Type-Based Verification of Message-Passing Parallel Programs

Vasco Thudichum Vasconcelos, Francisco Martins,
Eduardo R. B. Marques, Hugo A. López, César Santos,
and Nobuko Yoshida

DI–FCUL–TR–2014–4

DOI:10455/6902

(http://hdl.handle.net/10455/6902)

November 2014

Published at Docs.DI (http://docs.di.fc.ul.pt/), the repository of the
Department of Informatics of the University of Lisbon, Faculty of Sciences.

Type-Based Verification of Message-Passing Parallel Programs

Vasco Thudichum Vasconcelos[1], Francisco Martins[1], Eduardo R. B. Marques[1], Hugo A. López[1], César Santos[1], and Nobuko Yoshida[2]

[1] LaSIGE, University of Lisbon
[2] Imperial College London

Abstract. We present a type-based approach to the verification of the communication structure of parallel programs. We model parallel imperative programs where a fixed number of processes, each equipped with its local memory, communicates via a rich diversity of primitives, including point-to-point messages, broadcast, reduce, and array scatter and gather. The paper proposes a decidable dependent type system incorporating abstractions for the various communication operators, a form of primitive recursion, and collective choice. Term types may refer to values in the programming language, including integer, floating point and arrays. The paper further introduces a core programming language for imperative, message-passing, parallel programming, and shows that the language enjoys progress.

1 Introduction

Parallel programming finds wide demand from the high-performance and scientific computing community. One of the major challenges in developing parallel applications is ensuring that programs do not engage in undesired races or deadlocks. This constitutes a particularly difficult undertaking, to which nondeterminism and different semantics for message passing significantly contribute. The current practice of detecting problematic situations in parallel programming often resorts either to testing or to program debuggers and verifiers. Testing can become quite expensive since meaningful tests must run directly on the multicore or the cluster where the final application will eventually be deployed. Debuggers and verifiers (usually based on runtime testing or model checking) are only of limited assistance, and usually do not scale beyond a small number of processes [8, 22, 26, 27].

Observing the current practice of parallel programming development, one realises that part of the problem relies on the lack of adequate programming support. Development is usually performed in Fortran or C, languages characterised by a rather low degree of abstraction in general, and null on what concerns the particular operations of parallel programming, which are usually accessed via dedicated APIs such as MPI [4]. This observation constitutes the departing point for our work. We aim at providing the necessary language support

for the safe development of applications composed of a large number of processes that communicate with each other via different forms of message passing primitives.

The first contribution of this paper is the design of a language for protocol description, embodied as a dependent type language equipped with primitive recursion, and able to capture the usual sort of protocols found in the realm of parallel programming. The type language features specific constructors for the various communication primitives, including point-to-point message passing, broadcast, reduce, array gather and scatter. For example, a type $\mathsf{message}\ 1\ 2\ \mathsf{int}$ denotes a message exchange between process rank 1 and process rank 2 of an integer value, and $\mathsf{broadcast}\ 1\ x\colon \mathsf{float\ array}.T$ denotes an operation whereby process rank 1 passes a floating point array to all other processes. The rest of the protocol is captured by type T which may refer to the value passed via variable x. The type language further counts with the dependent product type, and a concrete version of primitive recursion [19], $\forall x \leq n.T$, where zero is the skip type (denoting a computation that does not engage in any communication), and the successor is sequential composition. In addition, a novel collective choice type operator, $p\,?\,T_1 : T_2$, provides for decisions common to all processes, based on a proposition (p) as opposed to explicit communication, a pattern commonly found in parallel programs. Following the practice of DML [29], our types may depend on a domain of index objects, which in our case describe integer, floating point and array values, further refined with the subset datatype $\{x\colon D \mid p\}$. This choice is in line with the usual requirements of parallel programming, making in addition type checking decidable, for a suitable choice of propositions.

Abstracting processes, term types provide a *local* view of computation. Program types, one the other hand, provide a *global* perspective of programs, that is, vectors of processes. Two major challenges addressed by this work are the identification of the conditions under which a vector of term types constitutes a program type in such a way that code that conform to program types does not deadlock, and the ability to move between global and local perspectives of communication via type equivalence.

A second contribution of this work is the formulation of a simple while language for imperative parallel programming, and the proof of its properties. The language includes, in addition to the standard constructs usually present in a while language, primitives for each of the communication constructors living at the type level, a for-loop matching primitive recursion, and a novel collective conditional expression, whereby all processes are guaranteed to either take the left or the right branch, without resorting to communication. Each well formed expression is assigned a type that describes its communication behaviour. In our setting, processes (that is, store-expression pairs) are assigned term types, and programs (vectors of processes) are assigned program types. We develop our theory along the lines of intuitionistic type theory [17], demonstrating the soundness of our proposal via two main results: agreement of program reduction and progress for programs.

We have implemented a verifier that checks protocols for good formation, as an Eclipse plugin [18]. We have also tested our type language and core programming language on a number of representative examples from the field of scientific computing, including one dimensional diffusion, Jacobi iteration, Laplace solver, N-body simulation, π calculation, and parallel vector dot product, all taken from standard textbooks such as [5, 7, 21].

The outline of the paper is as follows. The next section describes the language of dependent types for parallel algorithms. Section 3 presents a core language for imperative parallel programming and its main results. Section 4 discusses related work. Section 5 concludes the paper, pointing directions for further work.

2 The type language

This section introduces the type theory, including the notions of term types (Figure 1), term type equality (Figure 4), and program types (Figure 5).

2.1 Term types

Not all syntax may be judged to be a legal type. For example, one might not want consider "skip;;" a valid piece of syntax. Similarly "broadcast 0 x: float" is not a valid type if considered under a context where x is not deemed as a variable. We abbreviate the judgment "T is a type under context Γ" by $\Gamma \vdash T : \mathbf{type}$. Types depend on *datatypes*, hence we abbreviate a judgement "D is a datatype under context Γ" by $\Gamma \vdash D : \mathbf{dtype}$. In turn, datatypes rely on *propositions*, and so we abbreviate judgements "p is a proposition under context Γ" by $\Gamma \vdash p : \mathbf{prop}$. Now, propositions depend on *index terms*, and so we abbreviate judgements "i is an index term of datatype D under context Γ" by $\Gamma \vdash i : D$. All the above hypothetical judgements depend on datatype *contexts* Γ, (ordered) lists of the form $x_1 : D_1, \ldots, x_n : D_n$. We abbreviate judgements "Γ is a context" by $\Gamma : \mathbf{context}$.

Types rely on three base sets: that of variables (denoted x, y, z), that of integer values (k, l, m, n), and that of floating point values (f). There are two distinguished variables: size and rank; we use them to denote the total number of processes and the number of a given process within a collection of processes. It will always be the case that $1 \leq \mathsf{rank} \leq \mathsf{size}$. The axioms and inference rules for deciding on what counts as a type, a datatype, a proposition, an index term and a context, are in Figure 1. We briefly discuss the various type constructors, starting with index terms.

Index terms describe the values types may depend upon. Our language counts with variables, integer and floating-point constants, arithmetic operations, as well as different array operations, namely array values ($[v_1, \ldots, v_n]$), array access ($i_1[i_2]$) and the array length operation ($\mathsf{len}(i)$). Index terms formation rules further include the standard refinement introduction rule and datatype subsumption [6].

Type formation, $\Gamma \vdash T : \mathbf{type}$

$$\frac{\Gamma \vdash 1 \leq i_1, i_2 \leq \mathsf{size} \wedge i_1 \neq i_2 \ \mathbf{true} \quad \Gamma \vdash D : \mathbf{dtype}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D : \mathbf{type}} \qquad \frac{\Gamma \vdash 1 \leq i \leq \mathsf{size}\ \mathbf{true}}{\Gamma \vdash \mathsf{reduce}\ i : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \leq i \leq \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash D <: \{x\colon D' \,\mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}}{\Gamma \vdash \mathsf{scatter}\ i\ D : \mathbf{type}}$$

$$\frac{\text{see scatter}}{\Gamma \vdash \mathsf{gather}\ i\ D : \mathbf{type}} \qquad \frac{\Gamma \vdash 1 \leq i \leq \mathsf{size}\ \mathbf{true} \quad \Gamma, x\colon D \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{broadcast}\ i\ x\colon D.T : \mathbf{type}}$$

$$\frac{\Gamma, x\colon D \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{val}\, x\colon D.T : \mathbf{type}} \qquad \frac{\Gamma \vdash p : \mathbf{prop} \quad \Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}{\Gamma \vdash p\,?\,T_1 : T_2 : \mathbf{type}}$$

$$\frac{\Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}{\Gamma \vdash T_1; T_2 : \mathbf{type}} \qquad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{skip} : \mathbf{type}} \qquad \frac{\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type}}{\Gamma \vdash \forall x \leq i.T : \mathbf{type}}$$

Datatype formation, $\Gamma \vdash D : \mathbf{dtype}$

$$\frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{int} : \mathbf{dtype}} \qquad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{float} : \mathbf{dtype}} \qquad \frac{\Gamma \vdash D : \mathbf{dtype}}{\Gamma \vdash D\,\mathsf{array} : \mathbf{dtype}} \qquad \frac{\Gamma, x\colon D \vdash p : \mathbf{prop}}{\Gamma \vdash \{x\colon D \mid p\} : \mathbf{dtype}}$$

Proposition formation, $\Gamma \vdash p : \mathbf{prop}$

$$\frac{\Gamma \vdash p_1, p_2 : \mathbf{prop}}{\Gamma \vdash p_1 \wedge p_2 : \mathbf{prop}} \qquad \frac{\Gamma \vdash i_1, i_2 \colon \mathsf{int}}{\Gamma \vdash i_1 \leq i_2 : \mathbf{prop}} \qquad \frac{\Gamma, x\colon \mathsf{int} \vdash p : \mathbf{prop}}{\Gamma \vdash \forall x.p : \mathbf{prop}}$$

Index term formation, $\Gamma \vdash i\colon D$

$$\frac{\Gamma : \mathbf{context} \quad x\colon D \in \Gamma}{\Gamma \vdash x : D} \qquad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash f : \mathsf{float}} \qquad \frac{\Gamma \vdash i : D\,\mathsf{array}}{\Gamma \vdash \mathsf{len}(i) : \mathsf{int}}$$

$$\frac{\Gamma \vdash i_1 : \mathsf{int} \quad \Gamma \vdash i_2 : \mathsf{int}}{\Gamma \vdash i_1 + i_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash i_1 : D \quad \ldots \quad \Gamma \vdash i_n : D}{\Gamma \vdash [i_1, \ldots, i_n] : D\,\mathsf{array}}$$

$$\frac{\Gamma \vdash i_1 : D\,\mathsf{array} \quad \Gamma \vdash 1 \leq i_2 \leq \mathsf{len}(i_1)\ \mathbf{true}}{\Gamma \vdash i_1[i_2] : D}$$

$$\frac{\Gamma \vdash i : D \quad \Gamma \vdash p\{i/x\}\ \mathbf{true}}{\Gamma \vdash i : \{x\colon D \mid p\}} \qquad \frac{\Gamma \vdash i : D_1 \quad \Gamma \vdash D_1 <: D_2}{\Gamma \vdash i : D_2}$$

Context formation, $\Gamma : \mathbf{context}$

$$\frac{}{\varepsilon : \mathbf{context}} \qquad \frac{\Gamma : \mathbf{context} \quad \Gamma \vdash D : \mathbf{dtype} \quad x \notin \Gamma, D}{\Gamma, x\colon D : \mathbf{context}}$$

Fig. 1. Formation rules

Datatypes describe integer (int) and floating-point values (float), arrays of an arbitrary datatype ($D\,\mathsf{array}$), and refinements of the form $\{x\colon D \mid p\}$. Refinement datatypes allow to describe integer values smaller than a given index term i, such as $\{y\colon \mathsf{int} \mid y \leq i\}$, or arrays of a given length n, as in $\{a\colon \mathsf{float\,array} \mid \mathsf{len}(a) = n\}$. Datatypes rely on *propositions*. Figure 1 presents a couple of significant examples of propositions. More can be easily added, including further boolean and relational operators.

Datatype subtyping, $\Gamma \vdash D <: D$

$$\frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{int} <: \mathsf{int}} \quad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{float} <: \mathsf{float}} \quad \frac{\Gamma \vdash D_1 <: D_2}{\Gamma \vdash D_1\ \mathsf{array} <: D_2\ \mathsf{array}}$$

$$\frac{\Gamma \vdash D_1 <: D_2 \quad \Gamma, x\colon D_1 \vdash p\ \mathbf{true}}{\Gamma \vdash D_1 <: \{x\colon D_2 \mid p\}} \quad \frac{\Gamma \vdash D_1 <: D_2 \quad \Gamma, x\colon D_1 \vdash p : \mathbf{prop}}{\Gamma \vdash \{x\colon D_1 \mid p\} <: D_2}$$

Proposition entailment, $\Gamma \vdash p\ \mathbf{true}$

$$\frac{\Gamma \vdash p : \mathbf{prop} \quad \mathrm{formulae}(\Gamma) \vDash p}{\Gamma \vdash p\ \mathbf{true}}$$

Formulae in a context, $\mathrm{formulae}(\Gamma)$

$$\begin{aligned}
\mathrm{formulae}(\varepsilon) &\triangleq \emptyset \\
\mathrm{formulae}(\Gamma, x\colon D) &\triangleq \mathrm{formulae}(\Gamma) \cup \mathrm{forms}(x\colon D) \\
\mathrm{forms}(x\colon \mathsf{int}) &\triangleq \mathrm{forms}(x\colon \mathsf{float}) \triangleq \mathsf{true} \\
\mathrm{forms}(h\colon \{x\colon D \mid p\}) &\triangleq \mathrm{forms}(x\colon D) \wedge p\{h/x\} \\
\mathrm{forms}(h\colon D\ \mathsf{array}) &\triangleq \forall x.1 \leq x \leq \mathsf{len}(h) \rightarrow \mathrm{forms}(h[x]\colon D)
\end{aligned}$$

Fig. 2. Datatype subtyping and proposition entailment

All the formation rules rely on *contexts*, intuitively mapping variables into datatypes. Contexts are also subject to formation rules. Symbol ε denotes the empty context. The second rule for context formation ensures that types appearing in a context are well-formed with respect to the "initial" part of the context, a standard requisite in dependent type systems. Premise $x \notin \Gamma, D$ means that x does not occur in Γ and in D, that is, if $y\colon D'$ is an entry in Γ then y is different from x and x does not occur in D'.

A notion of *subtyping* is defined for index terms: we abbreviate judgements of the form "datatype D_1 is a sub-datatype of D_2 under context Γ" by $\Gamma \vdash D_1 <: D_2$. The rules, presented in Figure 2, are standard, including those for refinement datatypes [6]. The last rule, for example, allows to conclude that $\varepsilon \vdash \{a\colon \mathsf{float\ array} \mid \mathsf{len}(a) = 512\} <: \mathsf{float\ array}$. The notion of *proposition deducibility* is also standard [6]. The formulae relation collects the logical refinements present in a given context. They allow, e.g., to extract formula $\forall 1 \leq y \leq \mathsf{len}(a).a[y] \neq y \wedge \forall 1 \leq y \leq \mathsf{len}(a).1 \leq b[z] \leq \mathsf{size}$ from context $a\colon \{b\colon \{x\colon \mathsf{int} \mid 1 \leq x \leq \mathsf{size}\}\ \mathsf{array} \mid \forall 1 \leq y \leq \mathsf{len}(b).b[y] \neq y\}$. We rely on an auxiliary judgement $\mathrm{formulae}(\Gamma) \vDash p$ stating that a proposition p is deducible from the formulas in a context Γ.

We are now in a position to discuss types and type formation. A type of the form $\mathsf{message}\ i_1\ i_2\ D$ describes a point-to-point communication, from the i_1-ranked process to the i_2-ranked process, of a value of datatype D. Both index terms must denote valid ranks, that is they must lie between 1, the first rank, and size, the number of processes. Furthermore, the sending and receiving pro-

cesses must be different from each other.[3] A type of the form $\mathsf{reduce}\, i$ denotes a collective operation whereby all processes contribute with values that are used to produce a result (say, the maximum). This value is then transmitted to the i-ranked process, usually known as the *root* process. A type of the form $\mathsf{scatter}\, i\, D$ describes a collective operation whereby the i-ranked process (the *root* process) distributes an array among all processes, including itself. The type formation rule requires i to be a valid rank, and the length of the array to divide the number of processes, so that each process receives a sub-array of equal length and the whole array gets distributed. Type $\mathsf{gather}\, i\, D$ denotes the inverse operation, whereby each process proposes an array of identical length, the concatenation of which is delivered to the root process. In both cases, the premise $\Gamma \vdash D <: \{x\colon D' \,\mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}$ ensures that D is an array datatype whose length divides the number of processes.

A type of the form $\mathsf{broadcast}\, i\, x\colon D.T$ denotes a collective communication whereby the root process transmits a value of type D to all processes (including itself). The continuation type T may refer to the value transmitted via variable x. Type $\mathsf{val}\ x\colon D.T$ is the dependent product type. In our case, it denotes a collective operation whereby all processes agree on a common value of datatype D, without resorting to communication. Such a value may be referred to, in the continuation type T, via variable x. Typical applications include program constants and command-line values that protocols may depend upon. A type $p\,?\,T_1 : T_2$ denotes a *collective conditional*, whereby all processes jointly decide on proceeding as T_1 or as T_2, again without resorting to communication. Type $T_1; T_2$ describes a computation that first performs the operations as described by T_1 and then those described by T_2. Type skip describes any computation that does not engage in communication. skip-typed processes are not necessarily halted; they may still perform local operations. Finally, type $\forall x \leq i.T$ is a concrete instance of primitive recursion. A recursive sequence $\forall x \leq i.T$ uniquely determines an indexed family of types $T\{i/x\}; T\{i-1/x\}; \ldots; T\{1/x\}; \mathsf{skip}$.

In addition to the above type constructors, others could be easily added. For example, a type $\mathsf{barrier}$ would work similarly to, say, reduce, except that no value is transmitted. A type $\mathsf{allreduce}\, x\colon \mathsf{float}.T$, denoting an operation whereby all processes contribute with a floating point number, from which a value is computed and delivered to all process, could be introduced as an abbreviation or else as primitive for efficiency reasons. In the former case, such type could abbreviate $\mathsf{reduce}\, 1; \mathsf{broadcast}\, 1\, x\colon \mathsf{float}.T$, where process rank 1 collects a series of floating points values and then distributes the maximum to all processes (including itself). Similarly, a type $\mathsf{allgather}\, x\colon D.T$ could denote an operation whereby all processes contribute with an array, whose concatenation would then be further distributed to all processes: $\mathsf{gather}\, 1\, D; \mathsf{broadcast}\, 1\, x\colon D.T$.

The reader may have noticed that types such as $\mathsf{message}$ or reduce do not introduce value dependencies, whereas others such as $\mathsf{broadcast}$ do. The continuation of a $\mathsf{message}$ type, if existent, is captured by sequential composition,

[3] This requirement is particularly relevant for synchronous, or unbuffered, communication, where a message from, say, rank 2 to rank 2 would constitute a deadlock.

$\mathsf{val}\, n\colon \{x\colon \mathsf{int} \mid x \geq 0 \wedge x\%\mathsf{size} = 0\}.$
$\mathsf{broadcast}\, 1\, m\colon \mathsf{int}.$
$\mathsf{scatter}\, 1\, \{a\colon \mathsf{float\ array} \mid \mathsf{len}(a) * \mathsf{size} = n\};$
$\forall k \leq m.($
  $\forall l \leq \mathsf{size}.($
    $\mathsf{message}\ l\ (l\%\mathsf{size} + 1)\ \mathsf{float};$
    $\mathsf{message}\ l\ ((l - 2 + \mathsf{size})\%\mathsf{size} + 1)\ \mathsf{float});$
  $\mathsf{allreduce}\, x : \mathsf{float.skip});$
$\mathsf{gather}\, 1\, \{b\colon \mathsf{float\ array} \mid \mathsf{len}(b) * \mathsf{size} = n\}$

Fig. 3. The type for the finite differences algorithm

as in $\mathsf{message}\, 1\, 2\, \mathsf{float}; \mathsf{message}\, 2\, 1\, \mathsf{int}$. That of a broadcast is built into the type constructor itself; as in $\mathsf{broadcast}\, 1\, x\colon \mathsf{int}.\mathsf{broadcast}\, 1\, y\colon \{z\colon \mathsf{int} \mid z \geq x\}.\mathsf{skip}$. The fundamental reason for the difference lies in the target of the values transmitted. In the cases of message and reduce values are transmitted to a unique process, namely process i in types $\mathsf{message}\, i'\, i\, D$ and $\mathsf{reduce}\, i$. In that of broadcast all processes receive a *same* value. This value may then be safely substituted in the continuation of the types for *all* processes, thus preserving the natural properties of types for programs. For the same reason, reduce does not introduce a type dependency, whereas allreduce may. Even though all processes receive arrays of equal lengths in a scatter operation, the arrays themselves may be different, hence the type introduces no value dependency.

As a concrete example, consider the *finite differences* algorithm [5]. Given an initial vector X^0, the algorithm calculates successive approximations to the solution $X^1, X^2, \ldots$, until a pre-defined maximum number of iterations has been reached. A type for the algorithm is in Figure 3, where n denotes the length of the input vector, and m the number of iterations to be performed. The length of the array must evenly divide the number of processes, so that the root process may divide the whole array among all processes. The problem size is made known via a val type. The number of iterations is disseminated by process rank 1, via a broadcast operation. The same process then divides the input array among all processes. Each participant is responsible for the calculation of a local part of the solution. Towards this end, in each iteration each process exchanges boundary values with its right and left neighbours. When the pre-defined number of iterations is reached, process rank 1 collects the global error via a reduce operation and the solution to the problem via a gather type.

Topology passing is another example attesting the flexibility of our type language. Datatypes of the form $1\mathrm{D} = \{b\colon \{x\colon \mathsf{int} \mid 1 \leq x \leq \mathsf{size}\}\ \mathsf{array} \mid \forall 1 \leq y \leq \mathsf{len}(b).b[y] \neq y\}$ encode a one-dimensional network topology. In the below type process rank 1 distributes the topology; each process then exchanges a message

with its neighbour.

$$\mathsf{broadcast}\ 1\ t\colon 1\mathrm{D}.\forall x \leq \mathsf{len}(t).\mathsf{message}\ x\ t[x]\ \mathsf{float}$$

A ring topology (a linear array with wraparound link) of length 5 can be encoded in array $[2,3,4,5,1]$. A two dimensional topology can be written as $2\mathrm{D} = \{b\colon 1\mathrm{D} \mid \mathsf{len}(b) = 2\}$. A protocol where each node exchanges a message with its east and south neighbours can be written as

$$\begin{aligned}\mathsf{broadcast}\ 1\ t\colon 1\mathrm{D}.(&\forall x \leq \mathsf{len}(t[1]).\mathsf{message}\ x\ t[1][x]\ \mathsf{float};\\ &\forall y \leq \mathsf{len}(t[2]).\mathsf{message}\ y\ t[2][y]\ \mathsf{float})\end{aligned}$$

In this case a 2-D mesh with wraparound (a 2-D torus) of with 3×3 nodes can be encoded in array $[[2,3,1,5,6,2,8,9,3],[4,5,6,7,8,9,1,2,3]]$. The two examples above assume that each dimension wraps around, so that all processes have exactly one neighbour in each dimension. If that is not the case, we may control neighbourhood at type level by taking advantage of collective choice, as in

$$\mathsf{broadcast}\ 1\ t\colon \mathsf{int\ array}.(\forall x \leq \mathsf{len}(t).(1 \leq t[x] \leq \mathsf{size}\ ?\ \mathsf{message}\ x\ t[x]\ \mathsf{float} : \mathsf{skip}))$$

where messages are exchanged only if the array entry contains a valid process number. In this case, a linear array of length 5 with no wraparound link can be encoded as $[2,3,4,5,0]$.

We now revert to the technical development. Because types may include index terms, they may contain index term variables. We say that types $\forall x \leq i.T$, $\mathsf{val}\ x\colon D.T$ and $\mathsf{broadcast}\ i\ x\colon D.T$ bind the occurrences of variable x in type T. Datatype $\{x\colon D \mid p\}$ binds the occurrences of x in proposition p. The notions of free and bound variables are derived accordingly. We denote by $\mathrm{fv}(T)$ the set of *free variables* in type T, and similarly for datatypes, propositions, and index terms.

Before we proceed any further, we must make sure that the above formation rules are valid. For example, before we can talk about T being a type under context Γ, that is $\Gamma \vdash T : \mathbf{type}$, we must make sure that Γ is indeed a context, that is $\Gamma : \mathbf{context}$. The same applies to the other five judgements our type theory is composed of.

