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured interactional exceptions in session types. In *CONCUR 2008 - Concurrency Theory*. Springer, Berlin, 402–417.

[6] Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.* 34, 2 (2012), 8:1–8:78. https://doi.org/10.1145/2220365.2220367

[7] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *POPL*. ACM, 263–274. https://doi.org/10.1145/2429069.2429101

[8] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in Go: Statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 29:1–29:30. https://doi.org/10.1145/3290342

[9] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. 2009. VCC: A practical system for verifying concurrent C. In *TPHOLs*, Lecture Notes in Computer Science (LNCS), Vol. 5674. Springer, 23–42.

[10] Daniel DeFreez, Antara Bhowmick, Ignacio Laguna, and Cindy Rubio-González. 2020. Detecting and reproducing error-code propagation bugs in MPI implementations. In *PPoPP (PPoPP'20)*. ACM, New York, 187–201. https://doi.org/10.1145/3332466.3374515

[11] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised multiparty session types. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:6)2012

[12] Alexander Droste, Michael Kuhn, and Thomas Ludwig. 2015. MPI-checker: Static analysis for MPI. In *Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 3:1–3:10. https://doi.org/10.1145/2833157.2833159

[13] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*. ACM, 429–442. https://doi.org/10.1145/2500365.2500582

[14] Francisco Ferreira and Brigitte Pientka. 2014. Bidirectional elaboration of dependently typed programs. In *PPDP*. ACM, 161–174. https://doi.org/10.1145/2643135.2643153

[15] Vojtěch Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2017. Precise predictive analysis for discovering communication deadlocks in MPI programs. *ACM Trans. Program. Lang. Syst.* 39, 4 (2017), 15:1–15:27. https://doi.org/10.1145/3095075

[16] V. Forejt, D. Kroening, G. Narayanswamy, and S. Sharma. 2014. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *FM (LNCS)*, Vol. 8442. Springer, 263–278.

[17] Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard, Version 3.1*. University of Tennessee.
[18] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard, Version 4.0*. University of Tennessee.
[19] I. Foster. 1995. *Designing and Building Parallel Programs*. Addison-Wesley.
[20] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: Session types without tiers. *Proc. ACM Program. Lang.* 3, POPL, Article 28 (2019), 29 pages. https://doi.org/10.1145/3290341
[21] Timothy S. Freeman and Frank Pfenning. 1991. Refinement types for ML. In *PLDI*. ACM, 268–277. https://doi.org/10.1145/113445.113468
[22] X. Fu, Z. Chen, Y. Zhang, C. Huang, W. Dong, and J. Wang. 2015. MPISE: Symbolic execution of MPI programs. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. IEEE, 181–188. https://doi.org/10.1109/HASE.2015.35
[23] Simon J. Gay and Malcolm J. Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informaticæ* 42, 2/3 (2005), 191–225.
[24] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty languages: The choreographic and multitier cases (pearl). In *ECOOP (LIPIcs)*, Vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:27. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22
[25] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Formal analysis of MPI-based parallel programs. *Commun. ACM* 54, 12 (2011), 82–91. https://doi.org/10.1145/2043174.2043194
[26] A. D. Gordon and C. Fournet. 2010. Principles and applications of refinement types. In *International Summer School Logics and Languages for Reliability and Security*. IOS Press, 73–104.
[27] Ananth Grama, Anshul Gupta, George Karpsis, and Vipin Kumar. 2003. *Introduction to Parallel Computing*. Addison Wesley.
[28] W. Gropp, E. Lusk, and A. Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT press.
[29] Per Brinch Hansen. 1991. *The N-Body Pipeline*. Technical Report. Electrical Engineering and Computer Science Technical Reports Paper 120, College of Engineering and Computer Science, Syracuse University.
[30] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. 2012. MPI runtime error detection with MUST: Advances in deadlock detection. In *SC*. IEEE/ACM, 30:1–30:11.
[31] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller. 2009. MUST: A scalable approach to runtime error detection in MPI programs. In *Parallel Tools Workshop*. Springer, 53–66.
[32] C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.
[33] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR, Lecture Notes in Computer Science (LNCS)*, Vol. 715. Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35
[34] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Denielou, and Nobuko Yoshida. 2014. Structuring communication with session types. In *COB 2014, Lecture Notes in Computer Science, (LNCS)*, vol. 8665. Springer, 105–127.
[35] K. Honda, E. R. B. Marques, F. Martins, N. Ng, V. T. Vasconcelos, and N. Yoshida. 2012. Verification of MPI programs using session types. In *Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science (LNCS)*, vol. 7490. Springer, 291–293.
[36] K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida. 2011. Scribbling interactions with a formal foundation. In *ICDCIT, Lecture Notes in Computer Science (LNCS)*, vol. 6536. Springer, 55–75.
[37] K. Honda, V. T. Vasconcelos, and M. Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *ESOP, Lecture Notes in Computer Science (LNCS)*, vol. 1381. Springer, 122–138.
[38] K. Honda, N. Yoshida, and M. Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.
[39] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty asynchronous session types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695
[40] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare. 2018. Dynamic symbolic verification of MPI programs. In *Formal Methods, Lecture Notes in Computer Science (LNCS)*. Springer, 466–484.
[41] Dimitrios Kouzapas, Ramunas Gutkovas, and Simon J. Gay. 2014. Session types for broadcasting. In *PLACES (EPTCS)*, vol. 155. 25–31. https://doi.org/10.4204/EPTCS.155.4
[42] B. Krammer, T. Hilbrich, V. Himmler, B. Czink, K. Dichev, and M. S. Müller. 2008. MPI correctness checking with marmot. In *Parallel Tools Workshop*. Springer, 61–78.
[43] F. Lemos. 2014. *Synthesis of correct-by-construction MPI programs*. Master's Thesis. Department of Informatics, University of Lisbon.

