Iktara*in ConCert: Realizing a Certified Grid Computing Framework from a Programmer's Perspective

Bor-Yuh Evan Chang

May 3, 2002

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

With the vast amount of computing resources distributed throughout the world today, the prospect of effectively harnessing these resources has captivated the imaginations of many and motivated both industry and academia to pursue this dream. We believe that fundamental to the realization of this dream is the establishment of trust between application developers and resource donors as donors often receive little or no direct reward for their contributions. The ConCert project (to which this specific undertaking contributes) seeks to develop the theoretical and engineering foundation for grid computing in such a trustless setting based on the notion of certified code.

In this paper, we seek to drive an initial implementation of a real grid framework from a programmer's perspective. Specifically, we present a model for programming the grid and a case study of a specific application, namely a theorem prover for intuitionistic linear logic (Iktara), that provides motivation for and guides the design of such a programming model.

*Iktara is the name of a parallel theorem prover for intuitionistic linear logic developed as part of this project. An Iktara is an ancient one-stringed instrument from northern India.

Contents

1 Introduction **3**

2 Programming in ConCert **3**

2.1 Types of Parallelism 4
2.2 ML Interface 4
2.3 Implementation 7
2.4 Example: Merge Sort 8
2.5 Simulator 9
2.5.1 Interface 10
2.5.2 Implementation 11
2.5.3 Limitations 12

3 Parallel Theorem Proving for Linear Logic **12**

3.1 Why a Theorem Prover Application? 13
3.2 Bottom-Up Proof Search for Intuitionistic Linear Logic 13
3.2.1 Focusing 14
3.2.2 Resource Management 17
3.3 Parallelization 20
3.4 Implementation 21
3.4.1 Ordered Binary Decision Diagrams 21
3.4.2 Boolean Constraints 22
3.4.3 Focusing 24
3.4.4 ConCert Simulator Version 24

4 Conclusion **25**

A Dynamic Semantics of Tasks **30**

1 Introduction

In recent years, we have seen numerous organizations desperately seeking resource donors for applications from discovering a cure for AIDS [Fig00] to finding larger and larger prime numbers [GIM96] to searching for extraterrestrial intelligence [SET01]. Part of the difficulty in obtaining resources is establishing trust between the distributed-application developer and the resource donors. Because resource donors often receive little or no direct reward for their contributions, they will certainly demand assurances of safety, security, and privacy. Given the current environment where numerous malicious application developers populate the Internet and even benign developers have difficulty producing safe and reliable software, this is to be expected.

Today, establishing this trust in the distributed-application developer is limited to relying on faith—based on the reputation of the developer or possibly the reputation of some "quality assurance team" that endorses him. In addition, this faith must be reexamined each time the developer needs to pester the donor to download and install a software upgrade at the possible risk of the stability of his system. To address this issue, the ConCert project [Con01a] seeks to develop the theoretical and engineering foundation for grid computing in such a trustless setting. To this end, it has been proposed to use *certifying compilers* to allow software developers to produce efficient machine code along with checkable certificates that could be easily verified on behalf of the resource donor. This would ensure the software's compliance with policies specified by the donor. We believe this is the best means to create a grid infrastructure that allows distributed-application developers to propagate their software to as many hosts as possible and maximize the exploitation of these resources. The vision is to allow developers to "freely disseminate their software to willing hosts" and to establish trust "via rigorous, mathematical proof of the security and privacy properties of software" [Con01b].

We seek to develop an initial implementation of a real framework for the distribution and verification of software. Such a development process aims to inspire new ideas and to expose technical problems that both motivate further theoretical work and provide a testbed for the implementation of such ideas. In this paper, we focus on driving such a framework from a programmer's perspective. We search for a reasonable and effective model for programming the grid that will allow the development of as many applications as possible. In order to better understand the design space of such a framework and programming model, we perform a case study on a specific application, namely a parallel theorem prover for linear logic (Iktara), that imposes a unique and guiding set of requirements.

2 Programming in ConCert

A large part of our research is to develop techniques for programming grid applications. To this end, we have designed an imaginary (but we think reasonable) interface for the initial ConCert framework in Standard ML (SML) [MTHM97]. In this section, we briefly discuss the types of parallelism that motivate the design of our programming interface. We then explain the interface and how we arrived at it, discuss its implementation, and give a simple example. Finally, we present a simulator for our proposed programming interface.

2.1 Types of Parallelism

Currently, the most widely known grid applications, such as SETI@home (Search for Extraterrestrial Intelligence at Home) [SET01] and GIMPS (Great Internet Mersenne Prime Search) [GIM96] can be characterized as highly parallel applications over an enormous data set. The type of parallelism used by applications can be approximately characterized by the amount of communication between threads of computation and the depth and width of the parallel computation. Both these successful grid applications have the property that each thread of computation is fairly independent of each other (i.e. have little or no communication between them) and have a branching structure that is very wide and very shallow. Contrast this to parallel applications that are written for shared memory machines that usually have high communication requirements between threads of computation and often have narrower branching.

Given the non-homogeneous and failure-prevalent nature of the grid, we also intend to target applications that have very little or no communication requirements between threads of computation. However, we also seek to explore the feasibility of the space of applications that use less embarrassingly parallel algorithms, such as game tree search, than those used on grid networks today. In order to explore this space, it is imperative to have a reasonable programming model.

2.2 ML Interface

At a high level, a *job* is thought of as a whole program that is started and injected into the network on the command-line. If jobs were the sole unit of computation distributed on the network, then no additional programming constructs would be needed. Any language with a compiler capable of generating binaries supported by the distribution architecture would be sufficient. In this framework, we could likely implement some easily partitioned applications this way. Of course, this method is unacceptable for applications that do not have trivial branching and cumbersome even for programs that do.

We desire for the programmer to be able to expose the parallelism in his application in a convenient but simple way. We draw on the idea of *futures* from Multilisp [Hal85] and a similar construct in Cilk-NOW [BL97] for our design. We refine the notion of a job to be composed of one or more *tasks*. A task is then the unit of computation from the programmer's point of view.

Figure 1 gives a ML signature for creating and managing tasks. It should be noted that though it is convenient to express it this way, we would not actually be able to implement it in ML. In this regard, we might think of this as a proposed language extension rather than a library. The interface is intended to be minimalistic as to capture only the necessary constructs for programming the grid. For some amount of realism, we require that code injected into the network be closed, though we imagine that a language designed for grid computing could arrange for this automatically.

```
structure CCStatus =
struct

  datatype status =
    Disabled
  | Failed
  | Finished
  | Running
  | Waiting

end; (* structure CCStatus *)

signature CCTASKS =
sig

  (* an 'r task is a computation yielding a result of type 'r *)
  type 'r task
  exception InvalidTask

  val inject  : bool -> ('e -> 'r) * 'e -> 'r task
  val enable  : 'r task -> unit

  val sync    : 'r task -> 'r
  val syncall : 'r task list -> 'r list
  val relax   : 'r task list -> 'r * 'r task list

  val forget  : 'r task -> unit
  val status  : 'r task -> CCStatus.status

end; (* signature CCTASKS *)
```

Figure 1: CCTASKS Signature

To place a task on the network,

`inject : bool -> ('e -> 'r) * 'e -> 'r task`

is used. A task can either be injected into the network ready to run or be injected in a *disabled* state. A disabled task requires an explicit call to `enable` by some other task to indicate the task is ready to run. The expression `inject true (c,e)` indicates the given code `c` along with the initial environment `e` should be used to create a task that is ready to run, while `inject false (c,e)` gives a task that is initially disabled. See section 3.3 for a potential use of disabled tasks.