Lemma 1 (agreement for type formation).

$$\frac{\Gamma \vdash T : \mathbf{type}}{\Gamma : \mathbf{context}} \qquad \frac{\Gamma \vdash D : \mathbf{dtype}}{\Gamma : \mathbf{context}} \qquad \frac{\Gamma \vdash p : \mathbf{prop}}{\Gamma : \mathbf{context}}$$

$$\frac{\Gamma \vdash i : D}{\Gamma \vdash D : \mathbf{dtype}} \qquad \frac{\Gamma \vdash D_1 <: D_2}{\Gamma \vdash D_1 : \mathbf{dtype} \quad \Gamma \vdash D_2 : \mathbf{dtype}} \qquad \frac{\Gamma \vdash p\ \mathbf{true}}{\Gamma : \mathbf{context}}$$

Proof. By simultaneous rule induction on the various hypothesis.

The *substitution* operation on types, defined in the standard way (based on the variable bindings introduced above) and denoted by $T\{i/x\}$, replaces all the

Type equality, $\Gamma \vdash T \equiv T$

$$\frac{(\Gamma \vdash T : \mathbf{type})}{\Gamma \vdash T; \mathsf{skip} \equiv T} \quad \frac{(\Gamma \vdash T : \mathbf{type})}{\Gamma \vdash \mathsf{skip}; T \equiv T} \quad \frac{(\Gamma \vdash T_1, T_2, T_3 : \mathbf{type})}{\Gamma \vdash (T_1; T_2); T_3 \equiv T_1; (T_2; T_3)}$$

$$\frac{\Gamma \vdash i < 1 \ \mathbf{true} \quad (\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type})}{\Gamma \vdash \forall x \leq i.T \equiv \mathsf{skip}}$$

$$\frac{\Gamma \vdash i \geq 1 \ \mathbf{true} \quad (\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type})}{\Gamma \vdash \forall x \leq i.T \equiv (T\{i/x\}; \forall x \leq i - 1.T)}$$

$$\frac{\Gamma \vdash i_1, i_2 \neq \mathsf{rank} \ \mathbf{true} \quad (\Gamma \vdash 1 \leq i_1, i_2 \leq \mathsf{size} \wedge i_1 \neq i_2 \ \mathbf{true}) \quad (\Gamma \vdash D : \mathbf{dtype})}{\Gamma \vdash \mathsf{message}\, i_1\, i_2\, D \equiv \mathsf{skip}}$$

Omitting the rules pertaining to congruence and equivalence

Fig. 4. Type equality

occurrences of variable x by index term i in type T, leaving all other variables untouched. The substitution on datatypes $D\{i/x\}$, propositions $p\{i/x\}$, and on index terms $i_1\{i_2/x\}$ are also defined in the conventional, inductive, way. A standard substitution lemma summarises the main property of the operation (see Section B.1). One can easily show that $<:$ is a preorder (again, see Section B.1).

2.2 Term type equality

Type equality plays a central role in dependent type systems. In our case, type equality includes the monoidal rules for semicolon and skip, the expansion of primitive recursion, and for a form of *projection* of message types. The rules in Figure 4 determine what it means for two types to be equal under a given context. There are ten congruence rules (see Section B.2). The congruence rule for message types allows, for example, to show the following equality.

$$\mathsf{size}, \mathsf{rank}\colon \{x\colon \mathsf{int} \mid x = 3\} \vdash \mathsf{message}\ \mathsf{rank}\ (\mathsf{rank}\%\mathsf{size} + 1)\ \mathsf{float} \equiv \mathsf{message}\ 3\ 1\ \mathsf{float}$$

The first pair of rules in Figure 4 provides for the meaning of primitive recursion. The base case represents a terminated computation (from the point of view of communications), denoted by skip. The second premise in the rule ensures that $\forall x \leq i.T$ is indeed a type, playing no other role in the definition of type equality. In these cases, we enclose the premise in parenthesis. The induction step is a sequential composition made of the first iteration ($x = i$) and the rest of the iterations ($x \leq i-1$). The two rules, together with the congruence rules allow to show the following equality, where we abbreviate context $\mathsf{size}\colon \{x\colon \mathsf{int} \mid x = 3\}$ by $\mathsf{size} = 3$. Also, for the sake of brevity we omit the datatype in the message types.

$$\mathsf{size} = 3 \vdash \forall j \leq \mathsf{size}.(\mathsf{message}\, j\, (j\%\mathsf{size} + 1)) \equiv \mathsf{message}\, 3\, 1; \mathsf{message}\, 2\, 3; \mathsf{message}\, 1\, 2$$

The next rule in the figure says that a message type that plays no role for a given process rank is equal to skip. The rule effectively allows to *project* a given

type onto a given rank, a notion introduced in the context of multi-party session types [9], here cleanly captured as type equality. When projecting the above type onto rank 2 we obtain the following type equality.

$$\mathsf{size} = 3, \mathsf{rank} = 2 \vdash \forall j \leq \mathsf{size}.(\mathsf{message}\, j\, (j\%\mathsf{size} + 1)) \equiv \mathsf{skip}; \mathsf{message}\, 2\, 3; \mathsf{message}\, 1\, 2$$

Finally, from the monoidal rules we get:

$$\mathsf{size} = 3, \mathsf{rank} = 2 \vdash \forall j \leq \mathsf{size}.(\mathsf{message}\, j\, (j\%\mathsf{size} + 1)) \equiv \mathsf{message}\, 2\, 3; \mathsf{message}\, 1\, 2$$

2.3 Program types

Program types are vectors of term types. Not all vectors are nevertheless of interest. Program types in particular must not deadlock. Below are a few candidates that, albeit composed of term types, cannot be judged as program types. For the sake of brevity we once more omit the datatype in types.

$$(\mathsf{message}\, 1\, 2), (\mathsf{message}\, 2\, 1)$$
$$(\mathsf{scatter}\, 1), (\mathsf{reduce}\, 1)$$
$$(\mathsf{message}\, 1\, 3; \mathsf{scatter}\, 1), (\mathsf{message}\, 1\, 3; \mathsf{reduce}\, 1), (\mathsf{message}\, 1\, 3; \mathsf{scatter}\, 1)$$
$$(\mathsf{message}\, 3\, 1; \mathsf{message}\, 1\, 2), (\mathsf{message}\, 1\, 2; \mathsf{message}\, 2\, 3), (\mathsf{message}\, 2\, 3; \mathsf{message}\, 3\, 1)$$

The first vector of types is blocked for process rank 1 intends to send a message to rank 2, whereas rank 2 is ready to send a message to rank 1. A variant of the type—$(\mathsf{message}\, 1\, 2), (\mathsf{message}\, 1\, 2)$—constitutes a program type. The second vector also describes a deadlocked computation: process rank 1 is trying to distribute an array, whereas rank 2 is not ready to receive its part. The third case involves a 1–3 message that leads to a deadlocked situation, namely $(\mathsf{scatter}\, 1), (\mathsf{message}\, 1\, 3; \mathsf{reduce}\, 1), (\mathsf{scatter}\, 1)$; notice that the second type is equivalent to $\mathsf{reduce}\, 1$. The fourth case involves a circular waiting situation: the message between 3 and 1 cannot happen before that of 2 and 3 (see type for rank 3); the 2–3 message cannot happen before the 1–2 (rank 2); and finally, the 1–2 message cannot happen before the 3–1 message (rank 1). We judge such vector of types as not constituting program types, based on the intended synchronous (or unbuffered) message passing semantics.

The rules in Figure 5 are meaning determining for assertions of the form $S : \mathbf{ptype}$. They rely on two abbreviations: Γ^n for context $\mathsf{size}\colon \{x\colon \mathsf{int} \mid x = n\}$, and $\Gamma^{n,k}$ for context $\Gamma^n, \mathsf{rank}\colon \{x\colon \mathsf{int} \mid x = k\}$. The indispensable agreement result is Section B.3.

The central intuition of a program type is that it describes a non-deadlocked computation, that is, a computation that is either halted or that may reduce. With this in mind it is easy to understand the rules for the collective operations—reduce, $\mathsf{scatter}$, gather, $\mathsf{broadcast}$, val, if, and skip—they require all types to agree. The premises, in parenthesis, guarantee the validity of the rules: they ensure the good formation of the term types involved. The rule for sequential composition

Program type formation, S : **ptype**

$$\frac{(\Gamma^n \vdash 1 \leq l \leq n \ \mathbf{true})}{\mathsf{reduce}\, l, \ldots, \mathsf{reduce}\, l : \mathbf{ptype}}$$

$$\frac{(\Gamma^n \vdash 1 \leq l \leq n \ \mathbf{true}) \quad \Gamma^n \vdash D <: \{x\colon D' \,\mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}}{\mathsf{scatter}\, l\, D, \ldots, \mathsf{scatter}\, l\, D : \mathbf{ptype}}$$

$$\frac{(\Gamma^n \vdash 1 \leq l \leq n \ \mathbf{true}) \quad \Gamma^n \vdash D <: \{x\colon D' \,\mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}}{\mathsf{gather}\, l\, D, \ldots, \mathsf{gather}\, l\, D : \mathbf{ptype}}$$

$$\frac{(\Gamma^n \vdash 1 \leq l \leq n \ \mathbf{true}) \quad (\Gamma^n, x\colon D \vdash T : \mathbf{type})}{\mathsf{broadcast}\, l\, x\colon D.T, \ldots, \mathsf{broadcast}\, l\, x\colon D.T : \mathbf{ptype}} \qquad \mathsf{skip}, \ldots, \mathsf{skip} : \mathbf{ptype}$$

$$\frac{(\Gamma^n, x\colon D \vdash T : \mathbf{type})}{\mathsf{val}\, x\colon D.T, \ldots, \mathsf{val}\, x\colon D.T : \mathbf{ptype}} \qquad \frac{T_1, \ldots, T_n : \mathbf{ptype} \quad T'_1, \ldots, T'_n : \mathbf{ptype}}{(T_1; T'_1), \ldots, (T_n; T'_n) : \mathbf{ptype}}$$

$$\frac{\Gamma^n \vdash p : \mathbf{prop} \quad (T_1, \ldots, T_n) : \mathbf{ptype} \quad (T'_1, \ldots, T'_n) : \mathbf{ptype}}{p\,?\,T_1 : T'_1, \ldots, p\,?\,T_n : T'_n : \mathbf{ptype}}$$

$$\frac{(\Gamma^n \vdash l \neq m \ \mathbf{true}) \quad (\Gamma^n \vdash D : \mathbf{dtype})}{\mathsf{skip}_1, \ldots, (\mathsf{message}\, l\, m\, D), \mathsf{skip}_{l+1}, \ldots, \mathsf{skip}_{m-1}, (\mathsf{message}\, l\, m\, D), \ldots, \mathsf{skip}_n : \mathbf{ptype}}$$

$$\frac{T_1, \ldots, T_k, \ldots, T_n : \mathbf{ptype} \quad \Gamma^{n,k} \vdash T_k \equiv T'_k : \mathbf{type}}{T_1, \ldots, T'_k, \ldots, T_n : \mathbf{ptype}}$$

Fig. 5. Program type formation

requires programs to be composed of two parts, both of which conform to program types. The rule for the collective conditional requires all propositions to agree, forcing all process to take identical decisions. The rule for messages requires the vector of term types to contain two identical message types; the "rest" of the vector must itself be composed of skip types. Finally, the last rule allows to replace equal term types in program types.

With these rules in place we can easily check that the following vector of term types is a program type.

$$(\mathsf{message}\ 1\ 3; \mathsf{message}\ 2\ 4), \mathsf{message}\ 2\ 4, (\mathsf{message}\ 1\ 3; \mathsf{message}\ 2\ 4), \mathsf{message}\ 2\ 4$$

because

$$(\mathsf{message}\ 1\ 3; \mathsf{message}\ 2\ 4), (\mathsf{skip}; \mathsf{message}\ 2\ 4),$$
$$(\mathsf{message}\ 1\ 3; \mathsf{message}\ 2\ 4), (\mathsf{skip}; \mathsf{message}\ 2\ 4)$$

is a program type, because both

$$\mathsf{message}\ 1\ 3, \mathsf{skip}, \mathsf{message}\ 1\ 3, \mathsf{skip}$$
$$\mathsf{message}\ 2\ 4, \mathsf{message}\ 2\ 4, \mathsf{message}\ 2\ 4, \mathsf{message}\ 2\ 4$$

are program types.

Deciding when an arbitrary vector of term types constitutes a program type may not be an easy task. There is however a simple case: that of a vector of

Datatype formation, $\Gamma \vdash D : \mathbf{dtype}$

$$\frac{\Gamma \vdash D : \mathbf{dtype}}{\Gamma \vdash D\,\mathsf{ref} : \mathbf{dtype}}$$

Index formation, $\Gamma \vdash i \colon D$

$$\frac{\Gamma : \mathbf{context} \quad r \colon D \in \Gamma}{\Gamma \vdash r : D} \qquad \frac{\Gamma \vdash i : D}{\Gamma \vdash \mathsf{mkref}\, i : D\,\mathsf{ref}} \qquad \frac{\Gamma \vdash i : D\,\mathsf{ref}}{\Gamma \vdash\, !i : D} \qquad \frac{\Gamma \vdash i_1 : D\,\mathsf{ref} \quad \Gamma \vdash i_2 : D}{\Gamma \vdash i_1 := i_2 : D}$$

Context formation, $\Gamma : \mathbf{context}$

$$\frac{\Gamma : \mathbf{context} \quad \Gamma \vdash D : \mathbf{dtype} \quad r \notin \Gamma, D}{\Gamma, r \colon D : \mathbf{context}}$$

Datatype subtyping, $\Gamma \vdash D <: D$

$$\frac{\Gamma \vdash D_1 \equiv D_2}{\Gamma \vdash D_1\ \mathsf{ref} <: D_2\ \mathsf{ref}}$$

Fig. 6. Reference formation

identical types. Such a program type is of particular relevance for models of computation where all processes share the same code, hence the same initial term type.

Lemma 2 (load).

$$\frac{\Gamma^n \vdash T : \mathbf{type} \quad \ldots \quad \Gamma^n \vdash T : \mathbf{type}}{\underbrace{T, \ldots, T}_{n \text{ copies}} : \mathbf{ptype}}$$

Proof. A simple analysis on the ten type constructors available, using type equivalence in the case of messages and primitive recursion.

3 A core parallel imperative programming language

This section introduces our core programming language, including references expressions, stores, processes and programs, as well as the main results of the paper: agreement for programs (Theorem 1) and progress for programs (Theorem 3).

3.1 References

To deal with imperative features, we introduce the notion of *references*. We rely on an extra base set, that of *reference identifiers*, ranged over by r. References have impact on datatypes, index terms, contexts, and index subtyping. The rules in Figure 6 extend those in Figure 1. A new datatype, $D\,\mathsf{ref}$, describes references to values of type D. Four new index terms are introduced: references r, and the conventional operations on references. These include $\mathsf{mkref}\, i$, which evaluates

index term i and returns a reference to the value, $!i$ which retrieves the value associated to the reference described by i, and $i_1 := i_2$ which replaces the value associated with reference i_1 by the value of index term i_2. We also need a new formation rule for ref datatypes. Typing contexts Γ may now contain reference entries $r : D$ in addition to variable entries $x : D$; the new formation rule is in Figure 6. We can easily check that the various definitions are still well formed (cf. Lemma 1), and that index term subtyping remains a pre-order.

We designed our programming language in such a way that it directly handles index terms. The pure index terms of Figure 1 are however extended with effectful operations, such as reference creation and assignment. The meaning of expressions with effects when they occur as index objects to type families is undetermined. For this reason we are careful in requiring index objects appearing in types to remain pure.

3.2 Expressions

The constructors of our language can intuitively be divided in two parts: conventional expressions usually found in a while-language and communication-specific expressions. The rules in Figure 7 characterise what it means to be an *expression* e of a type T under a context Γ, abbreviated to $\Gamma \vdash e : T$.

In an expression of the form $\mathsf{send}\, i_1\, i_2$, index term i_1 (of datatype int) denotes the target process and index term i_2 (of datatype D) describes the value to be sent. The type of the send expression is $\mathsf{message}\, \mathsf{rank}\, i_1\, D$, representing a message from process rank to process i_1 containing a value of datatype D. The premises come naturally if one considers the hypothesis necessary for $\mathsf{message}\, \mathsf{rank}\, i_1\, D$ to be considered a type under context Γ, namely, i_1 must denote a valid process number and must be different from the sender's rank. The value to be sent, i_2, must naturally be of datatype D, so that it conforms to the value the message is supposed to exchange. An expression of the form $\mathsf{receive}\, i_1\, i_2$ denotes the reception of a value (of datatype D) from process i_1. The value is stored in the reference (of datatype $D\,\mathsf{ref}$) denoted by index term i_2. The type of the expression is $\mathsf{message}\, i_1 \mathsf{rank}\, D$, expressing the fact that a message is transmitted from process i_1 to the target process rank.

The reduce expression requires three index term arguments: the first is the target process (the one that receives the maximum of the values proposed by all processes), the second is the value each process proposes, and the third is the reference on which the target process saves the maximum of all values. The premises require i_1 to denote a valid rank, i_2 to be of float datatype, and i_3 to be a reference to a float. The type of the expression is simply $\mathsf{reduce}\, i_1$, for the values transmitted and received are not captured by the type, since it will always be a float.

The gather expression is similar to reduce on what concerns the meaning of its three arguments. The first is the target process (the one receiving the various subarrays), the second denotes the subarrays proposed by the various processes, and the third is the reference that will hold the concatenation of the subarrays. The premises reflect these conditions; notice how the types for the arrays embody

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size} \wedge i_1 \neq \mathsf{rank}\ \mathbf{true} \quad \Gamma \vdash i_2 : D}{\Gamma \vdash \mathsf{send}\, i_1\, i_2 : \mathsf{message}\, \mathsf{rank}\, i_1\, D}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size} \wedge i_1 \neq \mathsf{rank}\ \mathbf{true} \quad \Gamma \vdash i_2 : D\ \mathsf{ref}}{\Gamma \vdash \mathsf{receive}\, i_1\, i_2 : \mathsf{message}\, i_1\, \mathsf{rank}\, D}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : \mathsf{float} \quad \Gamma \vdash i_3 : \mathsf{float\ ref}}{\Gamma \vdash \mathsf{reduce}\, i_1\, i_2\, i_3 : \mathsf{reduce}\, i_1}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : D\ \mathsf{array\ ref} \quad \Gamma \vdash i_3 : \{x\colon D\ \mathsf{array} \mid \mathsf{len}(x) = \mathsf{size} * \mathsf{len}(i_2)\}}{\Gamma \vdash \mathsf{gather}\, i_1\, i_2\, i_3 : \mathsf{gather}\, i_1\, (D\ \mathsf{array})}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : \{x\colon D\ \mathsf{array} \mid \mathsf{len}(i_3) = \mathsf{size} * \mathsf{len}(x)\}\ \mathsf{ref} \quad \Gamma \vdash i_3 : D\ \mathsf{array}}{\Gamma \vdash \mathsf{scatter}\, i_1\, i_2\, i_3 : \mathsf{scatter}\, i_1\, (D\ \mathsf{array})}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : D \quad \Gamma, x\colon D \vdash e : T \quad \mathsf{rank} \notin \mathrm{fv}(i_1)}{\Gamma \vdash \mathsf{let}\, x\colon D = \mathsf{broadcast}\, i_1\, i_2\, \mathsf{in}\, e : \mathsf{broadcast}\, i_1\, x\colon D.T}$$

$$\frac{\Gamma \vdash i : D \quad \Gamma, x\colon D \vdash e : T}{\Gamma \vdash \mathsf{let}\, x\colon D = \mathsf{val}\, i\, \mathsf{in}\, e : \mathsf{val}\, x\colon D.T}$$

$$\frac{\Gamma \vdash p : \mathbf{prop} \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad \mathsf{rank} \notin \mathrm{fv}(p)}{\Gamma \vdash \mathsf{ifc}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : p\,?\,T_1 : T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1; e_2 : T_1; T_2} \qquad \frac{(\Gamma : \mathbf{context})}{\Gamma \vdash \mathsf{skip} : \mathsf{skip}} \qquad \frac{\Gamma, x\colon \{y\colon \mathsf{int} \mid y \le i\} \vdash e : T}{\Gamma \vdash \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e : \forall x \le i.T}$$

$$\frac{\Gamma \vdash p : \mathbf{prop} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T} \qquad \frac{\Gamma \vdash p : \mathbf{prop} \quad \Gamma \vdash e : \mathsf{skip}}{\Gamma \vdash \mathsf{while}\, p\, \mathsf{do}\, e : \mathsf{skip}}$$

$$\frac{\Gamma \vdash i : D \quad \Gamma, x\colon D \vdash e : T \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash \mathsf{let}\, x\colon D = i\, \mathsf{in}\, e : T} \qquad \frac{\Gamma \vdash e : T_1 \quad \Gamma \vdash T_1 \equiv T_2 : \mathbf{type}}{\Gamma \vdash e : T_2}$$

In all rules, T and D contain no ref datatypes.

Fig. 7. Expression formation

the relation between their lengths, as required by the type formation rule in Figure 1. The rule for the scatter expression is similar, except that the order of the last two parameters is exchanged (i_2 denotes the subarrays received by each process and i_3 the array to be distributed), in such a way that the last parameter is evaluated only at the root process.

In a broadcast expression, index term i_1 denotes the root process and index term i_2 the value to be distributed. The index term denoting the root process cannot refer to the special variable rank, for this has different values at different processes, precluding all processes from agreeing on a common root process. Contrary to the expressions studied so far, where the object of communications is stored in a reference, the value distributed by the root process is collected in a variable x and made available to an explicit continuation expression e. This strategy keeps the expression and the type aligned, as made clear by the type formation rule: variable x (of datatype D) is moved into the context to type the continuation, while retaining its presence in the dependent type for broadcast. The same applies to expression val, where the value of the index term i is made

available, via variable x, to the continuation expression e, while being present in the val-dependent type. The rule for the collective conditional expression requires variable rank not to occur in the proposition. The restriction allows all processes to decide equally, given that rank has different values in different processes.

The expression formation rule for sequential composition $e_1; e_2$ is standard, except perhaps for its type, $T_1; T_2$, composed of the types T_1 and T_2 for expressions e_1 and e_2. Expression skip has type skip as expected, even though many other expressions may have this type. To inhabit the $\forall$-type, the language counts with a for loop. In $\mathsf{for}\, x\colon i..1\, \mathsf{do}\, e$, variable x takes values $i, i-1, \ldots, 1$ in each different iteration of the loop. The rule for the conditional expression is standard, and so is the one for the while loop. Notice that the $\mathsf{while}\, p\, e$ expression requires e to be of type skip, not allowing the loop to perform any communication action. If communications are required in a loop body, then a for loop must be used. An expression of the form $\mathsf{let}\, x\colon D = i\, \mathsf{in}\, e$ evaluates index term i and continues as e with variable x replaced by the value of i. The type T of the let is that of the expression e, hence variable x cannot be free in T. The dependent version of this expression is the val expression introduced above. The last expression formation rule introduces type equality. Agreement for expression formation is in Section B.5.

The bindings for expressions are as follows. Variable x is bound in expressions $\mathsf{let}\, x\colon D = \mathsf{broadcast}\, i_1\, i_2\, \mathsf{in}\, e$, $\mathsf{let}\, x\colon D = \mathsf{val}\, i\, \mathsf{in}\, e$, $\mathsf{let}\, x\colon D = i\, \mathsf{in}\, e$, and $\mathsf{for}\, x\colon i..1\, \mathsf{do}\, e$. The notion of free variables, $\mathrm{fv}(e)$, is defined accordingly. Among all index terms, we call *values* to integer constants n, to floating point numbers f, to vectors of values $[v_1, \ldots, v_n]$, and to reference identifiers r, and collectively denote them by v. The notion of substitution, $e\{v/x\}$, is defined in the standard, inductive way, from that of free variables. The standard result of term substitution in dependent types is in Section B.5.

Figure 8 presents a schematic implementation of the finite differences algorithm in our language. We can show that

$$\mathsf{size}\colon \{x\colon \mathsf{int} \mid x \geq 2\}, \mathsf{rank}\colon \{y\colon \mathsf{int} \mid 1 \leq y \leq \mathsf{size}\} \vdash e : T$$

where e is the expression in Figure 8 and T the type in Figure 3. It worth pointing out that the internal $\forall$-type in Figure 3 ($\forall l \leq \mathsf{size}.$) is inhabited by the long for-loop in the second column in Figure 8. The different $\mathsf{send}/\mathsf{receive}$ orders, crucial to ensure absence of deadlock, conform to a type that is made equal to the $\forall$-type via type equality alone. Notice that this sort of code is extremely sensitive to variations, thus reinforcing the potential of the type-based approach to verification of parallel programming.