[44] Hugo A. López and Kai Heussen. 2017. Choreographing cyber-physical distributed control systems for the energy sector. In *Proceedings of the Symposium on Applied Computing*. ACM, 437–443. https://doi.org/10.1145/3019612.3019656
[45] Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. 2016. Enforcing availability in failure-aware communicating systems. In *FORTE, Lecture Notes in Computer Science (LNCS)*, vol. 9688. Springer, 195–211. https://doi.org/10.1007/978-3-319-39570-8_13
[46] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *OOPSLA*. ACM, 280–298. https://doi.org/10.1145/2814270.2814302
[47] E. R. B. Marques, F. Martins, V. T. Vasconcelos, N. Ng, and N. Martins. 2013. Towards deductive verification of MPI programs against session types. In *PLACES (EPTCS)*, Vol. 137. 103–113.
[48] P. Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis-Napoli.
[49] Francisco Martins, Vasco Thudichum Vasconcelos, and Hans Hüttel. 2017. Inferring types for parallel programs. In *PLACES (EPTCS)*, Vol. 246. 28–36. https://doi.org/10.4204/EPTCS.246.6
[50] M. Moskal. 2011. Verifying functional correctness of C programs with VCC. In *NASA Formal Methods, Lecture Notes in Computer Science (LNCS)*, vol. 6617. Springer, 56–57.
[51] Dimitris Mostrous and Vasco Thudichum Vasconcelos. 2014. Affine sessions. In *Coordination Models and Languages*. Springer, Berlin, 115–130.
[52] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00962ED2V01Y201910CAC049
[53] Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. 2015. Protocols by default - safe MPI code generation based on session types. In *CC, Lecture Notes in Computer Science (LNCS)*, vol. 9031. Springer, 212–232. https://doi.org/10.1007/978-3-662-46663-6_11
[54] Nicholas Ng and Nobuko Yoshida. 2015. Pabble: Parameterised scribble. *Service Oriented Computing and Applications* 9, 3-4 (2015), 269–284. https://doi.org/10.1007/s11761-014-0172-8
[55] Nicholas Ng and Nobuko Yoshida. 2017. *Behavioural Types: From Theory to Tools*. River Publishers, Chapter Protocol-Driven MPI Program Generation, 329–352.
[56] N. Ng, N. Yoshida, and K. Honda. 2012. Multiparty session C: Safe parallel programming with message optimisation. In *TOOLS Europe*, *Lecture Notes in Computer Science (LNCS)*, vol. 7304. Springer, 202–218.
[57] P. S. Pacheco. 1997. *Parallel Programming with MPI*. Morgan Kaufmann.
[58] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, R. Thakur, and W. Gropp. 2007. Practical model-checking method for verifying correctness of MPI programs. In *PVM/MPI, Lecture Notes in Computer Science, (LNCS)*, vol. 4757. Springer, 344–353.
[59] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.
[60] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100
[61] Thomas Rauber and Gudula Rünger. 2010. *Parallel Programming: For Multicore and Cluster Systems*. Springer.
[62] John C. Reynolds. 1997. *ALGOL-like Languages, Volume 1*. Birkhauser Boston Inc., Chapter The Essence of ALGOL, 67–88. http://dl.acm.org/citation.cfm?id=251167.251168.
[63] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid types. In *PLDI*. ACM, 159–169.
[64] C. Santos, F. Martins, and V. T. Vasconcelos. 2015. Deductive verification of parallel programs using Why3. In *ICE*, Vol. 189. EPCTS, 128–142. https://doi.org/10.4204/EPTCS.189
[65] M. Schulz and B. R. de Supinski. 2006. A flexible and dynamic infrastructure for MPI tool interoperability. In *ICPP*. IEEE, 193–202.
[66] Scribble 2015. Scribble Homepage. http://www.scribble.org/. (2015).
[67] T. Sheard and N. Linger. 2007. Programming in Omega. In *CEFP, Lecture Notes in Computer Science (LNCS)*, vol. 5161. Springer, 158–227.
[68] S. F. Siegel and G. Gopalakrishnan. 2011. Formal analysis of message passing. In *VMCAI, Lecture Notes in Computer Science (LNCS)*, vol. 6538. Springer, 2–18.
[69] S. F. Siegel and L. F. Rossi. 2008. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In *EuroPVM/MPI, Lecture Notes in Computer Science (LNCS)*, vol. 5205. Springer, 274–282.
[70] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The concurrency intermediate verification language. In *SC*. ACM, 61:1–61:12. https://doi.org/10.1145/2807591.2807635
[71] S. F. Siegel and T. K. Zirkel. 2011. FEVS: A functional equivalence verification suite for high performance scientific computing. *Mathematics in Computer Science* 5, 4 (2011), 427–435.
[72] S. F. Siegel and T. K. Zirkel. 2012. Loop invariant symbolic execution for parallel programs. In *VMCAI, Lecture Notes in Computer Science (LNCS)*, vol. 7148. Springer, 412–427.