Returning a result and receiving results from other tasks are the only form of communication between tasks. This helps to enforce the restartable nature of tasks, which will become important for implementing failure recovery. To obtain a result from another task, one might use `sync`. The `sync` function blocks the calling task until the desired result can be obtained from the network. There are four possible states of the task from which we seek the result:

1. it has completed execution successfully, so the result is already known on the network;
2. it is currently executing;
3. it has failed (or appears to have failed);
4. it has been disabled.

If scenario 1 is the case, then the result can be returned immediately to the calling task. Otherwise, if scenario 2 is the case, then the calling task is blocked until the other task finishes. If scenario 3 is the case, then a new process will be started to restart the execution of that task (possibly on another node on the network). Finally, if that task has been disabled (case 4), then the calling task will be blocked until the task has been enabled to run. To receive results from more than one task, `syncall` is provided as it can be implemented more efficiently than the semantically equivalent successive calls to `sync` for each desired task.

In the process of developing the theorem prover, we noticed that we desired some ability to continue execution as soon as one result was ready from a set of tasks. We propose a new construct `relax` that returns as soon as one result is ready returning that result along with the remaining tasks. The usual usage of `relax` would entail the consumption of all the results. In this regard, `relax` should be thought of as having a conjunctive meaning analogous to `syncall` except that the order of synchronization is *relaxed*. This construct implements in ConCert the `select-wrap` idiom often seen in CML [Rep99] without requiring the full generality of CML-like events. Also, notice that `sync` can be defined in terms of either `syncall` or `relax` as shown below:

```
fun sync t = hd (syncall [t])
fun sync t = #1 (relax [t])
```

As an optimization, tasks can provide a hint to the scheduler that a particular task is no longer needed by using the `forget` function. Since the architecture must deal with failures, the `forget` function is simply a special case of failure in which a task fails deliberately. This means that if any result requests are pending for the aborted task, the task may be restarted.

Lastly, we give the programmer the ability to query the status of any task (i.e. whether it is running, is finished, is disabled, is waiting, or has failed). This is not strictly necessary, but the programmer may be able to optimize his application with this information.

For further description of this programming model, see appendix A for a presentation of a dynamic semantics of these proposed language constructs.

2.3 Implementation

The implementation of such an interface has been coordinated with DeLap's work on the ConCert software [DeL02] that implements the code distribution and verification infrastructure as we seek to map this high-level programming language onto the low-level interface that ConCert provides. Similar to the Cilk-NOW architecture [BL97], we impose the following invariants on every program fragment:

1. the fragment is deterministic or in some notion each subsequent run is as good as any other;
2. program fragments do not communicate except through explicit dependencies;
3. once its dependencies are filled, a program fragment is able to run to completion.

Invariants 1 and 2 are intended to ensure that each fragment is restartable. Running a program fragment multiple times should result in the same answer, or if the programmer desires, an answer that is "as good as" any other answer he might receive. Invariant 3 is designed to simplify scheduling of program fragments by clearly separating the program fragments that are waiting and the program fragments that are running.

These invariants lead to the characterization of the smallest unit of computation in this formulation of the ConCert framework. In the Cilk-NOW terminology, this is called a *closure*. To disambiguate this from other uses of this term, we use the word *cord*. A *cord* consists of code, an environment, and a set of dependencies on other cords. We can think of the dependencies as extra arguments to the code. A cord is ready to run as soon as its dependencies have been fulfilled. The scheduling and distribution of cords is described in DeLap's senior thesis [DeL02].

Because of invariant 3, *sync*ing on a result will involve the termination of the current cord and then the re-spawning of a new cord with an added dependency for the desired result.

Figure 2 gives a conceptual picture of these terms. Each box represents a particular view on the unit of computation. In essence, a job is the unit of computation from the grid-application user's point of view, a task is the unit of computation from the grid-application developer's point of view, and a cord is the unit of computation from the ConCert software's point of view.

Work on a compiler for ML with these proposed extensions has begun but is not yet complete as there are still open questions concerning the marshaling of data and the low-level interface provided by the ConCert software is still evolving. However, this is certainly a high priority item.

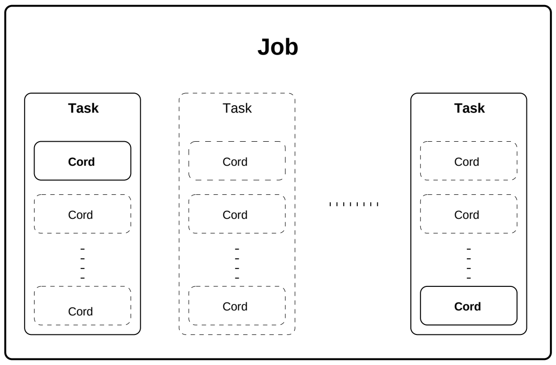

Figure 2: Jobs, Tasks, and Cords. A *job* is injected into the network from the command-line of some participating node. A job consists of several *tasks* possibly running on different machines. Only one *cord* is active per task and is the unit of computation scheduled and distributed. In the above diagram, solid lines indicate the job/task/cord is running while dashed lines indicate the job/task/cord has finished or has yet to run.