3.3 Stores

Intuitively *stores* are maps from reference identifiers into values. The formation rules are in Figure 9. Stores can be easily converted into contexts; the rules in Figure 9 determine the meaning of assertions $\rho\, to\, \Gamma$. A store entry of the form $r := v$ is transformed into a context entry $r\colon D\, \mathsf{ref}$, if the initial part of the store

```
let n: {x: int | x ≥ 0 ∧ x%size = 0} =
  val <read-the-dimension> in
let max: int ref = mkref 0 in
let global: {a: float array | len(a) = n} ref =
  mkfloatarray n in
if rank = 1 then
  max := <read-max-iterations>;
  global := <read-the-global-array>;
let m: int = broadcast 1 !max in
let local: {b: float array | len(b) * size = n}
  ref = mkfloatarray (n/size) in
scatter 1 local !global;
let left: {x: int | 1 ≤ x ≤ size} =
  (rank − 2 + size)%size + 1 in
let right: {x: int | 1 ≤ x ≤ size} =
  rank%size + 1 in
let lRecv : float ref = mkref 0.0 in
let rRecv : float ref = mkref 0.0 in
let gError: float ref = mkref 0.0 in
let lError: float ref = mkref 0.0 in
for i: m..1 do
  if rank = 1 then
    send left local[1]
    send right local[n/size]
    receive right !rRecv
    receive left !lRecv
  else if rank = size then
    receive right !rRecv
    receive left !lRecv
    send left local[1]
    send right local[n/size]
  else
    receive left !lRecv
    send left local[1]
    send right local[size/n]
    receive right !rRecv
  <compute-next-local-and-lError>
  let x: float = allreduce !lError in
  gError := x;
gather 1 !local global
```

Fig. 8. The algorithm for finite differences

Stores, ρ : **store**

$$\varepsilon : \textbf{store} \qquad \frac{\rho : \textbf{store} \quad r \notin \rho \quad \rho \; to \; \Gamma \quad \Gamma \vdash v : D}{\rho, r := v : \textbf{store}}$$

Stores as contexts, $\rho \; to \; \Gamma$

$$(\varepsilon : \textbf{store}) \; to \; (\varepsilon : \textbf{context}) \qquad \frac{\rho \; to \; \Gamma \quad \Gamma \vdash v : D \quad (r \notin \rho, \Gamma, D)}{(\rho, r := v) \; to \; (\Gamma, r : D \; \mathsf{ref})}$$

Fig. 9. Store formation and store-to-context conversion

is transformed in context Γ and $\Gamma \vdash v : D$. We can easily check that agreement holds (see Section B.6).

In the sequel we abuse the notation and write ρ where a context is expected. For example $\rho \vdash i : D$ means $\Gamma \vdash i : D$ where $\rho \; to \; \Gamma$. *Store update*, notation $\rho[r := v]$, is the store $\rho', r := v, \rho''$ if ρ is of the form $\rho', r := v', \rho''$ and $\rho' \vdash r : D \; \mathsf{ref}$ and $\rho' \vdash v : D$.

$$\frac{(\rho : \mathbf{store})}{(\rho, m)\downarrow^{n,k}(\rho, m) : \mathsf{int}} \quad \frac{(\rho : \mathbf{store})}{(\rho, f)\downarrow^{n,k}(\rho, f) : \mathsf{float}} \quad \frac{(\rho : \mathbf{store})}{(\rho, \mathsf{size})\downarrow^{n,k}(\rho, n) : \mathsf{int}}$$

$$\frac{(\rho : \mathbf{store})}{(\rho, \mathsf{rank})\downarrow^{n,k}(\rho, k) : \mathsf{int}} \quad \frac{r := v \in \rho \quad \rho \vdash v : D \quad (\rho : \mathbf{store})}{(\rho, r)\downarrow^{n,k}(\rho, r) : D\,\mathsf{ref}}$$

$$\frac{(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D \quad r \notin \rho_2}{(\rho_1, \mathsf{mkref}\, i)\downarrow^{n,k}((\rho_2, r := v), r) : D\,\mathsf{ref}} \quad \frac{(\rho_1, i)\downarrow^{n,k}(\rho_2, r) : D\,\mathsf{ref} \quad r := v \in \rho_2}{(\rho_1, !i)\downarrow^{n,k}(\rho_2, v) : D}$$

$$\frac{(\rho_1, i_1)\downarrow^{n,k}(\rho_2, r) : D\,\mathsf{ref} \quad (\rho_2, i_2)\downarrow^{n,k}(\rho_3, v) : D}{(\rho_1, i_1 := i_2)\downarrow^{n,k}(\rho_3[r := v], v) : D}$$

$$\frac{(\rho_1, i_1)\downarrow^{n,k}(\rho_2, v_1) : \mathsf{int} \quad (\rho_2, i_2)\downarrow^{n,k}(\rho_3, v_2) : \mathsf{int}}{(\rho_1, i_1 + i_2)\downarrow^{n,k}(\rho_3, v_1 + v_2) : \mathsf{int}}$$

$$\frac{(\rho_1, i_1)\downarrow^{n,k}(\rho_2, v_1) : D \quad \ldots \quad (\rho_n, i_n)\downarrow^{n,k}(\rho_{n+1}, v_n) : D}{(\rho_1, [i_1, \ldots, i_n])\downarrow^{n,k}(\rho_{n+1}, [v_1, \ldots, v_n]) : D\,\mathsf{array}}$$

$$\frac{\begin{array}{c}(\rho_1, i_1)\downarrow^{n,k}(\rho_2, [v_1, \ldots, v_l]) : \{x : D\,\mathsf{array} \mid \mathsf{len}(x) = l\} \\ (\rho_2, i_2)\downarrow^{n,k}(\rho_3, m) : \{y : \mathsf{int} \mid 1 \le y \le l\}\end{array}}{(\rho_1, i_1[i_2])\downarrow^{n,k}(\rho_3, v_m) : D}$$

$$\frac{(\rho_1, i)\downarrow^{n,k}(\rho_2, [v_1, \ldots, v_n]) : D\,\mathsf{array}}{(\rho_1, \mathsf{len}(i))\downarrow^{n,k}(\rho_2, n) : \mathsf{int}} \quad \frac{(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D_1 \quad \rho_2 \vdash D_2 <: D_1}{(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D_2}$$

$$\frac{(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D \quad \Gamma^{n,k}, \rho_2 \vdash p\{i/x\}\ \mathbf{true}}{(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : \{x : D \mid p\}}$$

Fig. 10. Index term evaluation

Index terms are evaluated against a store; evaluation also resolves the distinguished variables size and rank. The rules in Figure 10 are meaning determining for assertions of the form $(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D$, abbreviating "index term i of datatype D evaluates under store ρ_1, size $= n$, and rank $= k$, yielding a value v of datatype D and a new store ρ_2". The rules are straightforward; we briefly describe them. The two rules for constants (integer and floating point) evaluate the constants to themselves, keeping the store unchanged. The two rules for the special variables (size and rank) read the variables' values from the parameters n and k, again keeping the store unchanged. The rules for references and for arithmetic and array operations, should be easy to understand based on the explanations above and their intuitive meaning. The last two rule introduce refinement types and subtyping, necessary, e.g., to ensure that index term $i_1[i_2]$ is well-formed in the evaluation rule for array access; they should not be needed when the rules are used in "evaluation" mode. Agreement for evaluation is in Section B.6.

The following result links deducibility to index term evaluation, stating that evaluation succeeds on well formed index terms, thus contributing to the progress result.

$$\frac{(\rho, i)\downarrow^{n,k}(\rho', v)\colon D \quad (\Gamma^{n,k}, \rho, x\colon D \vdash e : T) \quad (x \notin \mathrm{fv}(T))}{(\rho, \mathsf{let}\, x\colon D = i\, \mathsf{in}\, e) \to^{n,k} (\rho', e\{v/x\})}$$

$$\frac{\Gamma^{n,k}, \rho \vdash p\ \mathbf{true} \quad (\Gamma^{n,k}, \rho \vdash e_1, e_2 : T)}{(\rho, \mathsf{if}\ p\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \to^{n,k} (\rho, e_1)} \qquad \frac{\Gamma^{n,k}, \rho \vdash \neg p\ \mathbf{true} \quad (\Gamma^{n,k}, \rho \vdash e_1, e_2 : T)}{(\rho, \mathsf{if}\ p\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \to^{n,k} (\rho, e_2)}$$

$$\frac{\Gamma^{n,k}, \rho \vdash p\ \mathbf{true} \quad (\Gamma^{n,k}, \rho \vdash e : \mathsf{skip})}{(\rho, \mathsf{while}\ p\ \mathsf{do}\ e) \to^{n,k} (\rho, (e; \mathsf{while}\ p\ \mathsf{do}\ e))} \qquad \frac{\Gamma^{n,k}, \rho \vdash \neg p\ \mathbf{true} \quad (\Gamma^{n,k}, \rho \vdash e : \mathsf{skip})}{(\rho, \mathsf{while}\ p\ \mathsf{do}\ e) \to^{n,k} (\rho, \mathsf{skip})}$$

$$\frac{\Gamma^{n,k}, \rho \vdash i \geq 1\ \mathbf{true} \quad (\Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash e : T)}{(\rho, \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e) \to^{n,k} (\rho, (e\{i/x\}; \mathsf{for}\, x\colon i - 1..1\, \mathsf{do}\, e))}$$

$$\frac{\Gamma^{n,k}, \rho \vdash i < 1\ \mathbf{true} \quad (\Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash e : T)}{(\rho, \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e) \to^{n,k} (\rho, \mathsf{skip})}$$

$$\frac{(\rho, e_1) \to^{n,k} (\rho', e_3) \quad (\Gamma^{n,k}, \rho \vdash e_2 : T)}{(\rho, (e_1; e_2)) \to^{n,k} (\rho', (e_3; e_2))} \qquad \frac{(\Gamma^{n,k}, \rho \vdash e : T)}{(\rho, (\mathsf{skip}; e)) \to^{n,k} (\rho, e)}$$

Fig. 11. Process reduction

Lemma 3 (evaluation succeeds).

$$\frac{\Gamma^{n,k}, \rho_1 \vdash i : D}{(\rho_1, i)\downarrow^{n,k}(\rho_2, v)\colon D}$$

Proof. By rule induction on the hypothesis.

3.4 Processes

A *process* p is a pair (ρ, e) composed of a store ρ and an expression e. The rule below determines the meaning for assertions of the form $\Gamma \vdash q : T$.

$$\frac{\Gamma, \rho \vdash e : T}{\Gamma \vdash (\rho, e) : T}$$

The rules in Figure 11 determine what it means for a process q_1 to *reduce* to a process q_2 given that $\mathsf{size} = n$ and $\mathsf{rank} = k$, that is, they determine the meaning of assertions $q_1 \to^{n,k} q_2$. The rules should be self-explanatory. The let expression evaluates index i to value v and proceeds with expression e with v replacing variable x. Since let is a local (process) operation, x cannot be free in T, as discussed before. The premises in parenthesis guarantee the good formation of the stores and expressions involved. The remaining rules—for conditionals, while and for loops, and sequential composition—are standard. Notice that process reduction does not change the type of the expressions involved. That the definition is well formed follows from the following result.

Lemma 4 (agreement for process reduction).

$$\frac{q \to^{n,k} q'}{\Gamma^{n,k} \vdash q : T \quad \Gamma^{n,k} \vdash q' : T}$$

Proof. By rule induction on the hypothesis.

Lemma 5 (process reduction is deterministic).

$$\frac{q_1 \to^{n,k} q_2 \quad q_1 \to^{n,k} q_3}{q_2 = q_3}$$

Proof (sketch). The case for sequential composition follows by a simple induction. All other cases follow from the fact that evaluation is deterministic and from the fact that either p or $\neq p$ is true.

The following lemma ensures that processes do not get stuck and will play its part in the main result of the paper.

Lemma 6 (progress for processes).

$$\frac{\Gamma^{n,k}, \rho \vdash e : \mathsf{skip}}{e \ \mathit{is}\ \mathsf{skip} \quad \mathit{or} \quad (\rho, e) \to^{n,k} q} \qquad \frac{\Gamma^{n,k}, \rho \vdash i : D \quad \Gamma^{n,k}, \rho, x\colon D \vdash e : T \quad x \notin \mathrm{fv}(T)}{(\rho, \mathsf{let}\, x\colon D = i\, \mathsf{in}\, e) \to^{n,k} q}$$

$$\frac{\Gamma^{n,k}, \rho \vdash p : \mathbf{prop} \quad \Gamma^{n,k}, \rho \vdash e_1, e_2 : T}{(\rho, \mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2) \to^{n,k} q} \qquad \frac{\Gamma^{n,k}, \rho \vdash p : \mathbf{prop} \quad \Gamma^{n,k}, \rho \vdash e : \mathsf{skip}}{(\rho, \mathsf{while}\, p\, \mathsf{do}\, e) \to^{n,k} q}$$

$$\frac{\Gamma^{n,k}, \rho \vdash i : \mathsf{int} \quad \Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash e : T}{(\rho, \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e) \to^{n,k} q}$$

Proof (sketch). By analysis of the hypotheses. For example, in the let rule, building from $\Gamma^{n,k}, \rho \vdash i : D$ and the fact that evaluation succeeds (Lemma 3), we obtain $(\rho, i) \downarrow^{n,k} (\rho', v)\colon D$. This, combined with premises $\Gamma^{n,k}, \rho, x\colon D \vdash e : T$ and $x \notin \mathrm{fv}(T)$ constitute the necessary conditions to apply reduction for let processes in Figure 11, obtaining $(\rho, \mathsf{let}\, x\colon D = i\, \mathsf{in}\, e) \to^{n,k} (\rho', e\{v/x\})$. All other cases excepting skip are similar. For $\Gamma^{n,k}, \rho \vdash e : \mathsf{skip}$, we rely on the inversion lemma for expression formation (cf. proof of Theorem 3) and analyse all expressions such that $\Gamma \vdash e : \mathsf{skip}$, showing that e is either skip or there exists a process reduction rule such that $(\rho, e) \to^{n,k} (\rho', e')$.

3.5 Programs

A *program* is a vector of processes $q_1, \ldots, q_n$. Not all such vectors are of interest to us. The following rule is meaning determining for assertions of the form $P : S$.

$$\frac{\Gamma^{n,1} \vdash q_1 : T_1 \quad \ldots \quad \Gamma^{n,n} \vdash q_n : T_n \quad T_1, \ldots, T_n : \mathbf{ptype}}{q_1, \ldots, q_n : T_1, \ldots, T_n}$$

The rules in Figure 12 determine what it means for a program P_1 to *reduce* to a program P_2, that is, they determine the meaning of assertions $P_1 \to P_2$. Program reduction is composed of six *collective* barrier-like rules—for send/receive, reduce, scatter, gather, broadcast, and val—one rule for collective decisions, and one rule that provides for *local* process reduction. As in the previous cases, the premises to the rule may be divided in two parts: those governing the reduction

$$\frac{i_l \downarrow^n m \quad (\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v)\colon D \quad i_m \downarrow^n l \quad (\rho_m, i'_m)\downarrow^{n,m}(\rho'_m, r)\colon D\,\mathsf{ref} \quad (l \neq m) \\ (\Gamma^{n,l} \vdash e_l : T) \quad (\Gamma^{n,m} \vdash e_m : T) \quad (\Gamma^{n,k} \vdash q_k : T) \quad {\scriptstyle (k=1..n,\, k\neq l,m)}}{q_1,\ldots,q_{l-1},(\rho_l, \mathsf{send}\ i_l\ i'_l; e_l), q_{l+1},\ldots,q_{m-1},(\rho_m, \mathsf{receive}\ i_m\ i'_m; e_m), q_{m+1}\ldots,q_n \to \\ q_1,\ldots,q_{l-1},(\rho'_l, e_l), q_{l+1},\ldots,q_{m-1},(\rho'_m[r := v], e_m), q_{m+1}\ldots,q_n}$$

$$\frac{i_k \downarrow^n l \quad (\rho_k, i'_k)\downarrow^{n,k}(\rho'_k, f_k)\colon \mathsf{float} \\ (\rho'_l, i''_l)\downarrow^{n,l}(\rho''_l, r)\colon \mathsf{float\,ref} \quad (\Gamma^n \vdash 1 \le i_k \le n\ \mathbf{true}) \quad {\scriptstyle (k=1..n)}}{(\rho_k, \mathsf{reduce}\ i_k\ i'_k\ i''_k)_{k=1}^n \to (\rho'_k, \mathsf{skip})_{k=1}^{l-1}, (\rho''_l[r := \mathsf{max}(v_1,..,v_n)], \mathsf{skip}), (\rho'_k, \mathsf{skip})_{k=l+1}^n}$$

$$\frac{i_k \downarrow^n l \quad (\rho_k, i'_k)\downarrow^{n,k}(\rho'_k, r_k)\colon \{x\colon D\,\mathsf{array} \mid \mathsf{len}(x) * \mathsf{size} = \mathsf{len}(i''_k)\}\,\mathsf{ref} \quad {\scriptstyle (k=1..n)} \\ (\rho'_l, i''_l)\downarrow^{n,l}(\rho''_l, [\vec{v}_1,\ldots,\vec{v}_n])\colon D\,\mathsf{array} \quad (\Gamma^n \vdash 1 \le i_k \le n\ \mathbf{true}) \quad (\Gamma^{n,k}, \rho_k \vdash i''_k : D\,\mathsf{array})}{(\rho_k, \mathsf{scatter}\ i_k\ i'_k\ i''_k)_{k=1}^n \to \\ (\rho'_k[r_k := [\vec{v}_k]], \mathsf{skip})_{k=1}^{l-1}, (\rho''_l[r_l := [\vec{v}_l]], \mathsf{skip}), (\rho'_k[r_k := [\vec{v}_k]], \mathsf{skip})_{k=l+1}^n}$$

$$\frac{i_k \downarrow^n l \quad (\rho_k, i'_k)\downarrow^{n,k}(\rho'_k, [\vec{v}_k])\colon D\,\mathsf{array} \quad (\rho'_l, i''_l)\downarrow^{n,l}(\rho''_l, r)\colon \{x\colon D\,\mathsf{array} \mid \mathsf{len}(x) = \mathsf{size} * \mathsf{len}(i'_k)\}\,\mathsf{ref} \\ (\Gamma^n \vdash 1 \le i_k \le n\ \mathbf{true}) \quad (\Gamma^{n,k}, \rho_k \vdash i'_k : D\,\mathsf{array}) \ {\scriptstyle (k=1..n)}}{(\rho_k, \mathsf{gather}\ i_k\ i'_k\ i''_k)_{k=1}^n \to (\rho'_k, \mathsf{skip})_{k=1}^{l-1}, (\rho''_l[r_l := [\vec{v}_1,\ldots,\vec{v}_n]], \mathsf{skip}), (\rho'_k, \mathsf{skip})_{k=l+1}^n}$$

$$\frac{i_k \downarrow^n l \quad (\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v)\colon D \quad (\Gamma^n \vdash 1 \le i_k \le n\ \mathbf{true}) \\ (\Gamma^{n,k}, \rho_k \vdash i'_k : D) \quad (\Gamma^{n,k}, x\colon D, \rho_k \vdash e_k : T) \quad {\scriptstyle (k=1..n)}}{(\rho_k, \mathsf{let}\, x\colon D = \mathsf{broadcast}\, i_k\, i'_k\, \mathsf{in}\, e_k)_{k=1}^n \to \\ (\rho_k, e_k\{v/x\})_{k=1}^{l-1}, (\rho'_l, e_l\{v/x\}), (\rho_k, e_k\{v/x\})_{k=l+1}^n}$$

$$\frac{i_k \downarrow^n v \quad (\Gamma^{n,k}, x\colon D, \rho_k \vdash e_k : T) \quad {\scriptstyle (k=1..n)}}{(\rho_k, \mathsf{let}\ x\colon D = \mathsf{val}\ i_k\ \mathsf{in}\ e_k)_{k=1}^n \to (\rho_k, e_k\{v/x\})_{k=1}^n}$$

$$\frac{\Gamma^n \vdash p_k\ \mathbf{true} \quad (\Gamma^{n,k}, \rho_k \vdash e_k : T) \quad (\Gamma^{n,k}, \rho_k \vdash e'_k : T')}{(\rho_1, \mathsf{ifc}\, p_1\, \mathsf{then}\, e_1\, \mathsf{else}\, e'_1), \ldots, (\rho_n, \mathsf{ifc}\, p_n\, \mathsf{then}\, e_n\, \mathsf{else}\, e'_n) \to (\rho_1, e_1), \ldots (\rho_n, e_n)}$$

$$\frac{(q_k, e_k)_{k=1}^n \to (q'_k, e''_k)_{k=1}^n \quad (\Gamma^{n,k} \vdash e'_k : T_k) \quad (T_1,\ldots,T_n : \mathbf{ptype}) \quad {\scriptstyle (k=1..n)}}{(q_k, (e_k; e'_k))_{k=1}^n \to (q'_k, (e''_k; e'_k))_{k=1}^n}$$

$$\frac{q_l \to^{n,l} q'_l \quad (\Gamma^{n,k} \vdash q_k : T_k) \quad (T_1,\ldots,T_n : \mathbf{ptype}) \quad {\scriptstyle (k=1..n)}}{q_1,\ldots,q_n \to q_1,\ldots,q_{l-1},q'_l,q_{l+1},\ldots,q_n}$$

In all rules, D and T contain no ref types and $\mathsf{rank} \notin \mathrm{fv}(D,T)$
Omitting dual rule for receive-send and the $\Gamma \vdash \neg p_k\ \mathbf{true}$ rule for ifc.

Fig. 12. Program reduction

process itself, and those guaranteeing the good formation of the programs involved. The latter are enclosed in parenthesis, as before. Notation $i \downarrow^n v$ abbreviates the evaluation of an int index term under the empty store, $(\varepsilon, i)\downarrow^n(\varepsilon, v)\colon \mathsf{int}$. The proviso in all rules that types and datatypes do not contain ref datatypes impedes reference passing (and the associated problem of dangling references at the receiving process). A similar reason forbids the rank variable in types, for this variable has a different value in each different process.

The rule for message-passing (the first rule in Figure 12), evaluates both index terms in both the send and the receive process. There is a fundamental

difference between the first and the second parameter in both cases. The first describes a process rank (target or source), the second the value to be passed, or the reference to hold the result. In general, index terms that denote process ranks cannot refer to the store, for these exact indices show up in the type of the processes ($\mathsf{message}\, m\, i_l\, D$, in the send case). In such cases we use the abbreviated evaluation, as in $i_l \downarrow^n m$. In all other cases, we use evaluation under a generic store, as in $(\rho_l, i'_l) \downarrow^{n,l} (\rho'_l, v) \colon D$. The send/receive processes reduce to skip (the stores evolve accordingly); the others remain unchanged. The case of the rule for reduce is illustrative of a feature of single-instruction-multiple-data models: the third parameter is evaluated at one site only (namely process l, cf. $(\rho'_l, i''_l) \downarrow^{n,l} (\rho''_l, r) \colon \mathsf{float\ ref}$), yet the source code (index term i''_k) appears in all processes. In the rule for broadcast (and let) we follow a strategy slightly different from that of reduce (and scatter/gather). Since a value is transmitted to all processes, the broadcast expression features an explicit continuation, allowing to substitute the value directly in the continuation process e_k (and in its type T), as opposed to using references.

That the definition is well formed is the theme of the next theorem.

Theorem 1 (agreement for program reduction).

$$\frac{P_1 \to P_2}{P_1 : S_1 \quad P_2 : S_2}$$

Proof (sketch). By rule induction on assertion $P_1 \to P_2$. There are eight cases to consider. The case for process reduction follows by agreement for process reduction (Lemma 4). The cases for collective operations and message passing all follow a similar strategy. P_1 is $(\rho_k, e_k)_{k=1}^n$, P_2 is $(\rho'_k, e'_k)_{k=1}^n$. From the premises of the rules for $(\rho_k, e_k)_{k=1}^n \to (\rho'_k, e'_k)_{k=1}^n$ we obtain the necessary hypotheses to build the expression typings for $\Gamma^{n,k}, \rho_k \vdash e_k : T$ and $\Gamma^{n,k}, \rho'_k \vdash e'_k : T'$. From agreement of expression formation we know that $\Gamma^{n,k}, \rho_k \vdash T : \mathbf{type}$ and $\Gamma^{n,k}, \rho'_k \vdash T' : \mathbf{type}$. We can eliminate ρ_k and ρ'_k from the contexts (strengthening), given that neither T nor T' contain references. That $T, \ldots, T : \mathbf{ptype}$ follows from the Load lemma (lemma 2). Showing that $T', \ldots, T' : \mathbf{ptype}$ amounts to checking that the premises of the appropriate rules in Figure 5 are satisfied for each case.

Program reduction is Church-Rosser. As usual this does not mean that it is strongly normalising: taking advantage of while-loops, processes may engage in infinite computations.