[73] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *POPL*. ACM, 256–270. https://doi.org/10.1145/2837614.2837655

[74] Bernardo Toninho and Nobuko Yoshida. 2017. Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.* 90 (2017), 61–83. https://doi.org/10.1016/j.jlamp.2016.11.005

[75] Bernardo Toninho and Nobuko Yoshida. 2018. Depending on session-typed processes. In *FOSSACS, Lecture Notes in Computer Science (LNCS)*, vol. 10803. Springer, 128–145. https://doi.org/10.1007/978-3-319-89366-2_7

[76] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws.

[77] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky. 2010. A scalable and distributed dynamic formal verifier for MPI programs. In *SC*. IEEE, 1–10. https://doi.org/10.1109/SC.2010.7

[78] W3C. 2004. W3C WS-CDL. http://www.w3.org/2002/ws/chor/. (2004).

[79] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A survey of multitier programming. *ACM Comput. Surv.* 53, 4 (2020), 81:1–81:35. https://doi.org/10.1145/3397495

[80] H. Xi. 2000. Imperative programming with dependent types. In *LICS*. IEEE, 375–387.

[81] H. Xi and F. Pfenning. 1999. Dependent types in practical programming. In *POPL*. ACM, 214–227.

[82] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The scribble protocol language. In *TGC (LNCS)*, Vol. 8358. Springer, 22–41.

[83] Hengbiao Yu. 2018. Combining symbolic execution and model checking to verify MPI programs. In *ICSE: Companion Proceeedings (ICSE'18)*. ACM, 527–530. https://doi.org/10.1145/3183440.3190336

[84] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 148:1–148:30. https://doi.org/10.1145/3428216

Received 20 May 2020; revised 4 April 2022; accepted 5 July 2022