2.4 Example: Merge Sort

Below is a (hypothetical) example of a simple implementation of merge sort that divides into three subproblems using the interface specified above. This is done especially to highlight the use of the `relax` construct. Lines 6-31 contain straightforward implementations of partitioning a list and merging two lists. Now, focus your attention on lines 44-46 where tasks for the subproblems are injected into the network. Then, in lines 50-56, we wait for these results. Notice, we begin merging as soon as we have two sorted lists.

```
 1   (* mergesort : int list * int -> int list *)
 2   fun mergesort (nil, _) = nil
 3     | mergesort ([x], _) = [x]
 4     | mergesort (l, cutoff) =
 5       let
 6         (* partition : int * int list -> int list * int list * int list *)
 7         fun partition (m, l) =
 8             let
 9               fun partition' (_, nil) = (nil, nil)
10                 | partition' (0, k as x::t) = (nil, k)
11                 | partition' (n, x::t) =
12                   let
13                     val (l',k') = partition' (n-1, t)
14                   in
15                     (x::l', k')
16                   end
```

```

            val (ll,lr)   = partition' (m, l)
            val (lrl,lrr) = partition' (m, lr)
          in
            (ll,lrl,lrr)
          end

     (* merge : int list * int list -> int list *)
     fun merge (l, nil) = l
       | merge (nil, k) = k
       | merge (l as x::l', k as y::k') =
         (case Int.compare (x,y) of
            LESS    => x::(merge (l',k))
          | EQUAL   => x::y::(merge (l',k'))
          | GREATER => y::(merge (l,k')))

     val len = List.length l
     val (lt,md,rt) = partition (len div 3, l)
   in
     if (len <= cutoff) then
       merge (mergesort (lt,cutoff),
              merge (mergesort (md,cutoff), mergesort (rt,cutoff)))
     else
       let
         open CCTasks

         (* Start sorting each partition *)
         val t1 = injectTask true (mergesort, (lt, cutoff))
         val t2 = injectTask true (mergesort, (md, cutoff))
         val t3 = injectTask true (mergesort, (rt, cutoff))

         (* Get the results of the three child tasks.  Start
     merging when two sorted lists have been received. *)
         val (sort1, sort2) =
           let
             val (a, rest) = relax [ t1, t2, t3 ]
             val (b, [last]) = relax rest
           in
             (merge (a,b), sync last)
           end
       in
         merge (sort1, sort2)
       end
   end
```

2.5 Simulator

To verify the usability of such a programming interface without depending on the progress of the ConCert software, we have developed a simulator of the network running on a single machine that provides functionality for the proposed interface. The simulator may also be a useful development tool for ConCert application developers to test their software in a controlled environment before deploying to the network. Although the simulator aims to be as realistic as possible, certain compromises had to be made as the simulator is developed entirely in CML and runs on a single machine. In this section, we discuss the implementation of the simulator and the points where we were forced to sacrifice some amount of realism.

2.5.1 Interface

The simulator is written in Concurrent ML (CML) [Rep99] using the SML/NJ compiler. The code will be made available from the ConCert webpage [Con01a]. Ideally, we would be able to implement the simulator ascribing to the signature given in figure 1; however, as mentioned before, we cannot implement that signature directly in SML/CML. Figure 3 gives the modified version of the `CCTASKS` signature. The most noticeable difference is that type `task` is monomorphic

```
signature CCTASKS =             (* Modified for the Simulator *)
sig

  type env
  type result

  type task                     (* "result" task *)
  exception InvalidTask

  val inject : bool -> (env -> result) * env -> task
  val enable : task -> unit

  val sync : task -> result
  val syncall : task list -> result list
  val relax  : task list -> result * task list

  val forget : task -> unit
  val status : task -> CCStatus.status

  val startSim : real -> (env -> result) * env -> result

end; (* signature CCTASKS *)
```

Figure 3: Modified CCTASKS signature for the simulator.

in that we have fixed types for the initial environment and result of each task: type `env` and type `result`, respectively. This is due to the fact that we cannot get at the compiled code or aggregate data with polymorphic types.

To start the simulation,

```
startSim : real -> (env -> result) * env -> result
```

is used. This is necessary as CML requires calling `RunCML.doit` before using any CML constructs. The expression `startSim` p (c, e) starts the simulator with code c and initial environment e as the initial task; parameter p specifies the frequency of failures to be induced by the simulator where $0.0 \leq p < 1.0$.

2.5.2 Implementation

In this implementation, we use CML threads to model ConCert tasks (not ConCert cords). This is due to an issue with using `callcc/throw` with CML, which does make the simulation slightly less realistic but seems reasonable. There are two kinds of CML threads running during the simulation: the threads that model ConCert tasks and the threads that are used for bookkeeping (e.g. ID servers, lock servers, and failure simulation).

The state of the simulation is managed by two global tables along with the appropriate locking mechanisms: one that maps tasks to attributes about the state of the task and one that maps CML thread IDs to tasks. This is clearly a bit unrealistic as this state must be kept implicitly on the network via some protocol and only local state on each host (e.g. work queues, etc.).

In the simulator, a task handle is simply a uniquely generated integer that indexes into the global task state table. This differs from the ConCert framework where we view task handles as a hash or digest of the code and the environment, but since we cannot get at the compiled code itself within SML/CML, we are unable to generate such a digest in the simulator. Dependencies amongst tasks are modeled using CML `ivars` [Rep99], which are write-once communication channels. This can be seen in the following fragment of `functor CCTasksFn` that implements the signature given in figure 3:

```
functor CCTasksFn(structure T : sig
                                  type env
                                  type result
                                end)
:> CCTASKS where type env = T.env and type result = T.result =
struct
  type env = T.env
  type result = T.result

  type task = int                        (* unique identifier *)

  structure TaskState :> ... =
  struct
    open SyncVar
    type taskState =
    { tid      : CML.thread_id option,   (* id of the thread that
                                            represents the task   *)
      status   : CCStatus.status,        (* current status        *)
      closure  : (env -> result) * env,  (* the code and the env  *)
      out      : result Msg ivar option  (* where to send the ans *)
    }
    ...
  end

  ...
end
```

Note that `taskState.out` is the only connection to an active task. It is an option type, for when a task fails, this field is updated to `NONE` to sever the link. According to [Rep99], CML threads are garbage collected if they are no longer reachable by any communication channel. This is vital for us to ensure that a "failed" task ends since CML does not provide any explicit *kill* primitive on threads.

Failure simulation is done by a thread that periodically tells the scheduler to `forget` a task. This marks the status of a task as `Failed` and issues a `FAILED` message. In other words, the simulator informs itself when a "failure" occurs. This is another place that is unrealistic as it side-steps the issue of failure detection with which the `ConCert` software must deal. Concerning the scheduling tasks in the simulator, we simply rely on the CML thread scheduler.

2.5.3 Limitations

The most obvious shortcoming of the simulator is the fixing of type `env` and type `result` as described above. This cannot be avoided without developing a compiler for at least core ML plus the proposed ConCert extensions for tasks. It turns out not to be a severe limitation as the developer can create a datatype that wraps the types of interest and dispatch based on the tag (at the cost of a runtime check). For example, suppose the developer wants to use two kinds of tasks in one application: one that returns a value of type `real` and one that returns a value of type `int`, here is a possible definition for the argument to `CCTaskFn`.

```
structure T =
struct
  type env = (* ... the desired environment type ... *)
  datatype result = Int of int | Real of real
end
```

Another limitation is that we have no way to enforce that the code given to `inject` is closed (even at runtime) as per our assumption. Similarly, since the simulator is written in CML, we have no way to ensure that data used in a program, which runs on the simulator, will be marshalable. This is partly due to our incomplete understanding about how to best implement marshaling. It is an open question whether it is possible to design a type system to statically verify the ability of a piece of data to be marshaled across the network. Lastly, global effects visible to all tasks can be incurred by individual tasks. For example, the update of a reference cell affects the memory of every task. As it stands today, enforcing these requirements are left to the programmer's discipline, which is certainly less than ideal.