Theorem 2 (Program reduction is Church-Rosser).

$$\frac{P_1 \to P_2 \quad P_1 \to P_3}{P_2 \to P_4 \quad P_3 \to P_4}$$

Proof (sketch). By rule induction on the first hypothesis. The cases of reduce, scatter, gather, broadcast, val, ifc (both rules) follow from the fact that evaluation is deterministic and that either p or $\neq p$ is true. When $P_1 \to P_2$ is obtained with

send/receive rule, we analyse the possible $P_1 \to P_3$ reductions. Given the form of P_1 only two cases may hold. If P_3 is obtained via the send/receive rule, the two processes involved are different from those involved in the $P_1 \to P_2$ reduction, otherwise P_1 would not be well formed. Then we can easily see that there is P_4 such that $P_2 \to P_4$ and $P_3 \to P_4$, both using the send/receive rule. If, on the other hand, P_3 is obtained via the process rule, then we know that the process involved is neither the send nor the receive process, since these expressions do not reduce via process reduction. Then again, we can easily build a program P_4 such that $P_2 \to P_4$ via the process rule, and $P_3 \to P_4$ via the send/receive rule.

In preparation for the progress result, we determine the meaning of assertions of the form P **halted** using the following rule.

$$\frac{(\rho_1 : \mathbf{store}) \quad \ldots \quad (\rho_n : \mathbf{store})}{(\rho_1, \mathsf{skip}), \ldots, (\rho_n, \mathsf{skip})\ \mathbf{halted}}$$

We are finally in a position to establish our progress result.

Theorem 3 (progress for programs).

$$\frac{P_1 : S}{P_1\ \mathbf{halted} \quad \textit{or} \quad P_1 \to P_2}$$

Proof (sketch). From the hypothesis and the formation rule for programs, we know that S : **ptype**. The proof proceeds by rule induction on this assertion. There are nine cases to consider. The case for the type equality rule follows by induction. For the others we establish an inversion lemma for each of the type constructors in our language. For example, if $\Gamma \vdash e : \mathsf{broadcast}\, i\, x\colon D.T$ then e is either

- $\mathsf{let}\, x\colon D_1 = \mathsf{broadcast}\, i_1\, i_2\, \mathsf{in}\, e_1$ and $\Gamma \vdash D_1 \equiv D : \mathbf{dtype}$ and $\Gamma \vdash i_1 = i\ \mathbf{true}$ and $\Gamma \vdash 1 \leq i_1 \leq \mathsf{size}\ \mathbf{true}$ and $\Gamma \vdash i_2 : D$ and $\Gamma, x\colon D \vdash e_1 : T$, or
- $\mathsf{let}\, y\colon D_1 = i_1\, \mathsf{in}\, e_1$ and $\Gamma \vdash i_1 : D_1$ and $y \notin \mathrm{fv}(\mathsf{broadcast}\, i\, x\colon D.T)$ and $\Gamma, y\colon D_1 \vdash e_1 : \mathsf{broadcast}\, i\, x\colon D.T$, or
- $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ and $\Gamma \vdash p : \mathbf{prop}$ and $\Gamma \vdash e_1 : \mathsf{broadcast}\, i\, x\colon D.T$ and $\Gamma \vdash e_2 : \mathsf{broadcast}\, i\, x\colon D.T$, or
- $e_1; e_2$ and $\Gamma \vdash e_1 : \mathsf{skip}$ and $\Gamma \vdash e_2 : \mathsf{broadcast}\, i\, x\colon D.T$.

We then distinguish two cases: a) the expressions in all processes are of the form $\mathsf{let}\, x\colon D_k = \mathsf{broadcast}\, i_k\, i'_k\, \mathsf{in}\, e_k$, b) at least one of the expressions is not broadcast. In the former case we show that P_1 reduces under the broadcast rule. For the latter, we analyse the three possibilities to conclude that, in each case, P_1 reduces under one of the process reduction rules (Lemma 6).

4 Related work

With more than 20 years of existence MPI [4] is the *de facto* standard for high-performance computing, admittedly the most widely used API for programming distributed message-passing applications. The verification effort for MPI

programs centres on debuggers and software verifiers. Tools such as ISP [22], DAMPI [27], and MUST [8] are runtime verifiers that aim at detecting deadlocks, hence are dependent on the quality of the tests. TASS [26] is a bounded model checker that uses symbolic execution to explore C+MPI program executions, verifying safety properties and functional equivalence to serial programs. The approach performs a number of checks besides deadlock detection (such as, buffer overflows and memory leaks), but, as expected, does not scale with the number of processes. MOPPER [3] searches for deadlocks using SAT and a partial-order encoding, offering a more scalable solution. The purpose of our work is not detecting deadlocks; our type checker ensures that programs cannot possibly deadlock.

The type theory here developed is part of a larger effort aiming at a comprehensive approach to type-based verification of parallel programs [18]. The project includes a toolchain for deductive verification of real world parallel programs written in the C programming language using MPI for inter-process communication. Currently, the toolchain is composed of an Eclipse plug-in, an annotated MPI library, a C annotator, and makes use of the Verifying C Compiler (VCC) and the Z3 SMT solver. The Eclipse plug-in allows for developing protocol specifications (the types in this paper), verifies that protocols are well formed (with the help of Z3 for constraint satisfiability), and generates protocol representations in VCC format. The annotated MPI library contains the contracts of a core subset of MPI primitives, plus the base logic for protocol representation and program-protocol matching. The C annotator is a Clang/LLVM application that analyses the C+MPI source code and generates VCC annotations. The annotated C+MPI program is then checked by VCC to prove its conformance against the given protocol. A key result is that C+MPI programs can be verified in constant time, independently of the number of processes or values for program inputs, unlike other approaches (e.g., model checking and symbolic execution) that stumble on the state-explosion problem. The Eclipse plugin may also synthesise fully functional C+MPI programs that are correct-by-construction. Such code may then be further complemented with user-provided C functions that merely implement the local computations eschewing MPI primitives. We have also transposed the theory presented in this paper to the Why3 language. The Eclipse plugin compiles the protocol into Why3 against which WhyML programs are checked. Details are provided in [10, 14–16, 24].

The rich field of session types [12] provides for a major source of inspiration for this work. From the theory of multiparty session types [9], Scribble [11], a concrete protocol specification language, was developed. Scribble protocols describe high level, global, communication patterns. Under this framework, a protocol is first defined that specifies interactions among all principals. From such a protocol, or global type, local types are obtained by end-point projection onto particular principals. Code may then be developed for each end-point, using specific libraries for the Scribble's message-passing primitives (Session C is one such example [20]). The whole development methodology ensures that the resulting code is exempt from deadlocks. In our case, rather than using two sep-

arate languages for global and local types, we see global types as vectors of local types subject to certain restrictions. The projection operator is cleanly captured in our setting via type equality. Another central distinction is that all our processes run the same code, as opposed to requiring the separate development for each different kind of participant in a distributed computation.

Among all works on session types, the closest to ours is probably that of Deniélou, Yoshida et al. [1], introducing dependent types and a form of primitive recursion into session types. The system here proposed provides for various communication primitives (in contrast to message passing only) and incorporates dependent collective choices. One the other hand, we do not allow passing protocols on messages, a feature not used in the field of parallel programming. At the term level, we work with a while language, as opposed to a variant of the the π-calculus. Kouzapas et al. introduce a notion of broadcast in the setting of session types [13]. A new operational semantics system allows to describe 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 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.

Following Martin-Löf's works on constructive type theory [17], 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), Xi and Pfenning [29] 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 nowadays are available. Our language follows this approach. Xanadu [28] incorporates these ideas in a imperative C-like language. Omega [25] and Liquid Types [23] are two further examples of pure functional languages that either resorting to theorem proving or do type inference. All these languages are functional; their type systems cannot abstract program's communication patterns.

5 Conclusion and further work

We developed a type theory for parallel, message-passing, programming. The type language includes constructs matching those usually found in the practice of parallel programming, namely different forms of communication primitives, as well as sequential composition, primitive recursion, and a novel collective choice operator. Type dependency is taken from the domain of integer, floating point and array values, making type checking decidable. We have also introduced a core while language equipped with primitives matching those at the type level. The main result is soundness in the form of agreement for program reduction (akin to subject-reduction) and progress for programs. The theory has been put into to practice in a number of forms, including the verification of real world C+MPI programs.

Even if processes run in parallel, our language is intrinsically deterministic, hence exempt from races. The practice of MPI programming has shown us that "wildcard receives" (whereby a receive process can match any sender targeting the process) are often used, either for efficiency purposes or simply for easing programming. Wildcard receive introduces additional challenges that we want to look at in the future. MPI is equipped with different communication semantics, including synchronous (or unbuffered, the one addressed in this paper) and buffered. For the latter case, MPI features non-blocking sends together with a wait operation that blocks until the memory space associated to the message contents can be reused. We believe that our framework can accommodate buffered semantics, at the expense of additional complexity both at the type and the operational semantics level. Finally, control-flow based on transmitted data (as opposed to explicit "control" messages usually found in, say, session types) is ubiquitous in MPI programming. Our type language already supports (dependent) collective decisions, but loops (catering for e.g., numerical convergence) introduce additional difficulties which we would like to address.

Acknowledgments. This work is supported by FCT through project Advanced Type Systems for Multicore Programming and project Liveness, Statically (PTDC/EIA-CCO/122547 and 117513/2010) and the LaSIGE lab (PEst-OE/EEI/UI0408/2011). We would like to thank Dimitris Mostrous for his insightful comments.

References

1. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Logical Methods in Computer Science 8(4) (2012)
2. FEVS: A functional equivalence verification suite, `http://vsl.cis.udel.edu/fevs/`
3. Forejt, V., Kroening, D., Narayanswamy, G., Sharma, S.: Precise predictive analysis for discovering communication deadlocks in mpi programs. In: FM. LNCS, vol. 8442, pp. 263–278. Springer (2014)
4. Forum, M.P.I.: MPI: A Message-Passing Interface Standard Version 3.0. University of Tennessee (2012)
5. Foster, I.: Designing and building parallel programs. Addison-Wesley (1995)
6. Gordon, A.D., Fournet, C.: Principles and applications of refinement types. In: Logics and Languages for Reliability and Security, pp. 73–104. IOS Press (2010)
7. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message passing interface. MIT press (1999)
8. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI runtime error detection with MUST: advances in deadlock detection. In: SC. p. 30. IEEE/ACM (2012)
9. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
10. Honda, K., Marques, E.R.B., Ng, N., Vasconcelos, V.T., Yoshida, N.: Verification of MPI programs using session types. In: Recent Advances in the Message Passing Interface. LNCS, vol. 7490, pp. 291–293. Springer (2012)

11. Honda, K., Mukhamedov, A., Brown, G., Chen, T.C., Yoshida, N.: Scribbling interactions with a formal foundation. In: ICDCIT. LNCS, vol. 6536, pp. 55–75. Springer (2011)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998)
13. Kouzapas, D., Gutkovas, R., Gay, S.J.: Session types for broadcasting. In: PLACES. EPTCS, vol. 155, pp. 25–31 (2014)
14. Lemos, F.: Synthesis of correct-by-construction MPI programs. Master's thesis, Department of Informatics, University of Lisbon (2014)
15. Marques, E.R.B., Martins, F., Vasconcelos, V.T., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: PLACES. EPTCS, vol. 137, pp. 103–113 (2013)
16. Marques, E.R.B., Martins, F., Vasconcelos, V.T., Santos, C., Ng, N., Yoshida, N.: Protocol-based verification of MPI programs. DI-FCUL 5, University of Lisbon (2014)
17. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis-Napoli (1984)
18. MPI Sessions, `http://gloss.di.fc.ul.pt/MPISessions`
19. Nelson, N.: Primitive recursive functionals with dependent types. In: MFPS. LNCS, vol. 598, pp. 125–143. Springer (1991)
20. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe parallel programming with message optimisation. In: TOOLS Europe. LNCS, vol. 7304, pp. 202–218. Springer (2012)
21. Pacheco, P.: Parallel programming with MPI. Morgan Kaufmann (1997)
22. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Palmer, R., Thakur, R., Gropp, W.: Practical model-checking method for verifying correctness of MPI programs. In: PVM/MPI. LNCS, vol. 4757, pp. 344–353. Springer (2007)
23. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI. pp. 159–169. ACM (2008)
24. Santos, C.: Protocol based programming of concurrent systems. Master's thesis, Department of Informatics, University of Lisbon (2014)
25. Sheard, T., Linger, N.: Programming in Omega. In: CEFP. LNCS, vol. 5161, pp. 158–227. Springer (2007)
26. Siegel, S.F., Zirkel, T.K.: Automatic formal verification of MPI-based parallel programs. In: PPOPP. pp. 309–310. ACM (2011)
27. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: SC. pp. 1–10. IEEE (2010)
28. Xi, H.: Imperative programming with dependent types. In: LICS. pp. 375–387. IEEE (2000)
29. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL. pp. 214–227. ACM (1999)

A Further examples

This section presents a few examples attesting the flexibility of the type language and the programming language.

A.1 Diffusion 1-D

The **Diffusion 1-D** example calculates the evolution of the diffusion (heat) equation in one dimension over time. The program iterates for a given number of steps, computing at each step the new temperatures at each point in the domain. This example was adapted from the FEVS benchmark suite [2].

The type for the diffusion 1-D example:

```
1   val maxIter: {x: nat | x > 0}.
2   val n: {x: nat | x%size = 0}.
3   broadcast 1 a: int.
4   broadcast 1 b: float.
5   broadcast 1 c: float.
6   broadcast 1 d: float.
7   ∀i ≤ size − 1.
8     message 1 (i + 1) {x: float array | len(x) = n/size};
9   ∀iter ≤ maxIter.
10    ∀i ≤ size − 1.
11      message (i + 1) i float;
12    ∀i ≤ size − 1.
13      message i (i + 1) float
```

The code for the diffusion 1-D example:

```
1   let on: float = val if rank = 1 then 0.01 else 0.0 in
2   let omaxIter: float = val if rank = 1 then 10000 else 0 in
3   let ok: float = if rank = 1 then 0.01 else 0.0 in
4   let owstep: int = if rank = 1 then 10 else 0 in
5   let n: int = broadcast 1 on in
6   let k: float = broadcast 1 ok in
7   let maxIter: int = broadcast 1 omaxIter in
8   let wstep: int = broadcast 1 owstep in
9   let localn: int = n/size in
10  let u: float array ref = mkref mkfloatarray localn + 2 in
11  let unew: float array ref = mkref mkfloatarray localn + 2 in
12  if rank = 1 then
13    let buf: float array = <fill-buffer> in
14    for i: size − 1..1 do
15      send (i + 1) buf
16  else
17    receive 1 u;
18  let left: int = rank − 1 in
19  let right: int = rank + 1 in
20  for iter: maxIter..1 do
21    if rank > 1 then
```

```
22          send left !u
23        else
24          receive right u;
25        if rank > 1 then
26          receive right u
27        else
28          send left !u;
29        <update-unew>
```

A.2 Jacobi iteration

The **Jacobi iteration** example solves a linear equation of the form $Ax = b$ using Jacobi's method. Each diagonal element is solved for, and an approximate value is plugged in. The example was changed to use a maximum number of iterations in place of the original data convergence method. This example was adapted from the Parallel Programming with MPI book [21].

The type for the Jacobi iteration example:

```
1   val n: {x: positive | x%size = 0}.
2   val maxIter: int.
3   scatter 1 {x: float array | len(x) = n * n};
4   scatter 1 {x: float array | len(x) = n};
5   allgather a: {x: float array | len(x) = n/size}.
6   ∀i ≤ maxIter.
7     (allgather a: {x: float array | len(x) = n/size}.
8     skip);
9   gather 1 {x: float array | len(x) = n/size}
```

The code for the Jacobi iteration example:

```
1   let Ainitial: int array = n * n in
2   let binitial: int array = n in
3   let n: int = val <get-problem-size> in
4   let maxIter: int = val <get-max-iterations> in
5   let nsplit: int = n/size in
6   let Alocal: float array ref = mkref n * nsplit in
7   let xlocal: float array ref = mkref nsplit in
8   let blocal: float array ref = mkref nsplit in
9   let xfinal: float array ref = mkref n in
10  let xtemp2: float array ref = mkref n in
11  scatter 1 Alocal !Ainitial;
12  scatter 1 blocal !binitial;
13  for i: nsplit..1 do
14    <initialize-xlocal>;
15  let xold: float array = allgather !xlocal in
16  for iter: maxIter..1 do
17    (for i: nsplit..1 do
18      <update-xlocal>;
19    let xnew: float array = allgather !xlocal in skip);
```

```
20          <swap-xold-xnew>;
21        for i : nsplit..1 do
22          <initialize-xlocal>;
23        gather 1 !xlocal xfinal
```

A.3 Laplace solver

The **Laplace solver** example iteratively calculates a solution to the 2-D Laplace equation. The example was adapted from the FEVS benchmark suite [2].

The type for the Laplace solver example:

```
1         val maxIter : {x : positive | x > 0}.
2         val nx : {x : positive | x > 0}.
3         val ny : {x : positive | x > 0}.
4         ∀i ≤ size − 1.
5           (∀j ≤ ny − 2.
6             message i + 1 1 {x : float array | len(x) = nx});
7         message size 1 {x : float array | len(x) = nx});
8         ∀iter ≤ maxIter.
9           (∀i ≤ size − 1.
10            message (i + 1) i {x : float array | len(x) = 10});
11          (∀i ≤ size − 1.
12            message i (i + 1) {x : float array | len(x) = 10});
13          allreduce s : float.
14          skip
```

The code for the Laplace solver example:

```
1         let maxIter : int = val <read-max-iterations> in
2         let dimx : int = val <read-horizontal-dimension> in
3         let dimy : int = val <read-vertical-dimension> in
4         let lower : int = rank − 1 in
5         let upper : int = rank + 1 in
6         let time : int ref = mkref 0 in
7         let u1 : float array ref = mkref (mkfloatarray (dimy ∗ dimx)) in
8         let u2 : float array ref = mkref (mkfloatarray (dimy ∗ dimx)) in
9         if rank > 1 then
10          for prow : dimy − 2..1 do
11            send 1 !u1;
12          if rank = size
13            then send 1 !u1 else skip
14        else
15          (let rbuf : float array ref = mkref mkfloatarray dimx in
16          for proc : size − 1..1 do
17            for prow : dimy − 2..1 do
18              receive proc + 1 rbuf
19          receive size rbuf);
20        for iter : maxIter..1 do
21          (if rank = 0 then
```

```
22          receive upper u1;
23          send upper !u1
24        else
25          if rank = size then
26            send lower !u1 :
27            receive lower u1
28          else
29            send lower !u1;
30            receive upper u1;
31            receive lower u1;
32            send upper !u1);
33        let error: float = <calculate-error> in
34        let globalerror: float = allreduce error in
35        <update-time>;
36        <swap-u1-u2>
```

A.4 N-body simulation

The **N-body simulation** example simulates a dynamic system of particles under the influence of physical forces, particularly gravity, including the effect particles have on each other. Each process is responsible for a fixed subset of the particles. This example was adapted from the Using MPI book [7].

The type for the N-body simulation example:

```
1   val n: {x: nat | x%size = 0}.
2   val maxIter: {x: nat | x > 0}.
3   ∀iter ≤ maxIter.
4     (∀pipe ≤ size − 1.
5       ∀i ≤ size.
6         message i (i + 1 ≤ p ? i + 1 : 1) {x: float array | len(x) = n ∗ 4});
7     allreduce z: float.
8   skip
```

The code for the N-body simulation example:

```
1   let npart: int = val <get-problem-size> in
2   let maxIter: int = val <get-max-iterations> in
3   let left: int = if rank = 1 then size else rank − 1 in
4   let right: int = if rank = size then 1 else rank + 1 in
5   let simt: float ref = mkref 0 in
6   let count: int = npart ∗ 4 in
7   let out: float array = mkfloatarray count in
8   let in: float array ref = mkref (mkfloatarray count) in
9   for iter: maxIter..1 do
10    for pipe: size..1 do
11      if rank = 1 then
12        send right out;
13        receive left in
14      else
```

```
15              receive left in;
16              send right out
17          let dtlocal: float = 0.01 in
18          let dt: float = allreduce dtlocal in
19          <update-simt>
```

A.5 π calculation

The π **calculation** example calculates π through numerical integration. This example was adapted from the Using MPI book [7].

The type for the π calculation example:

```
1       ∀i ≤ size − 1.
2         message 1 (i + 1) {x: int | x > 1};
3       reduce 1
```

The code for the π calculation example:

```
1       let n: {x: int | x > 1} ref = mkref 2 in
2       let mypi: float ref = mkref 0.0 in
3       let pi: float ref = mkref 0.0 in
4       let sum: float ref = mkref 0.0 in
5       if rank = 1 then
6         n := <read-intervals>;
7         for i: p − 1..1 do
8           send (i + 1) !n
9       else
10        receive 1 n
11      let h: float = <calculate-h> in
12      sum := <calculate-sum>;
13      mypi := h/!sum;
14      reduce 1 !mypi pi
```

A.6 Vector dot product

The **vector dot product** example calculates the dot product of two vectors. This example was adapted from the Parallel Programming with MPI book [21].

The type for the Vector dot product example:

```
1       broadcast 0 v: {x: int | x > 0 ∧ x%size = 0}.
2       ∀i ≤ size − 1.message 1 (i + 1) float array;
3       ∀i ≤ size − 1.message 1 (i + 1) float array;
4       allreduce w: float.
5       ∀i ≤ size − 1.message (i + 1) 1 float array
```

The code for the vector dot product example:

```
1   let psize: int ref = mkref (if (rank = 1) then <initialize> else 0) in
2   let n: int = broadcast 1 !psize in
3   let localx: float array = mkfloatarray n in
4   let localy: float array = mkfloatarray n in
5   let temp: float array = mkfloatarray n in
6   let nbar: int = n/size in
7   if rank = 1 then
8     <scan-local-x>;
9     for i: size − 1..1 do
10      <scan-temp>;
11      send (i + 1) !temp
12  else
13    receive 1 localx;
14  if rank = 1 then
15    <scan-local-y>;
16    for i: size − 1..1 do
17      <scan-temp>;
18      send (i + 1) !temp
19  else
20    (let buf: float ref = mkref 0.0 in
21    receive 1 buf;
22    <update-localy>);
23  let localdot: float ref = <calculate-dot-product> in
24  let dot: float = allreduce !localdot in
25  let remotedot: float ref = mkref 0.0 in
26  if rank = 1 then
27    for i: size − 1..1 do
28      receive (i + 1) remotedot
29  else
30    send 1 !localdot
```

B Proofs

B.1 Results related to term types

Main results in this section: agreement for type formation (Lemma 1), weakening (Lemma 7), strengthening (Lemma 8), $<:$ is a preorder (Lemma 11), and the substitution lemma for types (Lemma 12).

Lemma 1 (agreement for type formation). Statement on page 8.

Proof. By simultaneous rule induction on the various hypotheses.

Lemma 7 (weakening). *Let* $\Gamma \vdash D : \mathbf{dtype}$.[4]

$$\frac{\Gamma \vdash T : \mathbf{type}}{\Gamma, x : D \vdash T : \mathbf{type}} \quad \frac{\Gamma \vdash D_2 : \mathbf{dtype}}{\Gamma, x\colon D \vdash D_2 : \mathbf{dtype}} \quad \frac{\Gamma \vdash p : \mathbf{prop}}{\Gamma, x : D \vdash p : \mathbf{prop}}$$

$$\frac{\Gamma \vdash i : D_1}{\Gamma, x\colon D \vdash i : D_1} \quad \frac{\Gamma \vdash D_1 <: D_2}{\Gamma, x\colon D \vdash D_1 <: D_2} \quad \frac{\Gamma \vdash p\ \mathbf{true}}{\Gamma, x : D \vdash p\ \mathbf{true}}$$

Proof. By simultaneous rule induction on the various hypotheses.

Lemma 8 (strengthening).

$$\frac{\Gamma, x\colon D \vdash T : \mathbf{type} \quad x \notin \mathrm{fv}(T)}{\Gamma \vdash T : \mathbf{type}} \quad \frac{\Gamma, x\colon D \vdash D_1 : \mathbf{dtype} \quad x \notin \mathrm{fv}(D_1)}{\Gamma \vdash D_1 : \mathbf{dtype}}$$

$$\frac{\Gamma, x\colon D \vdash p : \mathbf{prop} \quad x \notin \mathrm{fv}(p)}{\Gamma \vdash p : \mathbf{prop}} \quad \frac{\Gamma, x\colon D \vdash i : D_1 \quad x \notin \mathrm{fv}(D_1)}{\Gamma \vdash i : D_1}$$

$$\frac{\Gamma, x\colon D \vdash D_1 <: D_2 \quad x \notin \mathrm{fv}(D_1, D_2)}{\Gamma \vdash D_1 <: D_2} \quad \frac{\Gamma, x\colon D \vdash p\ \mathbf{true} \quad x \notin \mathrm{fv}(p)}{\Gamma \vdash p\ \mathbf{true}}$$

$$\frac{y : D_1 \in \Gamma, x : D_2 \quad x \notin y, \mathrm{fv}(D_1)}{y : D_1 \in \Gamma}$$

Proof. By simultaneous rule induction on the first hypothesis for each justified inference.