3 Parallel Theorem Proving for Linear Logic

As mentioned before, we hope to utilize the development process of a substantial application to guide the design of the initial ConCert framework and programming model. In this section, we talk in more detail about the design and implementation of such an application, namely a parallel theorem prover for linear logic (lktara). We discuss the motivation for choosing this application, the core algorithmic ideas, and the parallelization of the algorithm.

3.1 Why a Theorem Prover Application?

We have chosen to build a bottom-up or subgoal reduction-based parallel theorem prover for linear logic for a number of reasons. The primary reason is that this application should afford a different set of requirements compared to traditional grid applications in that the parallel algorithm will have non-trivial depth (similar to game-tree search) and the computation will be largely symbolic rather than numeric. We would like to explore whether such a space of applications can be made feasible on the grid. Secondarily, another advantage of this particular application is that we can temporarily ignore the issue of a malicious resource donor returning bogus answers. This is because the theorem prover application can return a proof of the theorem in question, which can then be verified easily. Lastly, there are few implementations of theorem provers for linear logic, so developing one would be a contribution in itself.

Linear logic, as first formulated by Girard [Gir87], has been considered a refinement of classical logic to incorporate reasoning about state. At the same time, there are a number of applications in logic and functional programming that are best expressed in an intuitionistic logic. We seek to develop a theorem prover for a fragment of intuitionistic linear logic as described in [Pfe01].

3.2 Bottom-Up Proof Search for Intuitionistic Linear Logic

For the sake of brevity, we assume the reader is familiar with Gentzen-style sequent calculi, which will form the basis for our proof search algorithm. A thorough treatment of the sequent calculus for intuitionistic linear logic is given in [Pfe01]. The fragment of linear logic we shall consider is the propositional fragment given by the following grammar:

$$\begin{array}{lllllll}
\text{Propositions} & A ::= P & & & & & \text{Atoms} \\
 & \mid A_1 \multimap A_2 & \mid A_1 \otimes A_2 & \mid \mathbf{1} & & & \text{Multiplicatives} \\
 & & \mid A_1 \mathbin{\&} A_2 & \mid \top & \mid A_1 \oplus A_2 & \mid \mathbf{0} & \text{Additives} \\
 & \mid A \supset B & \mid\ !A & & & & \text{Exponentials}
\end{array}$$

where P stands for the syntactic category of atomic formulas. We write the basic sequent as follows:

$$\Gamma; \Delta \Longrightarrow A$$

where Γ contains unrestricted hypotheses and Δ contains linear hypotheses. Γ and Δ are considered unordered collections (i.e. multisets). The formulas in Γ are regarded as being preceded by the modal operator !, so the form above corresponds to the more traditional linear logic sequent $!\Gamma, \Delta \Longrightarrow A$.

The literature is not in complete agreement on terminology, but we refer to *bottom-up* proof search as creating a derivation for a given judgment by working upwards. At any point in the algorithm, we have a partial derivation with subgoals yet to be shown at the top. At a high-level, we can summarize our algorithm as follows. We select a judgment that has yet to be derived and an inference rule by which that judgment could be inferred. There are some instances where we must make a choice on how a rule is to be applied. For example, we must choose how to split the hypotheses in the multiplicative conjunction right rule ($\otimes$R) as shown below.

$$\frac{\Gamma; \Delta_1 \Longrightarrow A \qquad \Gamma; \Delta_2 \Longrightarrow B}{\Gamma; (\Delta_1, \Delta_2) \Longrightarrow A \otimes B} \otimes\mathrm{R}$$

The premises of this chosen rule yield new judgments that remain to be derived. This idea forms the basis of our algorithm, but the general formulation of the sequent calculus is highly non-deterministic making proof search based solely on this infeasible.

The first observation to remedy this is that some rules are *invertible* meaning the premises are derivable whenever the conclusion is. Invertible rules can be applied whenever possible without losing completeness. A classification of the invertible and non-invertible rules is given in [Pfe01]. We also eliminate some non-determinism by restricting our attention to propositional linear logic (although extending to first-order logic should be straightforward using standard unification techniques).

3.2.1 Focusing

Extending the observation about invertible rules, we utilize a refinement of the sequent calculus to further reduce the non-determinism while remaining sound and complete. This refinement, known as *focusing*, was first presented in Andreoli's seminal paper [And92] for classical linear logic. A thorough treatment of focusing for intuitionistic linear logic has been presented in Pfenning's course notes [Pfe01] and Howe's Ph.D. thesis [How98]. Pfenning's system is a superset of the system we present here with the addition of the quantifiers ($\forall, \exists$), while Howe's system is very similar except that he does not consider the unrestricted implication ($\supset$).

To describe the focusing system, we classify rules into *strongly invertible*, *weakly invertible*, and *non-invertible* where *weakly invertible* rules are invertible but have some constraint on the linear context. We will also use the following definitions to separate propositions into distinct classes, which are based on the terms first introduced by Andreoli [And92].

Definition 1 (Classification of Propositions)

- *A is* right asynchronous *if the right rule of its top-level connective is strongly invertible.*

Right Asynchronous Phase.

$$\frac{\Gamma; \Delta; (\Omega, A) \Longrightarrow B \Uparrow}{\Gamma; \Delta; \Omega \Longrightarrow A \multimap B \Uparrow}\multimap\mathrm{R} \qquad \frac{(\Gamma, A); \Delta \Longrightarrow B \Uparrow}{\Gamma; \Delta \Longrightarrow A \supset B \Uparrow}\supset\mathrm{R} \qquad \frac{}{\Gamma; \Delta; \Omega \Longrightarrow \top \Uparrow}\top\mathrm{R}$$

$$\frac{\Gamma; \Delta; \Omega \Longrightarrow A \Uparrow \qquad \Gamma; \Delta; \Omega \Longrightarrow B \Uparrow}{\Gamma; \Delta; \Omega \Longrightarrow A \mathbin{\&} B \Uparrow}\mathbin{\&}\mathrm{R} \qquad \frac{\Gamma; \Delta; \Omega \Uparrow\Longrightarrow C \qquad C \text{ not right asynchronous}}{\Gamma; \Delta; \Omega \Longrightarrow C \Uparrow}\Uparrow\mathrm{R}$$

Left Asynchronous Phase.

$$\frac{\Gamma; \Delta; (\Omega, A, B) \Uparrow\Longrightarrow C}{\Gamma; \Delta; (\Omega, A \otimes B) \Uparrow\Longrightarrow C}\otimes\mathrm{L} \qquad \frac{\Gamma; \Delta; \Omega \Uparrow\Longrightarrow C}{\Gamma; \Delta; (\Omega, \mathbf{1}) \Uparrow\Longrightarrow C}\mathbf{1}\mathrm{L}$$

$$\frac{\Gamma; \Delta; (\Omega, A) \Uparrow\Longrightarrow C \qquad \Gamma; \Delta; (\Omega, B) \Uparrow\Longrightarrow C}{\Gamma; \Delta; (\Omega, A \oplus B) \Uparrow\Longrightarrow C}\oplus\mathrm{L} \qquad \frac{}{\Gamma; \Delta; (\Omega, \mathbf{0}) \Uparrow\Longrightarrow C}\mathbf{0}\mathrm{L}$$

$$\frac{(\Gamma, A); \Delta; \Omega \Uparrow\Longrightarrow C}{\Gamma; \Delta; (\Omega, !A) \Uparrow\Longrightarrow C}!\mathrm{L} \qquad \frac{\Gamma; (\Delta, A); \Omega \Uparrow\Longrightarrow C \qquad A \text{ not left asynchronous}}{\Gamma; \Delta; (\Omega, A) \Uparrow\Longrightarrow C}\Uparrow\mathrm{L}$$

Figure 4: Focusing Calculus - Asynchronous Phases

- *A is* left asynchronous *if the left rule of its top-level connective is strongly invertible.*
- *A is* right synchronous *if the right rule of its top-level connective is non-invertible or only weakly invertible.*
- *A is* left synchronous *if the left rule of its top-level connective is non-invertible or only weakly invertible.*

This yields the following classification table:

Atomic	P
Right Asynchronous	$A_1 \multimap A_2, A_1 \mathbin{\&} A_2, \top, A_1 \supset A_2$
Left Asynchronous	$A_1 \otimes A_2, A_1 \oplus A_2, \mathbf{0}, !A$
Right Synchronous	$A_1 \otimes A_2, A_1 \oplus A_2, \mathbf{0}, !A$
Left Synchronous	$A_1 \multimap A_2, A_1 \mathbin{\&} A_2, \top, A_1 \supset A_2$

Focusing can be broken down into two main phases, each with two subcategories—whether we are working on the left or the right. The first phase breaks down all the asynchronous connectives. Since the order in which the rules are applied does not matter, we fix the order by first breaking down the goal and then the hypotheses from right to left. This eliminates much of the "don't-care non-determinism" in the sequent calculus, which corresponds to

Decision.

$$\dfrac{\Gamma; \Delta \Longrightarrow C \Downarrow \qquad C \text{ not atomic}}{\Gamma; \Delta; \cdot \Uparrow\Longrightarrow C}\,\mathsf{decideR} \qquad \dfrac{\Gamma; \Delta; A \Downarrow\Longrightarrow C}{\Gamma; (\Delta, A); \cdot \Uparrow\Longrightarrow C}\,\mathsf{decideL} \qquad \dfrac{(\Gamma, A); \Delta; A \Downarrow\Longrightarrow C}{(\Gamma, A); \Delta; \cdot \Uparrow\Longrightarrow C}\,\mathsf{decideL!}$$

Right Focusing Phase.

$$\dfrac{\Gamma; \Delta_1 \Longrightarrow A \Downarrow \qquad \Gamma; \Delta_2 \Longrightarrow B \Downarrow}{\Gamma; (\Delta_1, \Delta_2) \Longrightarrow A \otimes B \Downarrow}\,\otimes\mathrm{R} \qquad \dfrac{}{\Gamma; \cdot \Longrightarrow \mathbf{1} \Downarrow}\,\mathbf{1}\mathrm{R}$$

$$\dfrac{\Gamma; \Delta \Longrightarrow A \Downarrow}{\Gamma; \Delta \Longrightarrow A \oplus B \Downarrow}\,\oplus\mathrm{R}_1 \qquad \dfrac{\Gamma; \Delta \Longrightarrow B \Downarrow}{\Gamma; \Delta \Longrightarrow A \oplus B \Downarrow}\,\oplus\mathrm{R}_2 \qquad \text{No } \mathbf{0} \text{ right rule}$$

$$\dfrac{\Gamma; \cdot; \cdot \Longrightarrow A \Uparrow}{\Gamma; \cdot \Longrightarrow !A \Downarrow}\,!\mathrm{R} \qquad \dfrac{\Gamma; \Delta; \cdot \Longrightarrow A \Uparrow \qquad A \text{ not right synchronous}}{\Gamma; \Delta \Longrightarrow A \Downarrow}\,\Downarrow\mathrm{R}$$

Left Focusing Phase.

$$\dfrac{\Gamma; \Delta_2; B \Downarrow\Longrightarrow C \qquad \Gamma; \Delta_1 \Longrightarrow A \Downarrow}{\Gamma; (\Delta_1, \Delta_2); A \multimap B \Downarrow\Longrightarrow C}\,\multimap\mathrm{L} \qquad \dfrac{\Gamma; \Delta; B \Downarrow\Longrightarrow C \qquad \Gamma; \cdot \Longrightarrow A \Downarrow}{\Gamma; \Delta; A \supset B \Downarrow\Longrightarrow C}\,\supset\mathrm{L}$$

$$\dfrac{\Gamma; \Delta; A \Downarrow\Longrightarrow C}{\Gamma; \Delta; A \,\&\, B \Downarrow\Longrightarrow C}\,\&\mathrm{L}_1 \qquad \dfrac{\Gamma; \Delta; B \Downarrow\Longrightarrow C}{\Gamma; \Delta; A \,\&\, B \Downarrow\Longrightarrow C}\,\&\mathrm{L}_1 \qquad \text{No } \top \text{ left rule}$$

$$\dfrac{}{\Gamma; \cdot; A \Downarrow\Longrightarrow A}\,\mathsf{init} \qquad \dfrac{\Gamma; \Delta; A \Uparrow\Longrightarrow C \qquad A \text{ not atomic and not left synchronous}}{\Gamma; \Delta; A \Downarrow\Longrightarrow C}\,\Downarrow\mathrm{L}$$

Figure 5: Focusing Calculus - Decision and Focusing Phases

our observation of invertible rules. The second phase is the focusing phase where a synchronous proposition must be selected to be processed. Once the synchronous proposition is selected, it must be broken down until we reach an atomic or an asynchronous proposition.

Given this, we now have the following four judgments for each phase:

Right Asynchronous	$\Gamma; \Delta; \Omega \Longrightarrow A \Uparrow$
Left Asynchronous	$\Gamma; \Delta; \Omega \Uparrow\Longrightarrow C$
Right Focusing	$\Gamma; \Delta \Longrightarrow A \Downarrow$
Left Focusing	$\Gamma; \Delta; A \Downarrow\Longrightarrow C$

where

Γ are unrestricted hypotheses and may contain arbitrary propositions
Δ are linear hypotheses and must contain only non-left asynchronous propositions
Ω are ordered linear hypotheses and may contain arbitrary propositions
A is an arbitrary proposition
C is a proposition that may not be right asynchronous

Figures 4 and 5 give the complete focusing calculus that describes the core proof search algorithm recalling that proof search is viewed as proceeding bottom-up. One can find soundness and completeness proofs of the focusing calculus with respect to the ordinary sequent calculus for intuitionistic linear logic in [How98].