Lemma 9 (inversion for subtyping).

1. *If* $\Gamma \vdash \mathsf{int} <: D$ *then* D *is* int *or* D *is* $\{x\colon D' \mid p\}$ *and* $\Gamma \vdash D' <: \mathsf{int}$.
2. *If* $\Gamma \vdash \mathsf{float} <: D$ *then* D *is* float *or* D *is* $\{x\colon D' \mid p\}$ *and* $\Gamma \vdash D' <: \mathsf{float}$.
3. *If* $\Gamma \vdash D_1\,\mathsf{array} <: D_2$ *then* D_2 *is* $D_3\,\mathsf{array}$ *and* $\Gamma \vdash D_1 <: D_3$ *or* D_2 *is* $\{x\colon D_3 \mid p\}$ *and* $\Gamma \vdash D_1\,\mathsf{array} <: D_3$ *and* $\Gamma, x\colon D_1\,\mathsf{array} \vdash p\ \mathbf{true}$.

[4] Assertion $\Gamma \vdash D : \mathbf{dtype}$ should be understood as a premise for all justified inference rules. For example, the inference rule for types is the following.

$$\frac{\Gamma \vdash D : \mathbf{dtype} \quad \Gamma \vdash T : \mathbf{type}}{\Gamma, x : D \vdash T : \mathbf{type}}$$

4. *If* $\Gamma \vdash \{x\colon D_1 \mid p_1\} <\colon D_2$ *then* $\Gamma \vdash D_1 <\colon D_2$ *and* $\Gamma, x\colon D_1 \vdash p_1 : \mathbf{prop}$ *or* D_2 *is* $\{y\colon D_3 \mid p_2\}$ *and* $\Gamma \vdash \{x\colon D_1 \mid p_1\} <\colon D_3$ *and* $\Gamma, y : \{x\colon D_1 \mid p_1\} \vdash p_2\ \mathbf{true}$.

Proof. By a case analysis on the rules for subtyping.

Lemma 10 (context subsumption). *Let* $\Gamma \vdash D_2 <\colon D_1$.

$$\frac{\Gamma, x\colon D_1 : \mathbf{context}}{\Gamma, x\colon D_2 : \mathbf{context}} \quad \frac{\Gamma, x\colon D_1 \vdash T : \mathbf{type}}{\Gamma, x\colon D_2 \vdash T : \mathbf{type}} \quad \frac{\Gamma, x\colon D_1 \vdash D_3 : \mathbf{dtype}}{\Gamma, x\colon D_2 \vdash D_3 : \mathbf{dtype}}$$
$$\frac{\Gamma, x\colon D_1 \vdash p\ \mathbf{true}}{\Gamma, x\colon D_2 \vdash p\ \mathbf{true}} \quad \frac{\Gamma, x\colon D_1 \vdash p : \mathbf{prop}}{\Gamma, x\colon D_2 \vdash p : \mathbf{prop}} \quad \frac{\Gamma, x\colon D_1 \vdash i : D_3}{\Gamma, x\colon D_2 \vdash i : D_3}$$

Proof. By simultaneous rule induction on the various hypotheses.

Lemma 11 (<: is a pre-order).

$$\frac{}{\Gamma \vdash D <\colon D} \qquad \frac{\Gamma \vdash D_1 <\colon D_2 \quad \Gamma \vdash D_2 <\colon D_3}{\Gamma \vdash D_1 <\colon D_3}$$

Proof (sketch). Reflexivity follows by case analysis on assertion $\Gamma \vdash D : \mathbf{dtype}$. Transitivity is proved by induction on assertion $\Gamma \vdash D_1 : \mathbf{dtype}$ followed by induction on assertion $\Gamma \vdash D_2 : \mathbf{dtype}$, using the inversion lemma for subtyping (Lemma 9) and context subsumption (Lemma 10).

Lemma 12 (substitution lemma).

$$\frac{\Gamma_1, x\colon D, \Gamma_2 \vdash T : \mathbf{type} \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} \vdash T\{i/x\} : \mathbf{type}} \quad \frac{\Gamma_1, x\colon D, \Gamma_2 \vdash D_1 : \mathbf{dtype} \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} \vdash D_1\{i/x\} : \mathbf{dtype}}$$
$$\frac{\Gamma_1, x\colon D, \Gamma_2 \vdash p : \mathbf{prop} \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} \vdash p\{i/x\} : \mathbf{prop}} \quad \frac{\Gamma_1, x\colon D, \Gamma_2 \vdash p\ \mathbf{true} \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} \vdash p\{i/x\}\ \mathbf{true}}$$
$$\frac{\Gamma_1, x\colon D, \Gamma_2 \vdash i_1 : D_1 \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} \vdash i_1\{i/x\} : D_1\{i/x\}} \quad \frac{\Gamma_1, x\colon D, \Gamma_2 \vdash D_1 <\colon D_2 \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} \vdash D_1\{i/x\} <\colon D_2\{i/x\}}$$
$$\frac{\Gamma_1, x\colon D, \Gamma_2 : \mathbf{context} \quad \Gamma_1 \vdash i : D}{\Gamma_1, \Gamma_2\{i/x\} : \mathbf{context}} \quad \frac{y\colon D \in (\Gamma_1, x\colon D, \Gamma_2) \quad \Gamma_1 \vdash i : D}{y\colon D\{i/x\} \in \Gamma\{i/x\}}$$

Proof. By simultaneous rule induction on the first hypothesis of each justified inference rule. We highlight a couple of cases.

hypothesis	$\Gamma_1 \vdash i : D$	(1)

Case the derivation ends with $\Gamma_1, x\colon D, \Gamma_2 \vdash \mathsf{broadcast}\, i_1\, y\colon D_1.T : \mathbf{type}$.

rule premise	$\Gamma_1, x\colon D, \Gamma_2 \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true}$	(2)
rule premise	$\Gamma_1, x\colon D, \Gamma_2, y\colon D_1 \vdash T : \mathbf{type}$	(3)
1, induction, def. subs.	$\Gamma_1, \Gamma_2\{i/x\} \vdash 1 \le i_1\{i/x\} \le \mathsf{size}\ \mathbf{true}$	(4)
2, induction, def. subs.	$\Gamma_1, \Gamma_2\{i/x\}, y\colon D_1\{i/x\} \vdash T\{i/x\} : \mathbf{type}$	(5)

$4, 5,$ formation, def. subs. $\quad \Gamma_1, \Gamma_2\{i/x\} \vdash (\mathsf{broadcast}\, i_1\, y\colon D_1.T)\{i/x\} : \mathbf{type}$ (6)

Case the derivation ends with $\Gamma_1, x\colon D, \Gamma_2 \vdash y : D_1$.

rule premise	$\Gamma_1, x\colon D, \Gamma_2 : \mathbf{context}$	(2)
rule premise	$y\colon D_1 \in (\Gamma_1, x\colon D, \Gamma_2)$	(3)

Subcase $y = x$

$3, y = x$	D_1 is D	(4)
2, context formation	$x \notin D$	(5)
$1, 4, 5, y = x,$ def. subs.	$\Gamma_1 \vdash y\{i/x\} : D_1\{i/x\}$	(6)
6, 2, lemma 7	$\Gamma_1, \Gamma_2\{i/x\} \vdash y\{i/x\} : D_1\{i/x\}$	(7)

Subcase $y \neq x$

1, 2, induction	$\Gamma_1, \Gamma_2\{i/x\} : \mathbf{context}$	(8)
1, 3, induction	$y\colon D_1\{i/x\} \in (\Gamma_1, \Gamma_2\{i/x\})$	(9)
8, 9, type formation	$\Gamma_1, \Gamma_2\{i/x\} \vdash y\{i/x\} : D_1\{i/x\}$	(10)

Case the derivation ends with $\Gamma_1, x\colon D, \Gamma_2 \vdash i_1 : \{y\colon D_1 \mid p\}$.

rule premise	$\Gamma_1, x\colon D, \Gamma_2 \vdash i_1 : D_1$	(2)
rule premise	$\Gamma_1, x\colon D, \Gamma_2 \vdash p\{i_1/y\}\ \mathbf{true}$	(3)
2, induction	$\Gamma_1, \Gamma_2\{i/x\} \vdash i_1\{i/x\} : D_1\{i/x\}$	(4)
2, induction	$\Gamma_1, \Gamma_2\{i/x\} \vdash p\{i_1/y\}\{i/x\}\ \mathbf{true}$	(5)
$5, (y \neq x),$ def. subs.	$\Gamma_1, \Gamma_2\{i/x\} \vdash p\{i/x\}\{i_1/y\}\ \mathbf{true}$	(6)
$4, 5,$ index formation	$\Gamma_1, \Gamma_2\{i/x\} \vdash i_1\{i/x\} : \{y\colon D_1\{i/x\} \mid p\{i/x\}\}$	(7)
7, def. subs.	$\Gamma_1, \Gamma_2\{i/x\} \vdash i_1\{i/x\} : \{y\colon D_1 \mid p\}\{i/x\}$	(8)

Case the derivation ends with $\Gamma_1, x\colon D, \Gamma_2 \vdash p\ \mathbf{true}$.

rule premise	$\Gamma_1, x\colon D, \Gamma_2 \vdash p : \mathbf{prop}$	(2)
rule premise	$\mathrm{formulae}(\Gamma_1, x\colon D, \Gamma_2) \vDash p$	(3)
2, induction	$\Gamma_1, \Gamma_2\{i/x\} \vdash p\{i/x\} : \mathbf{prop}$	(4)
1, 3, assumption on $\vDash$	$\mathrm{formulae}(\Gamma_1, \Gamma_2\{i/x\}) \vDash p\{i/x\}$	(5)
$4, 5,$ **true** formation	$\Gamma_1, \Gamma_2\{i/x\} \vdash p\{i/x\}\ \mathbf{true}$	(6)

Lemma 13 (context exchange). *Let* $\Gamma_1 \vdash D_1 : \mathbf{dtype}$

$$\frac{\Gamma_1, \Gamma_2, x : D_1 \vdash T : \mathbf{type}}{\Gamma_1, x : D_1, \Gamma_2 \vdash T : \mathbf{type}} \quad \frac{\Gamma_1, \Gamma_2, x : D_1 \vdash D : \mathbf{dtype}}{\Gamma_1, x : D_1, \Gamma_2 \vdash D : \mathbf{dtype}} \quad \frac{\Gamma_1, \Gamma_2, x : D_1 \vdash p\ \mathbf{true}}{\Gamma_1, x : D_1, \Gamma_2 \vdash p\ \mathbf{true}}$$

$$\frac{\Gamma_1, \Gamma_2, x : D_1 \vdash p : \mathbf{prop}}{\Gamma_1, x : D_1, \Gamma_2 \vdash p : \mathbf{prop}} \quad \frac{\Gamma_1, \Gamma_2, x : D_1 \vdash i : D_2}{\Gamma_1, x : D_1, \Gamma_2 \vdash i : D_2} \quad \frac{\Gamma_1, \Gamma_2, x : D \vdash \mathbf{context}}{\Gamma_1, x : D, \Gamma_2 : \mathbf{context}}$$

$$\frac{\Gamma_1, \Gamma_2, x : D_1 \vdash D_2 <: D_3}{\Gamma_1, x : D_1, \Gamma_2 \vdash D_2 <: D_3}$$

Type equality, $\Gamma \vdash T \equiv T$

$$\frac{\Gamma \vdash i_1, i_2 = i_3, i_4 \ \mathbf{true} \quad \Gamma \vdash D_1 \equiv D_2 \quad (\Gamma \vdash 1 \leq i_1, i_2 \leq \mathsf{size} \wedge i_1 \neq i_2 \ \mathbf{true})}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D_1 \equiv \mathsf{message}\ i_3\ i_4\ D_2}$$

$$\frac{\Gamma \vdash i_1 = i_2 \ \mathbf{true} \quad \Gamma \vdash D_1 \equiv D_2 \quad (\Gamma \vdash 1 \leq i_1 \leq \mathsf{size} \ \mathbf{true}) \quad (\Gamma \vdash D_1 <: \{x\colon D_3\ \mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\})}{\Gamma \vdash \mathsf{scatter}\ i_1\ D_1 \equiv \mathsf{scatter}\ i_2\ D_2}$$

$$\frac{\text{see } \mathsf{scatter}}{\Gamma \vdash \mathsf{gather}\ i_1\ D_1 \equiv \mathsf{gather}\ i_2\ D_2} \qquad \frac{\Gamma \vdash i_1 = i_2 \ \mathbf{true} \quad (\Gamma \vdash 1 \leq i_1 \leq \mathsf{size} \ \mathbf{true})}{\Gamma \vdash \mathsf{reduce}\ i_1 \equiv \mathsf{reduce}\ i_2}$$

$$\frac{\Gamma \vdash i_1 = i_2 \ \mathbf{true} \quad \Gamma \vdash D_1 \equiv D_2 \quad \Gamma, x\colon D_1 \vdash T_1 \equiv T_2 \quad (\Gamma \vdash 1 \leq i_1 \leq \mathsf{size} \ \mathbf{true})}{\Gamma \vdash \mathsf{broadcast}\ i_1\ x\colon D_1.T_1 \equiv \mathsf{broadcast}\ i_2\ x\colon D_2.T_2}$$

$$\frac{\Gamma \vdash D_1 \equiv D_2 \quad \Gamma, x\colon D_1 \vdash T_1 \equiv T_2}{\Gamma \vdash \mathsf{val}\, x\colon D_1.T_1 \equiv \mathsf{val}\, x\colon D_2.T_2} \qquad \frac{\Gamma \vdash T_1 \equiv T_3 \quad \Gamma \vdash T_2 \equiv T_4}{\Gamma \vdash T_1; T_2 \equiv T_3; T_4}$$

$$\frac{\Gamma \vdash p_1 \leftrightarrow p_2 \ \mathbf{true} \quad \Gamma \vdash T_1 \equiv T_2 : \mathbf{type} \quad \Gamma \vdash T_1' \equiv T_2' : \mathbf{type}}{\Gamma \vdash p_1\,?\,T_1 : T_1' \equiv p_2\,?\,T_2 : T_2' : \mathbf{type}}$$

$$\frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{skip} \equiv \mathsf{skip}} \qquad \frac{\Gamma \vdash i_1 = i_2 \ \mathbf{true} \quad \Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i_1\} \vdash T_1 \equiv T_2}{\Gamma \vdash \forall x \leq i_1.T_1 \equiv \forall x \leq i_2.T_2}$$

$$\frac{\Gamma \vdash i \geq 1 \ \mathbf{true} \quad (\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type})}{\Gamma \vdash \forall x \leq i.T \equiv (T\{i/x\}; \forall x \leq i - 1.T)}$$

$$\frac{\Gamma \vdash i < 1 \ \mathbf{true} \quad (\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type})}{\Gamma \vdash \forall x \leq i.T \equiv \mathsf{skip}}$$

$$\frac{\Gamma \vdash i_1, i_2 \neq \mathsf{rank} \ \mathbf{true} \quad (\Gamma \vdash 1 \leq i_1, i_2 \leq \mathsf{size} \wedge i_1 \neq i_2 \ \mathbf{true}) \quad (\Gamma \vdash D : \mathbf{dtype})}{\Gamma \vdash \mathsf{message}\, i_1\, i_2\, D \equiv \mathsf{skip}}$$

$$\frac{(\Gamma \vdash T : \mathbf{type})}{\Gamma \vdash T; \mathsf{skip} \equiv T} \quad \frac{(\Gamma \vdash T : \mathbf{type})}{\Gamma \vdash \mathsf{skip}; T \equiv T} \quad \frac{(\Gamma \vdash T_1, T_2, T_3 : \mathbf{type})}{\Gamma \vdash (T_1; T_2); T_3 \equiv T_1; (T_2; T_3)} \quad \frac{\Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash T_2 \equiv T_1} \quad \frac{\Gamma \vdash T_1 \equiv T_2 \quad \Gamma \vdash T_2 \equiv T_3}{\Gamma \vdash T_1 \equiv T_3}$$

Datatype equality, $\Gamma \vdash D \equiv D$

$$\frac{\Gamma \vdash D_1 <: D_2 \quad \Gamma \vdash D_2 <: D_1}{\Gamma \vdash D_1 \equiv D_2}$$

Fig. 13. Type equality

Proof. By mutual rule induction on the various "main" hypotheses.

B.2 Results related to type equality

The complete definition of type equality is in Figure 13. Main results in this section: agreement for type equality (Lemma 14) and type equality is an equivalence relation (Lemma 15).

Lemma 14 (agreement for type equality).

$$\frac{\Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}$$

Proof. By rule induction on the derivation of the hypothesis. We must analyse fourteen cases. The congruence/equivalence cases are standard. We detail the other four.

Case $\Gamma \vdash \mathsf{skip}; T \equiv T$:

hyp. premise	$\Gamma \vdash T : \mathbf{type}$	(1)
1, agreement for type formation	$\Gamma : \mathbf{context}$	(2)
2, type formation	$\Gamma \vdash \mathsf{skip} : \mathbf{type}$	(3)
1, 3, type formation	$\Gamma \vdash \mathsf{skip}; T : \mathbf{type}$	(4)

Case $\Gamma \vdash \forall x \leq i.T \equiv \mathsf{skip}$:

hyp. premise	$\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type}$	(1)
hyp. premise	$\Gamma \vdash i < 1\ \mathbf{true}$	(2)
1, type formation	$\Gamma \vdash \forall x \leq i.T : \mathbf{type}$	(3)
2, agreement for **true**	$\Gamma : \mathbf{context}$	(4)
4, type formation	$\Gamma \vdash \mathsf{skip} : \mathbf{type}$	(5)

Case $\Gamma \vdash \mathsf{message}\, i_1\, i_2\, D \equiv \mathsf{skip}$:

hyp. premise	$\Gamma \vdash i_1, i_2 \neq \mathsf{rank}\ \mathbf{true}$	(1)
hyp. premise	$\Gamma \vdash 1 \leq i_1, i_2 \leq \mathsf{size} \wedge i_1 \neq i_2\ \mathbf{true}$	(2)
hyp. premise	$\Gamma \vdash D : \mathbf{dtype}$	(3)
2, 3, type formation	$\Gamma \vdash \mathsf{message}\, i_1\, i_2\, D : \mathbf{type}$	(4)
3, agreement for **dtype**	$\Gamma : \mathbf{context}$	(5)
5, type formation	$\Gamma \vdash \mathsf{skip} : \mathbf{type}$	(6)

Case $\Gamma \vdash \forall x \leq i.T \equiv (T\{i/x\}; \forall x \leq i - 1.T)$:

hyp. premise	$\Gamma \vdash i \geq 1\ \mathbf{true}$	(1)
hyp. premise	$\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type}$	(2)
1, inversion	$\Gamma \vdash i \geq 1 : \mathbf{prop}$	(3)
3, inversion	$\Gamma \vdash i : \mathsf{int}$	(4)
2, lemma 1	$\Gamma, x : \{y : \mathsf{int} \mid y \leq i\} : \mathbf{context}$	(5)
5, inversion	$\Gamma \vdash \{y : \mathsf{int} \mid y \leq i\} : \mathbf{dtype}$	(6)
6, inversion	$\Gamma, y : \mathsf{int} \vdash y \leq i : \mathbf{prop}$	(7)
7, lemma 1	$\Gamma, y\colon \mathsf{int} : \mathbf{context}$	(8)

It is always the case that an index term is smaller or equal to itself:

4, deducibility	$\Gamma \vdash i \leq i\ \mathbf{true}$	(9)
8, def. of subs.	$\Gamma \vdash (y \leq i)\{i/y\}\ \mathbf{true}$	(10)

By application of index formation rules and the substitution lemma (Lemma 12):

$$\begin{array}{ll} 4, 10, \text{index formation} & \Gamma \vdash i : \{y\colon \mathsf{int} \mid y \leq i\} \quad (11) \\ 11, 2, \text{lemma 12} & \Gamma \vdash T\{i/x\} : \mathbf{type} \quad (12) \end{array}$$

It is easy to prove that $\Gamma \vdash \{y\colon \mathsf{int} \mid y \leq i-1\} <: \{y\colon \mathsf{int} \mid y \leq i\}$ (it follows from 9, standard deducibility, and index formation).

$$\begin{array}{ll} 2, \text{lemma 10} & \Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i-1\} \vdash T : \mathbf{type} \quad (13) \\ 16, \text{type formation} & \Gamma \vdash \forall x \leq i-1.T : \mathbf{type} \quad (14) \end{array}$$

Finally, we can join assertions 11 and 13 by applying the type formation rule for sequential composition:

$$\begin{array}{ll} 11, 13, \mathbf{type} \text{ formation} & \Gamma \vdash T\{i/x\}; \forall x \leq i-1.T : \mathbf{type} \quad (15) \\ 1, 2, \mathbf{type} \text{ formation} & \Gamma \vdash \forall x \leq i.T : \mathbf{type} \quad (16) \end{array}$$

Lemma 15 (type equality is an equivalence relation).

Proof. Reflexivity follows from the congruence rules; symmetry and transitivity are built into the definition.

B.3 Results related to program types

Main results in this section: agreement for program type formation (Lemma 16) and the load lemma (Lemma 2).

Lemma 16 (agreement for program type formation).

$$\frac{T_1, \ldots, T_n : \mathbf{ptype}}{\Gamma^{n,1} \vdash T_1 : \mathbf{type} \quad \ldots \quad \Gamma^{n,n} \vdash T_n : \mathbf{type}}$$

Proof. By rule induction on the hypothesis. We illustrate a few cases.
Case the derivation ends with type equality:

$$\begin{array}{ll} \text{rule premise} & T_1, \ldots, T'_k, \ldots, T_n : \mathbf{ptype} \quad (1) \\ 1, \text{inversion} & T_1, \ldots, T_k, \ldots, T_n : \mathbf{ptype} \quad (2) \\ 1, \text{inversion} & \Gamma^{n,k} \vdash T_k \equiv T'_k \quad (3) \\ \text{lemma 14} & \Gamma^{n,k} \vdash T'_k : \mathbf{type} \quad (4) \end{array}$$

Case the derivation ends with message passing:

$$\begin{array}{ll} \text{rule premise} & \mathsf{skip}_1, \ldots, (\mathsf{message}\, l\, m\, D), \mathsf{skip}_{l+1}, \\ & \quad \ldots, \mathsf{skip}_{m-1}, (\mathsf{message}\, l\, m\, D), \ldots, \mathsf{skip}_n \quad (1) \\ 1, \text{inversion} & \Gamma^n \vdash l \neq m\ \mathbf{true} \quad (2) \\ 1, \text{inversion} & \Gamma^n \vdash D : \mathbf{dtype} \quad (3) \end{array}$$

3, lemma 1	Γ^n : **context**	(4)
4, **ptype** formation	$\Gamma^n \vdash \mathsf{skip} : \mathbf{type}$	(5)
1, def. of l	$\Gamma^n \vdash 1 \leq l \leq \mathsf{size}\ \mathbf{true}$	(6)
1, def. of m	$\Gamma^n \vdash 1 \leq m \leq \mathsf{size}\ \mathbf{true}$	(7)
2, 3, 6, 7, **type** formation	$\Gamma^n \vdash \mathsf{message}\, l\, m\, D : \mathbf{type}$	(8)

Case the derivation ends with broadcast:

hyp. premise	$\mathsf{broadcast}\, l\, x\colon D.T, \ldots, \mathsf{broadcast}\, l\, x\colon D.T : \mathbf{ptype}$	(1)
hyp. premise	$\Gamma^n \vdash 1 \leq l \leq n\ \mathbf{true}$	(2)
hyp. premise	$\Gamma^n, x\colon D \vdash T : \mathbf{type}$	(3)
2, replacement in Γ^n	$\Gamma^n \vdash 1 \leq l \leq \mathsf{size}\ \mathbf{true}$	(4)
3, 4, **type** formation	$\Gamma^n \vdash \mathsf{broadcast}\, l\, x\colon D.T : \mathbf{type}$	(5)

Lemma 2 (load lemma). Statement on page 12.