3.2.2 Resource Management

Although focusing makes proof search much more feasible, a large source of non-determinism remains in choosing the splitting of hypotheses in the multiplicative rules (such as $\otimes$R). It is simply unacceptable to "try" each possible splitting. One possible solution is to try to use an *input/output model* of resource management due to Hodas and Miller [HM94] and also described in [CHP00] and §5.3 of [Pfe01]. However, this only works if we have chosen to serialize proving subgoals in say a left-to-right order. Since we may seek to parallelize this work, we choose a different method. We choose to adapt the use of boolean constraints for resource distribution as described by Harland and Pym [HP01] using ordered binary decision diagrams (OBDDs) [Bry86, Bry92] to implement these boolean constraints.

Harland and Pym consider resource management for classical linear logic in [HP01]. We seek to extend this notion to the intuitionistic case. It should be noted that this notion of resource management is orthogonal to focusing as discussed above and can be considered using the ordinary sequent calculus. However, for our purposes, we will build upon the focusing calculus described in the previous section. As in [HP01], we do not require arbitrary boolean expressions but only those given in the following grammar:

$$\begin{array}{rccl} \text{Boolean Expressions} & e & ::= & 0 \mid 1 \mid e \cdot x \mid e \cdot \overline{x} \\ \text{Constraints} & c & ::= & e = 0 \mid e = 1 \mid c_1 \wedge c_2 \mid \top \end{array}$$

where x is a boolean variable, $\cdot$ is boolean multiplication, and $\overline{x}$ is the boolean complement of x. We now annotate linear hypotheses A with a boolean expression e as $A[e]$. We write $exp(A)$ for the boolean expression associated with A where appropriate. The linear contexts Δ and Ω now contain these annotated formulas as described below:

$$\begin{array}{rccl} \text{Unrestricted Contexts} & \Gamma & ::= & \cdot \mid \Gamma, A \\ \text{Linear Contexts} & \Delta & ::= & \cdot \mid \Delta, A[e] \\ \text{Ordered Linear Contexts} & \Omega & ::= & \cdot \mid \Omega, A[e] \end{array}$$

where Γ and Δ are considered unordered and Ω is ordered as before. We also lift $exp(-)$ over contexts, so $exp(\Delta)$ yields the multiset of boolean expressions associated with the formulas in Δ.

Right Asynchronous Phase.

$$\frac{\Gamma; \Delta; (\Omega, A[1]) \overset{RM}{\Longrightarrow} B \Uparrow}{\Gamma; \Delta; \Omega \overset{RM}{\Longrightarrow} A \multimap B \Uparrow} \multimap\mathrm{R} \qquad \frac{(\Gamma, A); \Delta \overset{RM}{\Longrightarrow} B \Uparrow}{\Gamma; \Delta \overset{RM}{\Longrightarrow} A \supset B \Uparrow} \supset\mathrm{R} \qquad \frac{}{\Gamma; \Delta; \Omega \overset{RM}{\Longrightarrow} \top \Uparrow} \top\mathrm{R}$$

$$\frac{\Gamma; \Delta; \Omega \overset{RM}{\Longrightarrow} A \Uparrow \qquad \Gamma; \Delta; \Omega \overset{RM}{\Longrightarrow} B \Uparrow}{\Gamma; \Delta; \Omega \overset{RM}{\Longrightarrow} A \mathbin{\&} B \Uparrow} \&\mathrm{R} \qquad \frac{\Gamma; \Delta; \Omega \Uparrow\overset{RM}{\Longrightarrow} C \qquad C \text{ not right asynchronous}}{\Gamma; \Delta; \Omega \overset{RM}{\Longrightarrow} C \Uparrow} \Uparrow\mathrm{R}$$

Left Asynchronous Phase.

$$\frac{\Gamma; \Delta; (\Omega, A[1], B[1]) \Uparrow\overset{RM}{\Longrightarrow} C \qquad e = 1}{\Gamma; \Delta; (\Omega, (A \otimes B)[e]) \Uparrow\overset{RM}{\Longrightarrow} C} \otimes\mathrm{L} \qquad \frac{\Gamma; \Delta; \Omega \Uparrow\overset{RM}{\Longrightarrow} C \qquad e = 1}{\Gamma; \Delta; (\Omega, \mathbf{1}[e]) \Uparrow\overset{RM}{\Longrightarrow} C} \mathbf{1}\mathrm{L}$$

$$\frac{\Gamma; \Delta; (\Omega, A[1]) \Uparrow\overset{RM}{\Longrightarrow} C \quad \Gamma; \Delta; (\Omega, B[1]) \Uparrow\overset{RM}{\Longrightarrow} C \quad e = 1}{\Gamma; \Delta; (\Omega, (A \oplus B)[e]) \Uparrow\overset{RM}{\Longrightarrow} C} \oplus\mathrm{L} \qquad \frac{e = 1}{\Gamma; \Delta; (\Omega, \mathbf{0}[e]) \Uparrow\overset{RM}{\Longrightarrow} C} \mathbf{0}\mathrm{L}$$

$$\frac{(\Gamma, A); \Delta; \Omega \Uparrow\overset{RM}{\Longrightarrow} C \qquad e = 1}{\Gamma; \Delta; (\Omega, !A[e]) \Uparrow\overset{RM}{\Longrightarrow} C} !\mathrm{L} \qquad \frac{\Gamma; (\Delta, A[1]); \Omega \Uparrow\overset{RM}{\Longrightarrow} C \quad A \text{ not left asynchronous} \quad e = 1}{\Gamma; \Delta; (\Omega, A[e]) \Uparrow\overset{RM}{\Longrightarrow} C} \Uparrow\mathrm{L}$$

Figure 6: Focusing with Resource Management Calculus - Asynchronous Phases

We interpret the focused resource sequents

$$\begin{array}{l} \Gamma; A_1[e_1], \ldots, A_m[e_m]; A_{m+1}[e_{m+1}], \ldots, A_n[e_n] \overset{RM}{\Longrightarrow} C \Uparrow \\ \Gamma; A_1[e_1], \ldots, A_m[e_m]; A_{m+1}[e_{m+1}], \ldots, A_n[e_n] \Uparrow\overset{RM}{\Longrightarrow} C \\ \Gamma; A_1[e_1], \ldots, A_n[e_n] \overset{RM}{\Longrightarrow} C \Downarrow \\ \Gamma; A_1[e_1], \ldots, A_n[e_n]; A \Downarrow\overset{RM}{\Longrightarrow} C \end{array}$$

as saying that we have resource A_i if $e_i = 1$ or we do not have resource A_i if $e_i = 0$ for $i = 1 \ldots n$. We can also view the focus formula A as implicitly annotated with the boolean expression 1.

We now describe the changes to the focusing calculus to incorporate this style of resource management. Loosely speaking, boolean constraints are introduced by the multiplicative rules as in $\otimes$R:

$$\frac{\Gamma; \Delta \cdot V \overset{RM}{\Longrightarrow} A \Downarrow \qquad \Gamma; \Delta \cdot \overline{V} \overset{RM}{\Longrightarrow} B \Downarrow}{\Gamma; \Delta \overset{RM}{\Longrightarrow} A \otimes B \Downarrow} \otimes\mathrm{R}$$

where $\Delta = A_1[e_1], A_2[e_2], \ldots, A_n[e_n]$, $V = \{x_1, x_2, \ldots, x_n\}$ is a set of n new boolean variables, and $\overline{V}$ is the set of the boolean expressions $\{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$.