Proof. By rule induction on hypothesis $\Gamma^n \vdash T : \mathbf{type}$.
Case the derivation ends with the scatter rule.

rule premise	$\Gamma^n \vdash 1 \leq i \leq \mathsf{size}\ \mathbf{true}$	(1)
rule premise	$\Gamma^n \vdash D <: \{x\colon D'\ \mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}$	(2)
1, 2, **ptype** formation	$\mathsf{scatter}\, l\, D, \ldots, \mathsf{scatter}\, l\, D : \mathbf{ptype}$	(3)

Case the derivation ends with reduce, gather, broadcast, val, skip: as above.
Case the derivation ends with the rule for $T = \mathsf{message}\, i_1\, i_2\, D$.

rule premise	$\Gamma^n \vdash 1 \leq i_1, i_2 \leq \mathsf{size} \wedge i_1 \neq i_2\ \mathbf{true}$	(1)
rule premise	$\Gamma^n \vdash D : \mathbf{dtype}$	(2)
1, inversion(s)	$\Gamma^n \vdash i_1 : \mathsf{int} \quad \Gamma^n \vdash i_2 : \mathsf{int}$	(3)
1, assumption on **true**	$\Gamma^n \vdash i_1 = l\ \mathbf{true} \quad \Gamma^n \vdash i_2 = m\ \mathbf{true}$	(4)
4, lemma 7	$\Gamma^{n,k} \vdash i_1 = l\ \mathbf{true} \quad \Gamma^{n,k} \vdash i_2 = m\ \mathbf{true}$	(5)
1, 2, $(l, m \neq k)$, type eq.	$\Gamma^{n,k} \vdash \mathsf{message}\, l\, m\, D \equiv \mathsf{skip} : \mathbf{type}$	(6)
1, 2, 6, **ptype** formation	$\mathsf{skip}_1, \ldots, \mathsf{message}\, l\, m\, D, \mathsf{skip}_{l+1}, \ldots,$ $\mathsf{message}\, l\, m\, D, \mathsf{skip}_{m+1}, \ldots, \mathsf{skip}_n : \mathbf{ptype}$	(7)

Case the derivation ends with the rule for $T = T_1; T_2$.

rule premise	$\Gamma^n \vdash T_1 : \mathbf{type}$	(1)
rule premise	$\Gamma^n \vdash T_2 : \mathbf{type}$	(2)
1, induction	$T_1, \ldots, T_1 : \mathbf{ptype}$	(3)
2, induction	$T_2, \ldots, T_2 : \mathbf{ptype}$	(4)
3, 4, **ptype** formation	$T_1; T_2, \ldots, T_1; T_2 : \mathbf{ptype}$	(5)

Case the derivation ends with ifc: as above.
Case the derivation ends with $\forall x \leq i.T$:

rule premise	$\Gamma^n, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type}$	(1)
1, lemma 1	$\Gamma^n, x\colon \{y\colon \mathsf{int} \mid y \leq i\} : \mathbf{context}$	(2)
2, **context** formation	$\Gamma^n \vdash \{y\colon \mathsf{int} \mid y \leq i\} : \mathbf{dtype}$	(3)
3, inversion	$\Gamma^n, y : \mathsf{int} \vdash y \leq i : \mathbf{prop}$	(4)
4, inversion	$\Gamma^n, y : \mathsf{int} \vdash i : \mathsf{int}$	(5)
5, lemma 8	$\Gamma^n \vdash i : \mathsf{int}$	(6)

We analyse on the possible values of i:
Subcase $\Gamma^n \vdash i < 1$ **true**:

subcase	$\Gamma^n \vdash i < 1$ **true**	(7)
1, 7, type eq.	$\Gamma^n \vdash \forall x \leq i.T \equiv \mathsf{skip} : \mathbf{type}$	(8)
8, lemma 14	$\Gamma^n \vdash \mathsf{skip} : \mathbf{type}$	(9)
9, **ptype** formation	$\mathsf{skip}, \ldots, \mathsf{skip} : \mathbf{ptype}$	(10)

Subcase $\Gamma^n \vdash i \geq 1$ **true**:

subcase	$\Gamma^n \vdash i \geq 1$ **true**	(11)
1, 11, type eq.	$\Gamma^n \vdash \forall x \leq i.T \equiv T\{x/i\}; \forall x \leq i - 1.T : \mathbf{type}$	(12)
12, lemma 14	$\Gamma^n \vdash T\{x/i\}; \forall x \leq i - 1.T : \mathbf{type}$	(13)
13, inversion	$\Gamma^n \vdash T\{x/i\} : \mathbf{type}$	(14)
13, inversion	$\Gamma^n \vdash \forall x \leq i - 1.T : \mathbf{type}$	(15)
14, induction	$T\{x/i\}, \ldots, T\{x/i\} : \mathbf{ptype}$	(16)
15, induction	$\forall x \leq i - 1.T, \ldots, \forall x \leq i - 1.T : \mathbf{ptype}$	(17)
16, 17, **ptype** form.	$T\{x/i\}; \forall x \leq i - 1.T, \ldots, T\{x/i\}; \forall x \leq i - 1.T : \mathbf{ptype}$	(18)

B.4 Results related to references

Main results in this section: agreement for type formation, now with references (Lemma 17), index term subtyping remains a pre-order (Lemma 18) and that deducibility contains no references (Lemma 19).

Lemma 17 (agreement for type formation, with references). *The statement of this lemma is that of Lemma 1, page 8.*

Proof. By mutual rule induction on the hypotheses, reusing the cases from the proof of Lemma 1.

Lemma 18 (subtyping is still a pre-order). *The statement of this lemma is that of Lemma 11, page 34.*

Proof. By rule induction on the hypothesis, reusing the cases from the proof of Lemma 11. Inversion for $<:$ (Lemma 9) must be extended with the following case.

5. If $\Gamma \vdash D_1 \,\mathsf{ref} <: D_2$ then D_2 is $D_3 \,\mathsf{ref}$ and $\Gamma \vdash D_1 <: D_3$ or D_2 is $\{x\colon D_3 \mid p\}$ and $\Gamma \vdash D_1 \,\mathsf{ref} <: D_3$ and $\Gamma, x\colon D_1 \,\mathsf{ref} \vdash p$ **true**.

Lemma 19 (deducibility contains no references).

$$\frac{\Gamma \vdash p \textbf{ true}}{r \notin \mathrm{refs}(p)}$$

Proof. Assertion $\Gamma \vdash p$ **true** is not defined on reference identifiers.

B.5 Results related to expressions

Main results in this section: agreement for expression formation (Lemma 20), weakening for expression formation (Lemma 21), strengthening for expression formation (Lemma 22), substitution in expressions (Lemma 24), and the inversion lemma for expression formation (Lemma 26).

Lemma 20 (agreement for expression formation).

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash T : \textbf{type}}$$

Proof. By rule induction on the hypothesis, using Lemma 1.

Lemma 21 (weakening for expressions).

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash D : \textbf{dtype}}{\Gamma, x : D \vdash e : T}$$

Proof. By rule induction on the first hypothesis, using Lemma 7.

Lemma 22 (strengthening for expressions).

$$\frac{\Gamma, x\colon D \vdash e : T \quad x \notin \mathrm{fv}(e, T)}{\Gamma \vdash e : T}$$

Proof. By rule induction on the first hypothesis, using Lemma 8.

Lemma 23 (context subsumption for expressions).

$$\frac{\Gamma, x\colon D_1 \vdash e : T \quad \Gamma \vdash D_2 <: D_1}{\Gamma, x\colon D_2 \vdash e : T}$$

Proof. By rule induction on the first hypothesis, using Lemma 10.

Lemma 24 (substitution for expressions).

$$\frac{\Gamma, x\colon D \vdash e : T \quad \Gamma \vdash i : D}{\Gamma \vdash e\{i/x\} : T\{i/x\}}$$

Proof. By rule induction on the first hypothesis, using Lemma 12.

Lemma 25 (context exchange for expressions).

$$\frac{\Gamma_1, \Gamma_2, x : D_1 \vdash e : T \quad \Gamma_1 \vdash D_1 : \mathbf{dtype}}{\Gamma_1, x : D_1, \Gamma_2 \vdash e : T}$$

Proof. By rule induction on the first hypothesis, using Lemma 13.

The lemma below establishes the various expressions that may inhabit a given type. It forms the basis to the proof of the result on progress (Theorem 3).

Lemma 26 (inversion for expression formation).

1. *If* $\Gamma \vdash e : \mathsf{broadcast}\, i\, x\colon D.T$ *then* e *is either*
   - $\mathsf{let}\, x\colon D_1 = \mathsf{broadcast}\, i_1\, i_2\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash D_1 \equiv D : \mathbf{dtype}$ *and* $\Gamma \vdash i_1 = i\ \mathbf{true}$ *and* $\Gamma \vdash 1 \leq i_1 \leq \mathsf{size}\ \mathbf{true}$ *and* $\Gamma \vdash i_2 : D$ *and* $\mathsf{rank} \notin \mathrm{fv}(i_1)$ *and* $\Gamma, x\colon D \vdash e_1 : T$, *or*
   - $\mathsf{let}\, y\colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{broadcast}\, i\, x\colon D.T)$ *and* $\Gamma, y\colon D_1 \vdash e_1 : \mathsf{broadcast}\, i\, x\colon D.T$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{broadcast}\, i\, x\colon D.T$ *and* $\Gamma \vdash e_2 : \mathsf{broadcast}\, i\, x\colon D.T$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{broadcast}\, i\, x\colon D.T$.
2. *If* $\Gamma \vdash e : \mathsf{message}\, i\, i'\, D$ *then* e *is either*
   - $\mathsf{send}\, i_1\, i_2$ *and* $\Gamma \vdash \mathsf{rank} = i\ \mathbf{true}$ *and* $\Gamma \vdash i_1 = i'\ \mathbf{true}$ *and* $\Gamma \vdash i_2 : D$, *or*
   - $\mathsf{receive}\, i_1\, i_2$ *and* $\Gamma \vdash i_1 = i\ \mathbf{true}$ *and* $\Gamma \vdash \mathsf{rank} = i'\ \mathbf{true}$, *and* $\Gamma \vdash i_2 : D\ \mathsf{ref}$, *or*
   - $\mathsf{let}\, y\colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{message}\, i\, i'\, D)$ *and* $\Gamma, y\colon D_1 \vdash e_1 : \mathsf{message}\, i\, i'\, D$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{message}\, i\, i'\, D$ *and* $\Gamma \vdash e_2 : \mathsf{message}\, i\, i'\, D$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{message}\, i\, i'\, D$.
3. *If* $\Gamma \vdash e : \forall x \leq i.T$ *then* e *is either*
   - $\mathsf{for}\, x\colon i_1..1\, \mathsf{do}\, e_1$ *and* $\Gamma \vdash i_1 \leq i\ \mathbf{true}$ *and* $\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash e_1 : T$ *and* $\Gamma \vdash T\{i/x\} = T\{i-1/x\} = \cdots = \{i_1+1/x\} \equiv \mathsf{skip} : \mathbf{type}$, *or*
   - skip *and* $\Gamma \vdash i < 1\ \mathbf{true}$, *or*
   - $\mathsf{let}\, y\colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\forall x \leq i.T)$ *and* $\Gamma, y\colon D_1 \vdash e_1 : \forall x \leq i.T$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \forall x \leq i.T$ *and* $\Gamma \vdash e_2 : \forall x \leq i.T$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \forall x \leq i.T$, *or*
4. *If* $\Gamma \vdash e : (T; T')$ *then* e *is either*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : T$ *and* $\Gamma \vdash e_2 : T'$
   - $\mathsf{for}\, x\colon i_1..1\, \mathsf{do}\, e_1$ *and* $\Gamma \vdash i_1 \geq 1\ \mathbf{true}$ *and* $\Gamma, x\colon \{y\colon \mathsf{int} \mid y \leq i_1\} \vdash e_1 : T_1$ *and* $\Gamma \vdash T_1\{i/x\} \equiv T : \mathbf{type}$ *and* $\Gamma \vdash (\forall x \leq i_1 - 1.T_1) \equiv T' : \mathbf{type}$, *or*
   - $\mathsf{let}\, y\colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(T; T')$ *and* $\Gamma, y\colon D_1 \vdash e_1 : T; T'$, *or*

- $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : T; T'$ *and* $\Gamma \vdash e_2 : T; T'$, *or*
- $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : T; T'$, *or*

5. *If* $\Gamma \vdash e : \mathsf{skip}$ *then* e *is either*
   - skip, *or*
   - $\mathsf{while}\, p\, \mathsf{do}\, e_1$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$, *or*
   - $\mathsf{for}\, x \colon i..1\, \mathsf{do}\, e_1$ *and* $\Gamma \vdash i < 1\ \mathbf{true}$ *and* $\Gamma, x \colon \{y \colon \mathsf{int} \mid y \leq i\} \vdash e_1 : \mathsf{skip}$, *or*
   - $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{skip})$ *and* $\Gamma, y \colon D_1 \vdash e_1 : \mathsf{skip}$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{skip}$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{skip}$.
6. *If* $\Gamma \vdash e : \mathsf{gather}\, i\, D$ *then* e *is either*
   - $\mathsf{gather}\ i_1\ i_2\ i_3$ *and* $\Gamma \vdash i_1 = i\ \mathbf{true}$ *and* $\Gamma \vdash 1 \leq i_1 \leq \mathsf{size}\ \mathbf{true}$ *and* $\Gamma \vdash i_2 : \{x \colon D\, \mathsf{array} \mid \mathsf{len}(i_3) = \mathsf{size} * \mathsf{len}(x)\}$ *and* $\Gamma \vdash i_3 : D\, \mathsf{array\ ref}$, *or*
   - $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{gather}\, i\, D)$ *and* $\Gamma, y \colon D_1 \vdash e_1 : \mathsf{gather}\, i\, D$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{gather}\, i\, D$ *and* $\Gamma \vdash e_2 : \mathsf{gather}\, i\, D$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{gather}\, i\, D$.
7. *If* $\Gamma \vdash e : \mathsf{scatter}\, i\, D$ *then* e *is either*
   - $\mathsf{scatter}\ i_1\ i_2\ i_3$ *and* $\Gamma \vdash i_1 = i\ \mathbf{true}$ *and* $\Gamma \vdash 1 \leq i_1 \leq \mathsf{size}\ \mathbf{true}$ *and* $\Gamma \vdash i_2 : \{x \colon D\, \mathsf{array} \mid \mathsf{len}(i_3) = \mathsf{size} * \mathsf{len}(x)\}$ *and* $\Gamma \vdash i_3 : D\, \mathsf{array\ ref}$, *or*
   - $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{scatter}\, i\, D)$ *and* $\Gamma, y \colon D_1 \vdash e_1 : \mathsf{scatter}\, i\, D$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{scatter}\, i\, D$ *and* $\Gamma \vdash e_2 : \mathsf{scatter}\, i\, D$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{scatter}\, i\, D$.
8. *If* $\Gamma \vdash e : \mathsf{reduce}\, i$ *then* e *is either*
   - $\mathsf{reduce}\ i_1\ i_2\ i_3$ *and* $\Gamma \vdash i_1 = i\ \mathbf{true}$ *and* $\Gamma \vdash 1 \leq i_1 \leq \mathsf{size}\ \mathbf{true}$ *and* $\Gamma \vdash i_2 : \mathsf{float}$ *and* $\Gamma \vdash i_3 : \mathsf{float\ ref}$, *or*
   - $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{reduce}\, i_1)$ *and* $\Gamma, y \colon D_1 \vdash e_1 : \mathsf{reduce}\, i_1$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{reduce}\, i_1$ *and* $\Gamma \vdash e_2 : \mathsf{reduce}\, i_1$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{reduce}\, i_1$.
9. *If* $\Gamma \vdash e : \mathsf{val}\, x \colon D.T$ *then* e *is either*
   - $\mathsf{let}\, x \colon D_1 = \mathsf{val}\, i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash D_1 \equiv D : \mathbf{dtype}$ *and* $\Gamma \vdash i_1 : D$ *and* $\Gamma, x \colon D \vdash e_1 : T$
   - $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(\mathsf{val}\, x \colon D.T)$ *and* $\Gamma, y \colon D_1 \vdash e_1 : \mathsf{val}\, x \colon D.T$, *or*
   - $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $\Gamma \vdash p : \mathbf{prop}$ *and* $\Gamma \vdash e_1 : \mathsf{val}\, x \colon D.T$ *and* $\Gamma \vdash e_2 : \mathsf{val}\, x \colon D.T$, *or*
   - $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : \mathsf{val}\, x \colon D.T$.
10. *If* $\Gamma \vdash e : p\,?\,T : T'$ *then* e *is either*

- $\mathsf{if}\, p' \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$ *and* $\Gamma \vdash p \leftrightarrow p'$ **true** *and* $\Gamma \vdash e_1 : T$ *and* $\Gamma \vdash e_2 : T'$ *and* $\mathsf{rank} \notin \mathrm{fv}(p)$*, or*
- $\mathsf{let}\, y\colon D_1 = i_1 \,\mathsf{in}\, e_1$ *and* $\Gamma \vdash i_1 : D_1$ *and* $y \notin \mathrm{fv}(p\,?\,T\!:\!T')$ *and* $\Gamma, y\colon D_1 \vdash e_1 : p\,?\,T\!:\!T'$*, or*
- $\mathsf{if}\, p \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$ *and* $\Gamma \vdash p :$ **prop** *and* $\Gamma \vdash e_1 : p\,?\,T\!:\!T'$ *and* $\Gamma \vdash e_2 : p\,?\,T\!:\!T'$*, or*
- $e_1; e_2$ *and* $\Gamma \vdash e_1 : \mathsf{skip}$ *and* $\Gamma \vdash e_2 : p\,?\,T\!:\!T'$.

Proof. By a case analysis on the rules for expression formation.

B.6 Results related to stores

Main results in this section: agreement for store-to-context formation (Lemma 27), agreement for index evaluation (Lemma 32) and that evaluation induces deducibility (Lemma 33).

Lemma 27 (agreement for store-to-context conversion).

$$\frac{\rho \mathit{\ to\ } \Gamma}{\rho : \mathbf{store} \quad \Gamma : \mathbf{context}}$$

Proof. By rule induction on the hypothesis.

Lemma 28 (store-to-context inversion).

$$\frac{\rho : \mathbf{store}}{\rho \mathit{\ to\ } \Gamma}$$

Proof. By rule induction on the hypothesis.

Lemma 29 (store update does not affect store-to-context).

$$\frac{\rho \mathit{\ to\ } \Gamma \quad \Gamma \vdash r : D\ \mathsf{ref} \quad \Gamma \vdash v : D}{\rho[r := v] \mathit{\ to\ } \Gamma}$$

Proof. By rule induction on the hypothesis $\rho \mathit{\ to\ } \Gamma$.

Lemma 30 (store value is index).

$$\frac{\rho \vdash r : D\,\mathsf{ref} \quad r := v \in \rho}{\rho \vdash v : D}$$

Proof. From the second hypothesis we know that ρ is not empty. Case ρ is $\rho', r := v$. From the definition of $\rho \mathit{\ to\ } \Gamma$ we know that $(\rho', r := v) \mathit{\ to\ } (\Gamma', r : D\,\mathsf{ref})$. By inversion we get $\Gamma' \vdash v : D$. Applying weakening (Lemma 7) we conclude $\Gamma', r : D\ \mathsf{ref} \vdash v : D$. Case ρ is $\rho', r' := v'$ with $r \neq r'$ follows by induction.

Lemma 31 (inversion of store formation).

$$\frac{\rho \mathit{\ to\ } \Gamma \quad r\colon D \in \Gamma}{r := v \in \rho \quad D \mathit{\ is\ } D'\,\mathsf{ref} \quad \Gamma \vdash v : D'}$$

Proof. By rule induction on hypothesis $\rho \; to \; \Gamma$. From the second premise we know that Γ is not empty, then the only case to consider is when ρ is not empty.
Case ρ is $\rho', r \; := \; v'$:

store as context	$\rho', r \; := \; v' \; to \; \Gamma', r : D' \; \mathsf{ref}$	(1)
1, inversion	$\rho' \; to \; \Gamma'$	(2)
1, inversion	$\Gamma' \vdash v' : D'$	(3)
1, inversion	$r \notin \rho', \Gamma, D'$	(4)
1	$r \; := \; v' \in \rho', r \; := \; v'$	(5)
3, lemma 7	$\Gamma', r : D' \; \mathsf{ref} \vdash v' : D'$	(6)

Case ρ is $\rho', r' \; := \; v'$ and $r \neq r'$:

Case hyp., store as context	$\rho', r' \; := \; v' \; to \; \Gamma', r' : D' \; \mathsf{ref}$	(1)
Case hyp.	$r \neq r'$	(2)
Case hyp.	$r : D \in \Gamma', r' : D' \; \mathsf{ref}$	(3)
2, 3, lemma 8	$r : D \in \Gamma'$	(4)
1, inversion	$r : D \in \Gamma'$	(5)
4, 5, induction	$r := v'' \in \rho'$	(6)
4, 5, induction	$D \; \mathsf{is} \; D'' \; \mathsf{ref}$	(7)
4, 5, induction	$\Gamma' \vdash v : D''$	(8)

Lemma 32 (agreement for evaluation).

$$\frac{(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D}{\rho_1 \; to \; \Gamma_1 \quad \rho_2 \; to \; \Gamma_2 \quad \Gamma_2 \; is \; \Gamma_1, \Gamma_3 \quad \Gamma^{n,k}, \Gamma_2 \vdash i : D \quad \Gamma_2 \vdash v : D}$$

Proof. By rule induction on the hypothesis. We list the most representative cases.
Case $(\rho, f)\downarrow^{n,k}(\rho, f) : \mathsf{float}$

rule premise	$\rho : \mathbf{store}$	(1)
1, lemma 28	$\rho \; to \; \Gamma$	(2)
2, lemma 27	$\Gamma : \mathbf{context}$	(3)
3, index formation	$\Gamma \vdash f : \mathsf{float}$	(4)
4, lemmas 7, 13	$\Gamma^{n,k}, \Gamma \vdash f : \mathsf{float}$	(5)

Case $(\rho, \mathsf{size})\downarrow^{n,k}(\rho, n) : \mathsf{int}$

rule premise	$\rho : \mathbf{store}$	(1)
1, lemma 28	$\rho \; to \; \Gamma$	(2)
2, lemma 27	$\Gamma : \mathbf{context}$	(3)
3, lemmas 7,13	$\Gamma^{n,k}, \Gamma : \mathbf{context}$	(4)

4, index formation	$\Gamma^{n,k}, \Gamma \vdash \mathsf{size} : \mathsf{int}$	(5)
2, index formation	$\Gamma \vdash n : \mathsf{int}$	(6)

Case $(\rho, r)\downarrow^{n,k}(\rho, r)\colon D\,\mathsf{ref}$

rule premise	$\rho : \mathbf{store}$	(1)
rule premise	$r := v \in \rho$	(2)
rule premise	$\rho \vdash v : D$	(3)

From (2) we know that ρ is non empty. Subcase ρ is $\rho', r := v$:

1, lemma 28	$(\rho', r := v)\ \mathit{to}\ (\Gamma', r : D\ \mathsf{ref})$	(4)
4, lemma 27	$\Gamma', r : D\ \mathsf{ref} : \mathbf{context}$	(5)
5, index formation	$\Gamma', r : D\ \mathsf{ref} \vdash r : D\ \mathsf{ref}$	(6)
6, lemmas 7,13	$\Gamma^{n,k}, \Gamma', r : D\ \mathsf{ref} \vdash r : D\ \mathsf{ref}$	(7)

The subcase where ρ is $\rho', r' := v'$ with $r \neq r'$ follows by induction.