Decision.

$$\frac{\Gamma;\Delta \overset{RM}{\Longrightarrow} C \Downarrow \quad C \text{ not atomic} \quad \forall e \in \mathit{exp}(\Omega).e = 0}{\Gamma;\Delta;\Omega \Uparrow\overset{RM}{\Longrightarrow} C}\,\textsf{decideR}$$

$$\frac{\Gamma;\Delta;A \Downarrow\overset{RM}{\Longrightarrow} C \quad e = 1 \quad \forall e' \in \mathit{exp}(\Omega).e' = 0}{\Gamma;(\Delta, A[e]);\Omega \Uparrow\overset{RM}{\Longrightarrow} C}\,\textsf{decideL} \qquad \frac{(\Gamma, A);\Delta;A \Downarrow\overset{RM}{\Longrightarrow} C \quad \forall e \in \mathit{exp}(\Omega).e = 0}{(\Gamma, A);\Delta;\Omega \Uparrow\overset{RM}{\Longrightarrow} C}\,\textsf{decideL!}$$

Right Focusing Phase.

$$\frac{\Gamma;\Delta \cdot V \overset{RM}{\Longrightarrow} A \Downarrow \qquad \Gamma;\Delta \cdot \overline{V} \overset{RM}{\Longrightarrow} B \Downarrow}{\Gamma;\Delta \overset{RM}{\Longrightarrow} A \otimes B \Downarrow}\otimes\mathrm{R} \qquad \frac{\forall e \in \mathit{exp}(\Delta).e = 0}{\Gamma;\Delta \overset{RM}{\Longrightarrow} \mathbf{1} \Downarrow}\mathbf{1}\mathrm{R}$$

$$\frac{\Gamma;\Delta \overset{RM}{\Longrightarrow} A \Downarrow}{\Gamma;\Delta \overset{RM}{\Longrightarrow} A \oplus B \Downarrow}\oplus\mathrm{R}_1 \qquad \frac{\Gamma;\Delta \overset{RM}{\Longrightarrow} B \Downarrow}{\Gamma;\Delta \overset{RM}{\Longrightarrow} A \oplus B \Downarrow}\oplus\mathrm{R}_2 \qquad \text{No } \mathbf{0} \text{ right rule}$$

$$\frac{\Gamma;\cdot;\cdot \overset{RM}{\Longrightarrow} A \Uparrow \qquad \forall e \in \mathit{exp}(\Delta).e = 0}{\Gamma;\Delta \overset{RM}{\Longrightarrow} !A \Downarrow}!\mathrm{R} \qquad \frac{\Gamma;\Delta;\cdot \overset{RM}{\Longrightarrow} A \Uparrow \qquad A \text{ not right synchronous}}{\Gamma;\Delta \overset{RM}{\Longrightarrow} A \Downarrow}\Downarrow\mathrm{R}$$

Left Focusing Phase.

$$\frac{\Gamma;\Delta \cdot V;B \Downarrow\overset{RM}{\Longrightarrow} C \qquad \Gamma;\Delta \cdot \overline{V} \overset{RM}{\Longrightarrow} A \Downarrow}{\Gamma;\Delta;A \multimap B \Downarrow\overset{RM}{\Longrightarrow} C}\multimap\mathrm{L} \qquad \frac{\Gamma;\Delta;B \Downarrow\overset{RM}{\Longrightarrow} C \qquad \Gamma;\cdot \overset{RM}{\Longrightarrow} A \Downarrow}{\Gamma;\Delta;A \supset B \Downarrow\overset{RM}{\Longrightarrow} C}\supset\mathrm{L}$$

$$\frac{\Gamma;\Delta;A \Downarrow\overset{RM}{\Longrightarrow} C}{\Gamma;\Delta;A \,\&\, B \Downarrow\overset{RM}{\Longrightarrow} C}\&\mathrm{L}_1 \qquad \frac{\Gamma;\Delta;B \Downarrow\overset{RM}{\Longrightarrow} C}{\Gamma;\Delta;A \,\&\, B \Downarrow\overset{RM}{\Longrightarrow} C}\&\mathrm{L}_1 \qquad \text{No } \top \text{ left rule}$$

$$\frac{\forall e \in \mathit{exp}(\Delta).e = 0}{\Gamma;\Delta;A \Downarrow\overset{RM}{\Longrightarrow} A}\,\textsf{init} \qquad \frac{\Gamma;\Delta;A[1] \Uparrow\overset{RM}{\Longrightarrow} C \qquad A \text{ not atomic and not left synchronous}}{\Gamma;\Delta;A \Downarrow\overset{RM}{\Longrightarrow} C}\Downarrow\mathrm{L}$$

Figure 7: Focusing with Resource Management Calculus - Decision and Focusing Phases

We define $\Delta \cdot V$ to be $A_1[e_1 \cdot x_1], A_2[e_2 \cdot x_2], \ldots, A_n[e_n \cdot x_n]$ and $\Delta \cdot \overline{V}$ to be $A_1[e_1 \cdot \overline{x_1}], A_2[e_2 \cdot \overline{x_2}], \ldots, A_n[e_n \cdot \overline{x_n}]$.

For each left rule, we maintain that the principal formula must be constrained to 1 so that the application of the rule will contribute to the solution of the boolean equation. At the initial sequent,

$$\frac{\forall e \in \mathit{exp}(\Delta).e = 0}{\Gamma;\Delta;A \Downarrow\overset{RM}{\Longrightarrow} A}\,\textsf{init}$$

we ensure that only the focus formula is a usable resource.

Figures 6 and 7 give the complete focusing calculus with resource management. We view a derivation as carrying constraints c where each statement of the constraint $e = 0$ or $e = 1$ constrains c such that $c \leftarrow c \wedge (e = 0)$ or

$c \leftarrow c \wedge (e = 1)$, respectively. In proof search, we begin with the initial constraint $\top$ and fail whenever the constraints become unsatisfiable.

3.3 Parallelization

Examining figures 4 and 5, we can divide the possible parallelism into two categories: AND-parallelism and OR-parallelism. AND-parallelism can be found when we generate a set of new subgoals that each need to be solved to succeed. The order in which the subgoals are solved is irrelevant, so we can try to solve the subgoals in parallel. An example of such a rule is $\&$R (figure 4). OR-parallelism arises when there are several rules that can be applied (or several ways we can apply a given rule) where solving one of the possible subgoals would enable us to succeed. A simple example of this is the choice between applying $\oplus\mathrm{R}_1$ or $\oplus\mathrm{R}_2$ (figure 5).