Case $(\rho_1, !i)\downarrow^{n,k}(\rho_2, v)\colon D$

rule premise	$(\rho_1, i)\downarrow^{n,k}(\rho_2, r)\colon D\,\mathsf{ref}$	(1)
rule premise	$r := v \in \rho_2$	(2)
1, induction	$\rho_2\ \mathit{to}\ \Gamma_2$	(3)
1, induction	$\rho_1\ \mathit{to}\ \Gamma_1$	(4)
1, induction	Γ_2 is Γ_1, Γ_3	(5)
1, induction	$\Gamma^{n,k}, \Gamma_2 \vdash i : D\,\mathsf{ref}$	(6)
1, induction	$\Gamma_2 \vdash r : D\,\mathsf{ref}$	(7)
6, index formation	$\Gamma^{n,k}, \Gamma_2 \vdash !i : D$	(8)
2, 3, 7, lemma 30	$\Gamma_2 \vdash v : D$	(9)

Case $(\rho_1, i_1 := i_2)\downarrow^{n,k}(\rho_2[r := v], v)\colon D$

rule premise	$(\rho_1, i_1)\downarrow^{n,k}(\rho_3, r)\colon D\,\mathsf{ref}$	(1)
rule premise	$(\rho_3, i_2)\downarrow^{n,k}(\rho_2, v)\colon D$	(2)
1, induction	$\rho_1\ \mathit{to}\ \Gamma_1$	(3)
1, induction	$\Gamma^{n,k}, \Gamma_3 \vdash i_1 : D\,\mathsf{ref}$	(4)
1, induction	$\Gamma_3 \vdash r : D\,\mathsf{ref}$	(5)
1, induction	$\rho_3\ \mathit{to}\ \Gamma_3$	(6)
1, induction	Γ_3 is Γ_1, Γ_4	(7)
2, induction	$\Gamma^{n,k}, \Gamma_2 \vdash i_2 : D$	(8)
2, induction	$\Gamma_2 \vdash v : D$	(9)

2, induction	Γ_2 is Γ_3, Γ_5	(10)
2, induction	ρ_2 *to* Γ_2	(11)

Through simple context manipulation and the fact that store update does not affect store to context formation (Lemma 29) we obtain the remaining conclusions:

$7, 10$, equals for equals	Γ_2 is $\Gamma_1, \Gamma_4, \Gamma_5$	(12)
$4, 12$, lemma 7	$\Gamma^{n,k}, \Gamma_2 \vdash i_1 : D\ \mathsf{ref}$	(13)
$8, 13$, index formation	$\Gamma^{n,k}, \Gamma_2 \vdash i_1 := i_2 : D$	(14)
$5, 10$, lemma 7	$\Gamma_2 \vdash r : D\ \mathsf{ref}$	(15)
$9, 11, 15$, lemma 29	$\rho_2[r := v]$ *to* Γ_2	(16)

Case $(\rho_1, \mathsf{mkref}\, i)\downarrow^{n,k}((\rho_2, r := v), r) : D\ \mathsf{ref}$

rule premise	$(\rho_1, i)\downarrow^{n,k}(\rho_2, v) : D$	(1)
rule premise	r fresh	(2)
1, induction	$\Gamma^{n,k}, \Gamma_2 \vdash i : D$	(3)
1, induction	$\Gamma_2 \vdash v : D$	(4)
1, induction	Γ_2 is Γ_1, Γ_3	(5)
1, induction	ρ_1 *to* Γ_1	(6)
1, induction	ρ_2 *to* Γ_2	(7)
3, index formation	$\Gamma^{n,k}, \Gamma_2 \vdash \mathsf{mkref}\ i : D$	(8)
7, lemma 27	$\rho_2 :$ **store**	(9)
7, lemma 27	$\Gamma_2 :$ **context**	(10)
$2, 10, 7, 4$, store formation	$\rho_2, r := v :$ **store**	(11)
11, store as context formation	$\rho_2, r := v$ *to* $\Gamma_2, r : D\ \mathsf{ref}$	(12)
5, monotonicity	$\Gamma_2, r := v$ is $\Gamma_1, \Gamma_3, r := v$	(13)

Case $(\rho_1, [i_1, \ldots, i_n])\downarrow^{n,k}(\rho_{n+1}, [v_1, \ldots, v_n]) : D\ \mathsf{array}$

rule premise	$(\rho_1, i_1)\downarrow^{n,k}(\rho_2, v_1) : D$	(1)
rule premise	$(\rho_n, i_n)\downarrow^{n,k}(\rho_{n+1}, v_n) : D$	(2)
2, induction, for all $1 \leq l \leq n$	$\Gamma^{n,k}, \Gamma_{l+1} \vdash i_l : D$	(3)
2, induction, for all $1 \leq l \leq n$	$\Gamma_{l+1} \vdash v_l : D$	(4)
2, induction, for all $2 \leq l \leq n$	Γ_{l+1} is Γ_l, Γ'_l	(5)
2, induction, for all $1 \leq l \leq n+1$	ρ_l *to* Γ_l	(6)

By applying basic lemmas over contexts (weakening, strengthening), index formation rules, and transitivity we get:

5, transitivity	$\Gamma_{l+1} = \Gamma_1, \Gamma'_1, \ldots, \Gamma'_n$	(7)

$3, 7,$ lemma 7	$\Gamma^{n,k}, \Gamma_{l+1} \vdash i_l : D$	(8)
8, index formation	$\Gamma^{n,k}, \Gamma_{l+1} \vdash [i_1, \ldots, i_n] : D$ array	(9)
$4, 7,$ lemma 7	$\Gamma_{l+1} \vdash v_l : D$	(10)
11, index formation	$\Gamma_{l+1} \vdash [v_1, \ldots, v_n] : D$ array	(11)

Case $(\rho_1, i_1[i_2]) \downarrow^{n,k} (\rho_3, v_m) \colon D$

rule premise	$(\rho_1, i_1) \downarrow^{n,k} (\rho_2, [v_1, \ldots, v_l]) \colon \{x \colon D \text{ array} \mid \mathsf{len}(x) = l\}$	(1)
rule premise	$(\rho_2, i_2) \downarrow^{n,k} (\rho_3, m) \colon \{x \colon \mathsf{int} \mid 1 \leq x \leq l\}$	(2)
1, induction	$\Gamma^{n,k}, \Gamma_2 \vdash i_1 : \{D \text{ array} \mid \mathsf{len}(x) = l\}$	(3)
1, induction	$\Gamma_2 \vdash [v_1, \ldots, v_l] : \{D \text{ array} \mid \mathsf{len}(x) = l\}$	(4)
1, induction	ρ_1 *to* Γ_1	(5)
1, induction	$\Gamma_2 = \Gamma_1, \Gamma_4$	(6)
1, induction	ρ_2 *to* Γ_2	(7)
2, induction	$\Gamma^{n,k}, \Gamma_3 \vdash i_2 : \{x \colon \mathsf{int} \mid 1 \leq x \leq l\}$	(8)
2, induction	$\Gamma_3 \vdash m : \{x \colon \mathsf{int} \mid 1 \leq x \leq l\}$	(9)
2, induction	ρ_3 *to* Γ_3	(10)
2, induction	$\Gamma_3 = \Gamma_2, \Gamma_5$	(11)
$10, 15,$ equals for equals	Γ_3 is $\Gamma_1, \Gamma_4, \Gamma_5$	(12)
8, inversion, lemma 7	$\Gamma^{n,k}, \Gamma_3 \vdash 1 \leq i_1 \leq l$ **true**	(13)
$3, 13,$ index formation	$\Gamma^{n,k}, \Gamma_3 \vdash i_1[i_2] : D$	(14)
8, inversion, lemma 7	$\Gamma_3 \vdash v_m : D$	(15)

Case refinement introduction.

rule premise	$(\rho_1, i) \downarrow^{n,k} (\rho_2, v) \colon D$	(1)
rule premise	$\varepsilon \vdash p\{i/x\}$ **true**	(2)
1, induction	ρ_1 *to* Γ_1	(3)
1, induction	ρ_2 *to* Γ_2	(4)
1, induction	Γ_2 is Γ_1, Γ_3	(5)
1, induction	$\Gamma^{n,k}, \Gamma_2 \vdash i : D$	(6)
1, induction	$\Gamma_2 \vdash v : D$	(7)
2, lemma 7	$\Gamma^{n,k}, \Gamma_2 \vdash p\{i/x\}$ **true**	(8)
$6, 8,$ index formation	$\Gamma^{n,k}, \Gamma_2 \vdash i : \{x \colon D \mid p\}$	(9)
2, lemma 7	$\Gamma_2 \vdash p\{i/x\}$ **true**	(10)
$7, 10,$ index formation	$\Gamma_2 \vdash v : \{x \colon D \mid p\}$	(11)

Case subtyping.

rule premise	$(\rho_1, i) \downarrow^{n,k} (\rho_2, v) \colon D_1$	(1)

rule premise	$\rho_2 \vdash D_2 <: D_1$	(2)
1, induction	ρ_1 *to* Γ_1	(3)
1, induction	ρ_2 *to* Γ_2	(4)
1, induction	Γ_1 is Γ_1, Γ_3	(5)
1, induction	$\Gamma^{n,k}, \Gamma_2 \vdash i : D_1$	(6)
1, induction	$\Gamma_2 \vdash v : D_1$	(7)
2, 6, lemma 7, subtyping	$\Gamma^{n,k}, \Gamma_2 \vdash i : D_2$	(8)
2, 7, subtyping	$\Gamma_2 \vdash v : D_2$	(9)

Lemma 33 (eval to deducibility).

$$\frac{i \downarrow^n m}{\Gamma^n \vdash i = m \ \mathbf{true}}$$

Proof. By rule induction on the hypothesis, making use of basic assumptions on deducibility such as, $\Gamma \vdash i_1 = m_1$ **true** and $\Gamma \vdash i_2 = m_2$ **true** and m is $m_1 + m_2$ implies $\Gamma \vdash i_1 + i_2 = m$ **true**.

Lemma 3 (evaluation succeeds). Statement on page 18.

Proof. By rule induction on the hypothesis. We highlight a few representative cases.

premise	ρ *to* Γ	(1)
1, lemma 27	ρ : **store** $\quad \Gamma$: **context**	(2)

Case $\Gamma^{n,k}, \Gamma \vdash x : D$

premise	$x \colon D \in (\Gamma^{n,k}, \Gamma)$	(3)
1, 3	x is size or x is rank	(4)

Subcase x is size

1, eval rule	$(\rho, \mathsf{size}) \downarrow^{n,k} (\rho, n) \colon \mathsf{int}$	(5)

Subcase x is rank. As above.

Case $\Gamma^{n,k}, \Gamma \vdash r : D$

premise	$r \colon D \in (\Gamma^{n,k}, \Gamma)$	(3)
1, 3, lemma 31	D is $D'\mathsf{ref} \wedge \Gamma^{n,k}, \Gamma \vdash v : D' \wedge r := v \in \rho$	(4)
2, 4, eval rule	$(\rho, r) \downarrow^{n,k} (\rho, r) \colon D$	(5)

Case $\Gamma^{n,k}, \Gamma \vdash i_1 + i_2 : \mathsf{int}$

premise	$\Gamma^{n,k}, \Gamma \vdash i_1 : \mathsf{int}$	(3)

premise	$\Gamma^{n,k}, \Gamma \vdash i_2 : \mathsf{int}$	(4)
3, induction	$(\rho, i_1)\downarrow^{n,k}(\rho_1, m_1) \colon \mathsf{int}$	(5)
5, lemma 32	ρ_1 *to* Γ_1	(6)
5, lemma 32	Γ_1 is Γ, Γ'	(7)
4, 6, lemma 7	$\Gamma^{n,k}, \Gamma_1 \vdash i_2 : \mathsf{int}$	(8)
9, induction	$(\rho_1, i_2)\downarrow^{n,k}(\rho_2, m_2) \colon \mathsf{int}$	(9)
5, 9, eval rule	$(\rho, i_1 + i_2)\downarrow^{n,k}(\rho_2, v_1 + v_2) \colon \mathsf{int}$	(10)

Case $\Gamma^{n,k}, \Gamma \vdash i_1[i_2] : D$

premise	$\Gamma^{n,k}, \Gamma \vdash i_1 : \{x : D\ \mathsf{array} \mid \mathsf{len}(x) = l\}$	(3)
premise	$\Gamma^{n,k}, \Gamma \vdash 1 \leq i_2 \leq l$ **true**	(4)
4, inversion	$\Gamma^{n,k}, \Gamma \vdash i_2 : \mathsf{int}$	(5)
4, 5, index formation	$\Gamma^{n,k}, \Gamma \vdash i_2 : \{y : \mathsf{int} \mid 1 \leq y \leq l\}$	(6)
3, induction	$(\rho_0, i_1)\downarrow^{n,k}(\rho_1, v_1) \colon \{x : D\ \mathsf{array} \mid \mathsf{len}(x) = l\}$	(7)
7, lemma 32	ρ_1 *to* Γ_1	(8)
7, lemma 32	Γ_1 is Γ, Γ'	(9)
6, 9, lemma 7	$\Gamma^{n,k}, \Gamma_1 \vdash i_2 : \{y : \mathsf{int} \mid 1 \leq y \leq l\}$	(10)
10, induction	$(\rho_1, i_2)\downarrow^{n,k}(\rho_2, m) \colon \{y : \mathsf{int} \mid 1 \leq y \leq l\}$	(11)
7, 11, evaluation	$(\rho, i_1[i_2])\downarrow^{n,k}(\rho_2, v_m) \colon D$	(12)

Case Subtyping $\Gamma^{n,k}, \Gamma \vdash i : D_1$

premise	$\Gamma^{n,k}, \Gamma \vdash i : D_2$	(3)
premise	$\Gamma^{n,k}, \Gamma \vdash D_1 <: D_2$	(4)
3, induction	$(\rho, i)\downarrow^{n,k}(\rho', v) \colon D_2$	(5)
5, lemma 32	ρ' *to* Γ_2	(6)
5, lemma 32	Γ_2 is Γ, Γ'	(7)
4, 7, lem. 7	$\Gamma^{n,k}, \Gamma_2 \vdash D_1 <: D_2$	(8)
5, 8, evaluation	$(\rho, i)\downarrow^{n,k}(\rho', v) \colon D_1$	(9)

Case Refinement $\Gamma^{n,k}, \Gamma \vdash i : \{x : D \mid p\}$

premise	$\Gamma^{n,k}, \Gamma \vdash i : D$	(3)
premise	$\Gamma^{n,k}, \Gamma \vdash p\{i/x\}$ **true**	(4)
3, induction	$(\rho, i)\downarrow^{n,k}(\rho', v) \colon D$	(5)
5, lemma 32	ρ' *to* Γ_2	(6)
5, lemma 32	Γ_2 is Γ, Γ'	(7)

$4, 7,$ lemma 7	$\Gamma^{n,k}, \Gamma_2 \vdash p\{i/x\}$ **true**	(8)
$5, 8,$ evaluation	$(\rho, i)\downarrow^{n,k}(\rho', v)\colon \{x : D \mid p\}$	(9)

Case $\Gamma \vdash \mathsf{mkref}\, i : D\ \mathsf{ref}$

premise	$\Gamma^{n,k}, \Gamma \vdash i : D$	(3)
3, induction	$(\rho, i)\downarrow^{n,k}(\rho', v)\colon D$	(4)
ref. ids are countable	$r \notin \rho'$	(5)
$4, 6,$ eval.	$(\rho, \mathsf{mkref}\ i)\downarrow^{n,k}((\rho', r := v), r)\colon D'\ \mathsf{ref}$	(6)

Case $\Gamma \vdash !i : D$

premise	$\Gamma^{n,k}, \Gamma \vdash i : D\ \mathsf{ref}$	(3)
3, induction	$(\rho, i)\downarrow^{n,k}(\rho', r)\colon D\ \mathsf{ref}$	(4)
$4,$ lemma 32	ρ' *to* Γ'	(5)
$4,$ lemma 32	$\Gamma' \vdash r : D\ \mathsf{ref}$	(6)
6, inversion	$r : D\ \mathsf{ref} \in \Gamma'$	(7)
$5, 7,$ lemma 31	$r := v \in \rho'$	(8)
$4, 8,$ evaluation	$(\rho, !i)\downarrow^{n,k}(\rho', v)\colon D$	(9)

Case $\Gamma \vdash i_1 := i_2 : D$

premise	$\Gamma^{n,k}, \Gamma \vdash i_1 : D\ \mathsf{ref}$	(3)
premise	$\Gamma^{n,k}, \Gamma \vdash i_2 : D$	(4)
3, induction	$(\rho, i_1)\downarrow^{n,k}(\rho', r)\colon D\ \mathsf{ref}$	(5)
$4,$ lemma 32	ρ' *to* Γ'	(6)
$4,$ lemma 32	Γ' is Γ, Γ''	(7)
$4, 7,$ lemma 7	$\Gamma^{n,k}, \Gamma' \vdash i_2 : D$	(8)
8, induction	$(\rho', i_2)\downarrow^{n,k}(\rho'', r)\colon D$	(9)
$5, 9,$ evaluation	$(\rho, i_1 := i_2)\downarrow^{n,k}(\rho''[r := v], v)\colon D$	(10)

B.7 Results related to processes

Main results in this section: agreement for process reduction (Lemma 4) and progress for processes (Lemma 6).

Lemma 34 (agreement for process formation).

$$\frac{\Gamma \vdash q : T}{\Gamma \vdash T : \mathbf{type}}$$

Proof. From the hypothesis and the only process formation rule, we know that q is (ρ, e) and that $\Gamma, \rho \vdash e : T$. Agreement (Lemma 1) tells us that $\Gamma, \rho \vdash T : \mathbf{type}$. The premise of all expression formation rules tells us that T contains no ref types. From strengthening (Lemma 8) we get the result.

Lemma 4 (agreement for process reduction). Statement on page 18.

Proof. By rule induction on the hypothesis. We have nine simple cases, and illustrate one of them.

$$(\rho, \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e) \to^{n,k} (\rho, (e\{i/x\}; \mathsf{for}\, x\colon i-1..1\, \mathsf{do}\, e)) \quad (1)$$

rule premise $\quad \Gamma^{n,k}, \rho \vdash i \geq 1\ \mathbf{true}$ (2)

rule premise $\quad \Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash e : T$ (3)

The type for the left hand side follows from the application of expression and process formation rules:

3, exp. formation $\quad \Gamma^{n,k}, \rho \vdash \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e : \forall x \leq i.T$ (4)

4, proc. formation $\quad \Gamma^{n,k} \vdash (\rho, \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e) : \forall x \leq i.T$ (5)

The right hand side requires slightly more effort.

2, inversion $\quad \Gamma^{n,k}, \rho \vdash i : \mathsf{int}$ (6)

6, tautology $\quad \Gamma^{n,k}, \rho \vdash i \leq i\ \mathbf{true}$ (7)

7, def. of subs. $\quad \Gamma^{n,k}, \rho \vdash (y \leq i)\{i/y\}\ \mathbf{true}$ (8)

6, 8, index form. $\quad \Gamma^{n,k}, \rho \vdash i : \{y\colon \mathsf{int} \mid y \leq i\}$ (9)

3, 9, lemma 24 $\quad \Gamma^{n,k}, \rho \vdash e\{i/x\} : T\{i/x\}$ (10)

3, agreement $\quad \Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i\} : \mathbf{context}$ (11)

11, subtyping $\quad \Gamma^{n,k}, \rho \vdash \{y\colon \mathsf{int} \mid y \leq i-1\} <: \{y\colon \mathsf{int} \mid y \leq i\}$ (12)

3, 12, lemma 23 $\quad \Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i-1\} \vdash e : T$ (13)

13, exp. formation $\quad \Gamma^{n,k}, \rho \vdash \mathsf{for}\, x\colon i-1..1\, \mathsf{do}\, e : \forall x \leq i-1.T$ (14)

10, 14, exp. formation $\quad \Gamma^{n,k}, \rho \vdash (e\{i/x\}; \mathsf{for}\, x\colon i-1..1\, \mathsf{do}\, e)\colon T\{i/x\}; \forall x \leq i-1.T$ (15)

3, lemma 20 $\quad \Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \leq i\} \vdash T : \mathbf{type}$ (16)

2, 16, type eq. $\quad \Gamma^{n,k}, \rho \vdash (T\{i/x\}; \forall x \leq i-1.T) \equiv \forall x \leq i.T$ (17)

10, 17, exp.+proc. form. $\quad \Gamma^{n,k} \vdash (\rho, e\{i/x\}; \mathsf{for}\, x\colon i-1..1\, \mathsf{do}\, e) : \forall x \leq i.T$ (18)

Lemma 6 (progress for processes). Statement on page 19.

Proof. By analysis of the hypotheses, with one case for each rule, and a special treatment for $\Gamma^{n,k}, \rho \vdash e : \mathsf{skip}$.

Case let rule: Building from $\Gamma^{n,k}, \rho \vdash i : D$ and the fact that evaluation succeeds (Lemma 3), we obtain $(\rho, i)\downarrow^{n,k}(\rho', v) \colon D$. This, combined with premises $\Gamma^{n,k}, \rho, x \colon D \vdash e : T$ and $x \notin \mathrm{fv}(T)$ constitute the necessary conditions to apply reduction for let processes in Figure 11, obtaining $(\rho, \mathsf{let}\, x \colon D = i\, \mathsf{in}\, e) \rightarrow^{n,k} (\rho', e\{v/x\})$.

Case $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$, $\mathsf{while}\, p\, \mathsf{do}\, e$, and $\mathsf{for}\, x \colon i..1\, \mathsf{do}\, e$ rules: Similar to let.

Case $\Gamma^{n,k}, \rho \vdash e : \mathsf{skip}$: By induction on this assertion, using the inversion lemma for expression formation (Lemma 26).

Subcase e is skip: we are done.

Subcase e is $\mathsf{while}\, p\, \mathsf{do}\, e_1$ and $\Gamma^{n,k}, \rho \vdash p : \mathbf{prop}$ and $\Gamma^{n,k}, \rho \vdash e_1 : \mathsf{skip}$. Analysing the possible truth values of proposition p, we know that either $\mathrm{formulae}(\Gamma^{n,k}, \rho) \models p$ or $\mathrm{formulae}(\Gamma^{n,k}, \rho) \not\models p$. We show the first case (the other is similar). If $\mathrm{formulae}(\Gamma^{n,k}, \rho) \models p$ and $\Gamma^{n,k}, \rho \vdash p : \mathbf{prop}$ then $\Gamma^{n,k}, \rho \vdash p\ \mathbf{true}$. By applying process reduction rules we conclude that $(\rho, \mathsf{while}\, p\, \mathsf{do}\, e_1) \rightarrow^{n,k} (\rho, e_1; \mathsf{while}\, p\, \mathsf{do}\, e_1)$.

Subcase e is $\mathsf{for}\, x \colon i..1\, \mathsf{do}\, e_1$, $\Gamma^{n,k}, \rho \vdash i < 1\ \mathbf{true}$ and $\Gamma^{n,k}, \rho, x \colon \{y \colon \mathsf{int} \mid y \leq i\} \vdash e_1 : \mathsf{skip}$. By direct application of the process reduction rules we can conclude that $(\rho, \mathsf{for}\, x \colon i..1\, \mathsf{do}\, e_1) \rightarrow^{n,k} (\rho, \mathsf{skip})$.

Subcase e is $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1$, and (1) $\Gamma^{n,k}, \rho \vdash i_1 : D_1$ and (2) $y \notin \mathrm{fv}(\mathsf{skip})$, and (3) $\Gamma^{n,k}, \rho, y \colon D_1 \vdash e_1 : \mathsf{skip}$: By applying evaluation always succeeds (lemma 3) to (1), we derive that (4) $(\rho, i) \downarrow^{n,k} (\rho', v) \colon D$. By applying (2–4) and process reduction rules, we can conclude that $(\rho, \mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e_1) \rightarrow^{n,k} (\rho', e_1\{v/y\})$.

Subcase e is $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$, and $\Gamma^{n,k} \vdash p : \mathbf{prop}$, and $\Gamma^{n,k}, \rho \vdash e_1 : \mathsf{skip}$, and $\Gamma^{n,k}, \rho \vdash e_2 : \mathsf{skip}$: Similar to while.

Subcase e is $e_1; e_2$, and (1) $\Gamma^{n,k}, \rho \vdash e_1 : \mathsf{skip}$, and (2) $\Gamma^{n,k}, \rho \vdash e_2 : \mathsf{skip}$. By induction on (1) we obtain e_1 is skip or $(\rho, e_1) \rightarrow^{n,k} q$. It is easy to show that for both cases there is a process reduction rule such that $(\rho, e_1; e_2) \rightarrow^{n,k} q$.

B.8 Results related to programs

Main results in this section: agreement for program formation (Lemma 35), agreement for program reduction (Theorem 1), and progress for programs (Theorem 3).

Lemma 35 (agreement for program formation).

$$\frac{P : S}{S : \mathbf{ptype}}$$

Proof. Directly from the premise of the only rule for assertion $P : S$.

Theorem 1 (agreement for program reduction). Statement on page 21.

Proof. By case analysis on the rules concluding the hypothesis. We have eight cases: one dealing with message passing, one dealing with process reductions and six dealing with collective operations. We illustrate one case for each category.
Case the derivation ends with the broadcast rule:

rule premise	$(\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v) : D$	(1)
rule premise	$\Gamma^n \vdash 1 \leq i_k \leq n$ **true**	(2)
rule premise	$\Gamma^{n,k}, \rho_k \vdash i'_k : D$	(3)
rule premise	$\Gamma^{n,k}, x : D, \rho_k \vdash e_k : T$	(4)
rule premise	D, T contain no ref types	(5)
rule premise	$\mathsf{rank} \notin \mathrm{fv}(D, T)$	(6)

Building the first result, $P_1 : S_1$:

2–4, 6, lem. 7, exp.+proc. form.	$\Gamma^{n,k} \vdash (\rho_k, \mathsf{let}\, x : D = \mathsf{broadcast}\, i_k\, i'_k\, \mathsf{in}\, e_k) : \mathsf{broadcast}\, i_k\, x : D.T$	(7)
4, 6, lemmas 20,8	$\Gamma^n, x : D \vdash T :$ **type**	(8)
2, 8, type form., lemma 2	$(\mathsf{broadcast}\, i_k\, x : D.T)_{k=1}^n :$ **ptype**	(9)
7, 9, prog. form.	$(\rho_k, \mathsf{let}\, x : D = \mathsf{broadcast}\, i_k\, i'_k\, \mathsf{in}\, e_k)_{n=1}^n : (\mathsf{broadcast}\, i_k\, x : D.T)_{k=1}^n$	(10)

Building the second result, $P_2 : S_2$:

2, lemma 32	$\Gamma^{n,l}, \rho_l \vdash i'_l : D$	(11)
2, lemma 32	$\rho'_l \vdash v : D$	(12)
2, lemma 32	$\rho'_l = \rho_l, \rho''_l$	(13)
2, 4, 6, lemma 8	$\varepsilon \vdash v : D$	(14)

$\mathsf{rank} \neq l$:

14, lemma 7	$\Gamma^{n,k}, \rho_k \vdash v : D$	(15)
5, 8, lemma 24	$\Gamma^{n,k}, \rho_k \vdash e\{v/x\} : T\{v/x\}$	(16)
14, proc. formation	$\Gamma^{n,k}, \rho_k \vdash (\rho_k, e\{v/x\}) : T\{v/x\}$	(17)

$\mathsf{rank} = l$:

$5, 13,$ lemma 7	$\Gamma^{n,l}, \rho'_l \vdash e_l : T$	(18)
as for $\mathsf{rank} \neq l$	$\Gamma^{n,l}, \rho'_l \vdash (\rho_l, e\{v/x\}) : T\{v/x\}$	(19)
14, lemma 7	$\Gamma^n, \rho_k \vdash v : D$	(20)
$8, 20,$ lemmas 12,2	$(T\{v/x\})_{k=1}^n$: **ptype**	(21)
$17, 19, 21,$ proc+proc. form.	$(\rho_k, e_k\{v/x\})_{k=1}^{l-1}, (\rho'_l, e_l\{v/x\}), (\rho_k, e_k\{v/x\})_{k=l+1}^n :$ $(T\{v/x\})_{k=1}^n$	(22)

Case the derivation ends with the message rule:

shape of the rule	$1 \leq l, m \leq n$	(1)
rule premise	$i_l \downarrow^n m$	(2)
rule premise	$(\rho_l, i'_l) \downarrow^{n,l} (\rho'_l, v) : D$	(3)
rule premise	$i_m \downarrow^n l$	(4)
rule premise	$(\rho_m, i'_m) \downarrow^{n,m} (\rho'_m, r) : D\ \mathsf{ref}$	(5)
rule premise	$\varepsilon \vdash l \neq m$ **true**	(6)
rule premise	$\Gamma^{n,l}, \rho_l \vdash i'_l : D$	(7)
rule premise	$\Gamma^{n,m}, \rho_m \vdash i'_m : D\,\mathsf{ref}$	(8)
rule premise	$\Gamma^{n,k} \vdash q_k : \mathsf{skip} \quad (1 \leq k \leq n, k \neq l, m)$	(9)
rule premise	D, T contain no ref types	(10)
rule premise	$\mathsf{rank} \notin \mathrm{fv}(D, T)$	(11)

We proceed by building the first result, namely $P_1 : S_1$. The types for processes $(\rho_l, \mathsf{send}\ i_l\ i'_l)$ and $(\rho_m, \mathsf{receive}\ i_m\ i'_m)$ follow from Lemma 33 and from the constraints imposed on l and m:

2, lemma 33	$\Gamma^n \vdash i_l = m$ **true**	(12)
$1, 12,$ transitivity	$\Gamma^n \vdash 1 \leq i_l \leq \mathsf{size}$ **true**	(13)

From the definition of $\Gamma^{n,l}$, we know that $\Gamma^{n,l} \vdash \mathsf{rank} = l$ **true**.

$$6, \Gamma^{n,l} \vdash \mathsf{rank} = l \textbf{ true} \qquad \Gamma^{n,l} \vdash m \neq \mathsf{rank} \textbf{ true} \tag{14}$$

By applying agreement for evaluation (Lemma 32), we infer:

$$3, \text{lemma } 32 \qquad \Gamma^{n,l}, \rho_l \vdash i'_l : D \tag{15}$$

Weakening (Lemma 7) is used in rules (13–15) to extend the type environment to $\Gamma^{n,l}, \rho_l$ (details omitted). Such steps build the type for send:

$$13, 14, 15, \text{lem. } 7, \text{ exp. form.} \qquad \Gamma^{n,l}, \rho_l \vdash \mathsf{send}\ i_l\ i'_l : \mathsf{message}\ l\ i_l\ D \tag{16}$$

A similar derivation yields the type for receive:

4, lemma 33	$\Gamma^n \vdash i_m = l$ **true**	(17)
$1, 19$, transitivity	$\Gamma^n \vdash 1 \leq i_m \leq \mathsf{size}$ **true**	(18)
$6, \Gamma^{n,m} \vdash \mathsf{rank} = m$ **true**	$\Gamma^{n,m} \vdash l \neq \mathsf{rank}$ **true**	(19)
5, lemma 32	$\Gamma^{n,m}, \rho_m \vdash i'_m : D\ \mathsf{ref}$	(20)
$18, 19, 20$, lem. 7, exp. form.	$\Gamma^{n,m}, \rho_m \vdash \mathsf{receive}\ i_m\ i'_m : \mathsf{message}\ i_m\ m\ D$	(21)
$16, 12$, proc. form.	$\Gamma^{n,l} \vdash (\rho_l, \mathsf{send}\ m\ i'_l) : \mathsf{message}\ l\ m\ D$	(22)
$21, 17$, proc. form.	$\Gamma^{n,m} \vdash (\rho_m, \mathsf{receive}\ l\ i'_m) : \mathsf{message}\ l\ m\ D$	(23)

Building the **ptype**:

$7, 10, 11$, lemma 8	$\Gamma^n \vdash i'_l : D$	(24)
24, lemma 1	$\Gamma^n \vdash D :$ **dtype**	(25)
6, lemma 7	$\Gamma^n \vdash l \neq m$ **true**	(26)
$25, 26$, **ptype** form.	$\mathsf{skip}_1, \ldots, \mathsf{skip}_{l-1}, \mathsf{message}\ l\ m\ D, \mathsf{skip}_{l+1}, \ldots, \mathsf{skip}_{m-1}, \mathsf{message}\ l\ m\ D, \mathsf{skip}_{m+1}, \ldots, \mathsf{skip}_n :$ **ptype**	(27)
$9, 22, 23, 27$, prog. form.	$(q_k)_{k=1}^{l-1}, (\rho_l, \mathsf{send}\ i_l\ i'_l), (q_k)_{k=l+1}^{m-1}, (\rho_m, \mathsf{receive}\ i_m\ i'_m), (q_k)_{k=m+1}^{n} : (\mathsf{skip}_k)_{k=1}^{l-1}, \mathsf{message}\ l\ m\ D, (\mathsf{skip}_k)_{k=l+1}^{m-1}, \mathsf{message}\ l\ m\ D, (\mathsf{skip}_{k=m+1}^{n})$	(28)

Building the second result, $P_2 : S_2$:

3, lemma 32	$\rho'_l \vdash v : D$	(29)
29, lemma 17 - applied twice	$\rho'_l :$ **context**	(30)
30, lemmas 7, 13	$\Gamma^{n,l}, \rho'_l :$ **context**	(31)
31, exp.+proc. formation	$\Gamma^{n,l} \vdash (\rho'_l, \mathsf{skip}) : \mathsf{skip}$	(32)
5, lemma 32	$\rho'_m \vdash r : D\ \mathsf{ref}$	(33)
$10, 29$, lemma 8	$\varepsilon \vdash v : D$	(34)
34, lemma 7	$\rho'_m \vdash v : D$	(35)
$33, 35$, store update	$\rho'_m[r := v] \vdash r : D\ \mathsf{ref}$	(36)
36, lemma 17	$\rho'_m[r := v] :$ **context**	(37)
37, lemmas 21, 25	$\Gamma^{n,m}, \rho'_m[r := v] :$ **context**	(38)
38, exp.+ proc. formation	$\Gamma^{n,m} \vdash (\rho'_m[r := v], \mathsf{skip}) : \mathsf{skip}$	(39)
ptype formation	$(\mathsf{skip}_k)_{k=1}^{n} :$ **ptype**	(40)
$9, 32, 39, 41$, prog. form	$(q_k)_{k=1}^{l-1}, (\rho'_l, \mathsf{skip}), (q_k)_{k=l+1}^{m-1}, (\rho'_m[r := v], \mathsf{skip}), (q_k)_{k=m+1}^{n} : (\mathsf{skip}_k)_{k=1}^{n}$	(41)

Case the derivation ends with process reduction:

rule premise	$q_l \rightarrow^{n,l} q'_l$	(1)

rule premise	$\Gamma^{n,k} \vdash q_k : T_k$	(2)
rule premise	$T_1, \ldots, T_l, \ldots, T_n$: **ptype**	(3)

The proof relies on agreement for process reduction (lemma 4).

2, 3, prog. form.	$q_1, \ldots, q_l, \ldots, q_n : T_1, \ldots, T_l, \ldots, T_n$	(4)
1, lemma 4	$\Gamma^{n,l} \vdash q_l' : T_l$	(5)
2, 4, 5, prog. form.	$q_1, \ldots, q_l', \ldots, q_n : T_1, \ldots, T_l, \ldots, T_n$	(6)

Theorem 3 (progress for programs). Statement on page 22.

Proof. From the hypothesis and the formation rules for programs and processes we know that:

P_1 is $(\rho_1, e_1), \ldots, (\rho_n, e_n)$	(1)
S_1 is $T_1, \ldots, T_n$	(2)
$\Gamma^{n,k}, \rho_k \vdash e_k : T_k \quad (k = 1..n)$	(3)
$T_1, \ldots, T_n$: **ptype**	(4)

The proof proceeds by rule induction on assertion $T_1, \ldots, T_n$: **ptype**. There are ten cases to consider. We illustrate a few.
Case the derivation ends with the type equality rule:

rule premise	$T_1, \ldots, T_k', \ldots, T_n$: **ptype**	(5)
rule premise	$\Gamma^{n,k} \vdash T_k \equiv T_k' :$ **type**	(6)
3, 6, exp. formation	$\Gamma^{n,k}, \rho_k \vdash e_k : T_k'$	(7)
1, 3, 7, 5, proc+prog. formation	$P_1 : T_1, \ldots, T_k', \ldots, T_n$	(8)
8, induction	$P + 1$ **halted** *or* $P_1 \to P_2$	(9)

Case the derivation ends with the broadcast rule:

rule premise	$\Gamma^n \vdash 1 \leq l \leq n$ **true**	(5)
rule premise	$\Gamma^n, x\colon D \vdash T :$ **type**	(6)
6, lemma 7, lemma 13	$\Gamma^{n,k}, \rho_k, x : D \vdash T :$ **type**	(7)

Each expression e_k $(1 \leq k \leq n)$ may be of four different forms, according to inversion for expression formation:
Subcase each expression e_k is $\mathsf{let}\, x\colon D_1 = \mathsf{broadcast}\, i_k\, i_k'\, \mathsf{in}\, e_k'$:

lemma 26	$\Gamma^{n,k}, \rho_k \vdash D_1 \equiv D :$ **dtype**	(8)
lemma 26	$\Gamma^{n,k}, \rho_k \vdash i_k = l$ **true**	(9)
lemma 26	$\Gamma^{n,k}, \rho_k \vdash 1 \leq i_k \leq \mathsf{size}$ **true**	(10)
lemma 26	$\Gamma^{n,k}, \rho_k \vdash i_k' : D$	(11)

$$\text{lemma 26} \qquad \mathsf{rank} \notin \mathrm{fv}(i_1) \tag{12}$$

$$\text{lemma 26} \qquad \Gamma^{n,k}, x\colon D_1, \rho_k \vdash e'_k : T \tag{13}$$

Building from index typing, deducibility and the fact that evaluation always succeeds, we get:

$$5, \text{inversion} \qquad \Gamma^n \vdash 1 \leq l \leq n : \mathbf{prop} \tag{14}$$

$$14, \text{index form.} \qquad \Gamma^n \vdash l : \mathsf{int} \tag{15}$$

$$14, 15, \text{index form.} \qquad \Gamma^n \vdash l : \{y : \mathsf{int} \mid 1 \leq y \leq n\} \tag{16}$$

$$9, 12, \text{inversion, lemmas 19, 8} \qquad \Gamma^n \vdash i_k : \mathsf{int} \tag{17}$$

$$17, \text{lemma 3} \qquad i_k \downarrow^n l \tag{18}$$

$$11, \text{lemma 3} \qquad (\rho_k, i'_k)\downarrow^{n,k} (\rho''_k, v)\colon D \tag{19}$$

We have now all the ingredients to perform reduction:

$$10, 11, 13, 18, 19, \text{prog. red.} \qquad (\rho_k, \mathsf{let}\, x\colon D = \mathsf{broadcast}\, i_k\, i'_k\, \mathsf{in}\, e'_k)_{k=1}^n \to (\rho_k, e'_k\{v/x\})_{k=1}^{i-1}, (\rho''_i, e'_i\{v/x\}), (\rho_k, e'_k\{v/x\})_{k=i+1}^n \tag{20}$$

Subcase at least one of the expressions e_k is not broadcast.

From inversion of expression formation we have three additional cases.

Subsubcase e_k is $\mathsf{let}\, y_k\colon D_k = i'_k\, \mathsf{in}\, e'_k$:

$$\text{lemma 26} \qquad \Gamma^{n,k}, \rho_k \vdash i'_k : D_k \tag{21}$$

$$\text{lemma 26} \qquad y \notin \mathrm{fv}(\mathsf{broadcast}\, i\, x\colon D.T) \tag{22}$$

$$\text{lemma 26} \qquad \Gamma^{n,k}, \rho_k, y_k : D_k \vdash e'_k : \mathsf{broadcast}\, i_k\, x\colon D.T \tag{23}$$

We apply progress for processes followed by program reduction:

$$21, 22, 23, \text{lemma 6} \qquad (\rho_k, \mathsf{let}\, y_k\colon D_k = i'_k\, \mathsf{in}\, e_k) \to^{n,k} q_k \tag{24}$$

$$3, 4, 24, \text{prog. red.} \qquad P_1 \to (\rho_1, e_1), \ldots, q_k, \ldots, (\rho_n, e_n) \tag{25}$$

Subsubcase e_k is $\mathsf{if}\, p\, \mathsf{then}\, e'_k\, \mathsf{else}\, e''_k$:

$$\text{lemma 26} \qquad \Gamma^{n,k}, \rho_k \vdash p : \mathbf{prop} \tag{26}$$

$$\text{lemma 26} \qquad \Gamma^{n,k}, \rho_k \vdash e'_k : \mathsf{broadcast}\, i_k\, x\colon D.T \tag{27}$$

$$\text{lemma 26} \qquad \Gamma^{n,k}, \rho_k \vdash e''_k : \mathsf{broadcast}\, i_k\, x\colon D.T \tag{28}$$

We just need to apply progress to processes and program reduction.

$$26, 27, 28, \text{lemma 6} \qquad \mathsf{if}\, p\, \mathsf{then}\, e'_k\, \mathsf{else}\, e''_k \to^{n,k} q_k \tag{29}$$

$$3, 4, 29, \text{ prog. red.} \qquad P_1 \to (\rho_1, e_1), \ldots, q_k, \ldots, (\rho_n, e_n) \tag{30}$$

Subsubcase e_k is $e'_k; e''_k$:

$$\text{lemma 26} \qquad \Gamma^{n,k}, \rho_k \vdash e'_k : \mathsf{skip} \tag{31}$$

lemma 26	$\Gamma^{n,k}, \rho_k \vdash e''_k : \mathsf{broadcast}\, i_k\, x \colon D.T$	(32)

From progress for processes applied to e'_k we have:

31, lemma 6	e'_k is skip $\quad or \quad (\rho_k, e'_k) \to^{n,k} (\rho'_k, e'''_k)$	(33)

Subsubsubcase e'_k is skip: Follows by simple application of process and program reduction rules:

33	e'_k is skip	(34)
32, 34, proc. red.	$(\rho_k, \mathsf{skip}; e''_k) \to^{n,k} (\rho_k, e''_k)$	(35)
3, 4, 36, prog. red.	$P_1 \to (\rho_1, e'_k), \ldots, (\rho_k, e''_k), \ldots, (\rho_n, e'_n)$	(36)

Subsubsubcase e'_k reduces: Again a simple application of process and program reduction rules yields:

33	$(\rho_k, e'_k) \to^{n,k} (\rho'_k, e'''_k)$	(37)
32, 37, proc. red.	$(\rho_k, (e'_k; e''_k)) \to^{n,k} (\rho'_k, (e'''_k, e''_k))$	(38)
3, 4, 38, prog. red.	$P_1 \to (\rho_1, e'_k), \ldots, (\rho'_k, (e'''_k, e''_k)), \ldots, (\rho_n, e'_n)$	(39)

Case the derivation ends with the message rule.

rule premise	$\Gamma^n \vdash l \neq m$ **true**	(1)
rule premise	$\Gamma^n \vdash D : \mathbf{dtype}$	(2)

According to inversion for expression formation lemma we have five cases for each of the two e_j, e_m expressions.
Subcase e_l is $\mathsf{send}\, i_l\, i'_l$ and e_m is $\mathsf{receive}\, i_m\, i'_m$ and $l < m$.

lemma 26	$\Gamma^n \vdash \mathsf{rank} = l$ **true**	(3)
lemma 26	$\Gamma^n \vdash i_l = m$ **true**	(4)
lemma 26	$\Gamma^n \vdash i'_l : D$	(5)
lemma 26	$\Gamma^n \vdash i_m = l$ **true**	(6)
lemma 26	$\Gamma^n \vdash \mathsf{rank} = m$ **true**	(7)
lemma 26	$\Gamma^n \vdash i'_m : D\ \mathsf{ref}$	(8)
4, lemma 7	$\Gamma^{n,l}, \rho_l \vdash i_l = m$ **true**	(9)
9, lemma 3	$(\rho_l, i_l) \downarrow^{n,k} (\rho'_l, m) \colon D$	(10)
10, i_l does not contain references	$i_l \downarrow^n m$	(11)

By applying to (9-11) similar steps as for i_m:

steps 9-11, applied to i_m	$i_l \downarrow^n m$	(12)

We can infer that $(\rho_l, i'_l) \downarrow^{n,l} (\rho'_l, v) \colon D$ and $(\rho_m, i'_m) \downarrow^{n,m} (\rho'_m, r) \colon D$ via weakening and that evaluation always succeeds (lemma 3):

5, lemma 7	$\Gamma^{n,l}, \rho_l \vdash i'_l : D$	(13)

$$\begin{array}{ll} 13, \text{lemma } 3 & (\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v) \colon D \quad (14) \\ 8, \text{lemma } 7 & \Gamma^{n,m}, \rho_m \vdash i'_m : D \ \mathsf{ref} \quad (15) \\ 15, \text{lemma } 3 & (\rho_m, i'_m)\downarrow^{n,m}(\rho'_m, r) \colon D \quad (16) \end{array}$$

Finally, we need to derive $\varepsilon \vdash l \neq m$ **true**:

$$\begin{array}{ll} 1, \text{deducibility inversion} & \Gamma^n \vdash l \neq m : \mathbf{prop} \quad (17) \\ 3, \text{deducibility and prop inversions} & \Gamma^n \vdash l : \mathsf{int} \quad (18) \\ 7, \text{deducibility and prop inversions} & \Gamma^n \vdash m : \mathsf{int} \quad (19) \\ 17, \text{lemma } 8, \text{ truth formation} & \varepsilon \vdash l \neq m \ \mathbf{true} \quad (20) \end{array}$$

And putting all together:

$$11\text{–}16, 20, \text{form.} \quad q_1, \ldots, (\rho_l, \mathsf{send}\ i_l\ i'_l), \ldots, q_{m-1}, (\rho_m, \mathsf{receive}\ i_m\ i'_m), \ldots, q_n \to q_1, \ldots, (\rho'_l, \mathsf{skip}), \ldots, q_{m-1}, (\rho'_m[r := v], \mathsf{skip}), \ldots, q_n \quad (21)$$

Subcase e_l is $\mathsf{send}\, i_l\, i'_l$ and e_m is $\mathsf{receive}\, i_m\, i'_m$ and $l > m$ (from (1) we know that the case for $l = m$ does not apply).

Similar to the case above.

Subcase Either e_l or e_m are not receive or send. We show the case for e_l, the other is similar.

Subsubcase e_l is $\mathsf{let}\, y \colon D_1 = i_1\, \mathsf{in}\, e'_l$

$$\begin{array}{ll} \text{lemma } 26 & \Gamma^n \vdash i_1 : D_1 \quad (22) \\ \text{lemma } 26 & y \notin \mathrm{fv}(\mathsf{message}\, l\, m\, D) \quad (23) \\ \text{lemma } 26 & \Gamma^n, y : D_1 \vdash e'_l : \mathsf{message}\, l\, m\, D \quad (24) \\ 22, \text{lemma, } 7 & \Gamma^{n,l}, \rho_l \vdash i_1 : D_1 \quad (25) \\ 24, \text{lemma } 21, \text{ lemma } 25 & \Gamma^{n,l}, \rho_l, y : D_1 \vdash e'_l : \mathsf{message}\, l\, m\, D \quad (26) \end{array}$$

By applying progress for processes:

$$\begin{array}{ll} 23, 25, 26, \text{lem. } 6 & (\rho_l, e_l) \to^{n,l} q_l \quad (27) \\ 27, \text{prog. red.} & \mathsf{skip}_1, \ldots, (\rho_l, e_l), \mathsf{skip}_{l+1}, \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \to \mathsf{skip}_1, \ldots, q_l, \mathsf{skip}_{l+1}, \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \quad (28) \end{array}$$

Subsubcase e_l is $\mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$

From inversion of expression formation we get:

$$\begin{array}{ll} \text{lemma } 26 & \Gamma^n \vdash p : \mathbf{prop} \quad (29) \\ \text{lemma } 26 & \Gamma^n \vdash e_1 : \mathsf{message}\, l\, m\, D \quad (30) \\ \text{lemma } 26 & \Gamma^n \vdash e_2 : \mathsf{message}\, l\, m\, D \quad (31) \end{array}$$

We weaken the context with ρ_l:

$$29, \text{lemma } 7 \qquad \Gamma^{n,l}, \rho_l \vdash p : \mathbf{prop} \quad (32)$$

$$30, \text{lemma } 21 \qquad \Gamma^{n,l}, \rho_l \vdash e_1 : \mathsf{message}\, l\, m\, D \tag{33}$$

$$31, \text{lemma } 21 \qquad \Gamma^{n,l}, \rho_l \vdash e_2 : \mathsf{message}\, l\, m\, D \tag{34}$$

apply progress for processes:

$$32, 33, 34, \text{lem. } 6 \quad (\rho_l, \mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2) \to^{n,l} q_l \tag{35}$$

$$35, \text{prog. red.} \quad \mathsf{skip}_1, \ldots, (\rho_l, e_l), \mathsf{skip}_{l+1}, \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \to \mathsf{skip}_1, \ldots, q_l, \mathsf{skip}_{l+1}, \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \tag{36}$$

Subsubcase e_l is $e_1; e_2$:

From inversion of expression formation and weakening we get:

$$\text{lemma } 26 \qquad \Gamma^n \vdash e_1 : \mathsf{skip} \tag{37}$$

$$\text{lemma } 26 \qquad \Gamma^n \vdash e_2 : \mathsf{message}\, l\, m\, D \tag{38}$$

$$37, \text{lemma } 21 \qquad \Gamma^{n,l}, \rho_l \vdash e_1 : \mathsf{skip} \tag{39}$$

$$38, \text{lemma } 21 \qquad \Gamma^{n,l}, \rho_l \vdash e_2 : \mathsf{message}\, l\, m\, D \tag{40}$$

From progress for processes we have that:

$$39, \text{lemma } 6 \qquad e_1 \text{ is } \mathsf{skip} \text{ or } (\rho_l, e_1) \to^{n,l} (\rho_l', e_3) \tag{41}$$

Subsubsubcase e_1 is skip: By applying process and program reduction as we in the previous case:

$$40, \text{proc. red.} \quad (\rho_l, e_1; e_2) \to^{n,l} (\rho_l, e_2) \tag{42}$$

$$42, \text{prog. red.} \quad \mathsf{skip}_1, \ldots, (\rho_l, e_l), \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \to \mathsf{skip}_1, \ldots, (\rho_l, e_2), \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \tag{43}$$

Subsubsubcase $(\rho_l, e_1) \to^{n,l} (\rho_l', e_3)$: By applying process and program reductions we get:

$$40, \text{proc. red.} \quad (\rho_l, e_1; e_2) \to^{n,l} (\rho_l', e_3; e_2) \tag{44}$$

$$44, \text{prog. red.} \quad \mathsf{skip}_1, \ldots, (\rho_l, e_l), \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \to \mathsf{skip}_1, \ldots, (\rho_l', e_3; e_2), \ldots, \mathsf{skip}_{m-1}, (\rho_m, e_m), \ldots, \mathsf{skip}_n \tag{45}$$