The next critical observation is that the focusing strategy concentrates much of the non-determinism into the "decision" phase (the $\mathsf{decideR}, \mathsf{decideL}$, and $\mathsf{decideL!}$ rules) so that there are several OR-parallel branches at this point (depending on the size of the contexts). We presume that this is the place we can get the most benefit from the grid architecture. Other places where we could introduce parallelism grid may or may not be beneficial on the grid. An option is to also parallelize using local threads.

Given these observations, we see that once we spawn tasks on the grid for OR-parallel branches, once some task succeeds, we may want to terminate the other tasks to save resources on the network since they are no longer needed. This motivated the need for `forget` primitive in our proposed language extension.

A challenge with parallelizing for the grid such a linear logic prover is dealing with resource distribution as discussed in section 3.2.2. One possible parallelization strategy is to have the subtasks communicate with each other as they use the resources, but this seems infeasible in the grid setting as there should be low communication between tasks. However, using boolean constraints, each subgoal can be solved independently returning the resources that were used. We then check that the resource usage is consistent. One issue that arises if we pursue this further is that there may be multiple proofs for the same goal using different resources. One usage of resources may prevent the other branch from succeeding. Furthermore, we do not have the ability to tell the subtask which usage of resources will be acceptable *a priori*, so we must consider all possible proofs of the sub-branch. To implement this, we would like to think of each task returning multiple results or more specifically, a stream of results. One option for implementation would be to return code representing a suspended task that gives the next result if asked. Another option is to use disabled tasks, which are introduced in section 2.2.

Modeling result streams is what we currently envision as the primary use of disabled tasks. As described above, it appears that a substantial amount of code will be shipped back and forth using the first option. An alternative (with different drawbacks) would be for the subtask to inject a disabled task that would compute the next result and return a much smaller piece of code that would enable this disabled task if the next result in the stream is needed. Though this method may reduce the amount of transmitted code, it has the potential to add a substantial amount of bookkeeping to the underlying code distribution and verification software (ConCert).

3.4 Implementation

We began the development of Iktara with a purely sequential version written in SML [MTHM97] using the SML/NJ compiler. Then, we explored the parallelization of the theorem proving algorithm by developing a concurrent version using Concurrent ML (CML) [Rep99] that motivated much of the design of the programming model discussed in section 2. Finally, a version was written using the ConCert simulator described in section 2.5. The code for all three versions will be made available from the ConCert webpage [Con01a]. In this section, we describe the components that comprise the various versions of the theorem prover and how they fit together. Figure 8 diagrams the modules of which the theorem provers are comprised.

3.4.1 Ordered Binary Decision Diagrams

A standard implementation of ordered binary decision diagrams (OBDDs) [Bry86, Bry92] was written in SML roughly following Andersen's course notes on OBDDs [And97]. The key requirement for our implementation is that the data structure must be marshalable, for we will need to be able to propagate OBDDs across the ConCert network. This is facilitated by ensuring that the implementation adheres to a functional interface and that each OBDD can exist independently of any other one. In the OBDD signature below, we state that the boolean variables will simply be represented by `int`s and a set of binary boolean operators is given in `oper`. OBDDs are initially built from either `zero`, `one`, or `mkVar`. The function `mkVar` creates an OBDD that represents the boolean expression consisting of simply the given variable. To build more complex boolean expressions, one uses `not` and `apply`. The function `not` yields the OBDD that represents the negation of the given OBDD, while `apply` combines the two given OBDDs with the given binary operator.

```
signature OBDD =
sig
  type var = int
  datatype oper = And | Or | Imp | BiImp | Xor | Nand | Nor
```

```
    type obdd
    val zero      : obdd
    val one       : obdd
    val mkVar     : var -> obdd
    val not       : obdd -> obdd
    val apply     : oper -> obdd * obdd -> obdd
    val eq        : obdd * obdd -> bool
    val toString  : obdd -> string
  end
```

As described in [And97], nodes are represented by integers $0, 1, 2, \ldots$ with 0 and 1 reserved for the terminal nodes. We define the following to describe variable nodes:

$var(u)$	the variable name of node u
$low(u)$	the node that is at the end of the 0-branch of node u
$high(u)$	the node that is at the end of the 1-branch of node u

To represent OBDDs, we rely on two tables: $T : u \mapsto (i, l, h)$ that maps node u to the variable node attributes (i, l, h) where $var(u) = i$, $low(u) = l$, and $high(u) = h$. and $H : (i, l, h) \mapsto u$ that is inverse mapping of table T. The significant difference between the description given in [And97] and the implementation of `structure Obdd :> OBDD` is that rather than using a global table of nodes, each OBDD contains its own table of nodes. This is at the cost of memory as the nodes are no longer shared across OBDDs, but without this, we would certainly not be able to marshal this structure.

3.4.2 Boolean Constraints

Using OBDDs, we develop a module that provides operations for implementing the resource management system described in section 3.2.2. In the following fragment of `signature CONSTRAINTS`, we specify one abstract type for boolean expressions `bexp`, one abstract type for constraints `constraints`, and operations for resource management:

```
signature CONSTRAINTS =
sig
  structure Prop  : PROP
  structure Cxt   : CXT

  type bexp             (* boolean expressions *)
  val one  : bexp
  val zero : bexp

  type linearCxt       = (Prop.prop * bexp) Cxt.Cxt
  type unrestrictedCxt = Prop.prop Cxt.Cxt

  type constraints      (* boolean constraints *)

  val empty          : constraints
  val addConstraints : constraints * constraints -> constraints
  val constrainExp   : bexp * bool -> constraints

  val distributeRes  : linearCxt -> linearCxt * linearCxt

  val verifyEmpty    : linearCxt * constraints -> constraints option
  val unsatisfiable  : constraints -> bool

  ...
end
```

The boolean expressions e in section 3.2.2 are represented by values of type `bexp` and constraints c are represented by values of type `constraints`. Let $\ulcorner e \urcorner$ denote the value of type `bexp` that represents the boolean expression e, $\ulcorner c \urcorner$ denote the value of type `constraints` that represents the constraint c, and $\ulcorner \Delta \urcorner$ denote the value of type `linearCxt` that represents the linear context Δ. Then,

$$\begin{aligned}
\texttt{one} &= \ulcorner 1 \urcorner \\
\texttt{zero} &= \ulcorner 0 \urcorner \\
\\
\texttt{empty} &= \ulcorner \top \urcorner \\
\texttt{addConstraints}\,(c_1, c_2) &= \ulcorner c_1 \wedge c_2 \urcorner \\
\texttt{constrainExp}\,(e_1, \texttt{true}) &= \ulcorner e_1 = 1 \urcorner \\
\texttt{constrainExp}\,(e_1, \texttt{false}) &= \ulcorner e_1 = 0 \urcorner \\
\\
\texttt{distributeRes}\,\Delta &= (\ulcorner \Delta \cdot V \urcorner, \ulcorner \Delta \cdot \overline{V} \urcorner)
\end{aligned}$$

where $|\Delta| = n$, V is a set of new variables $\{x_1, x_2, \ldots, x_n\}$, and $\overline{V} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$. It is clear that `distributeRes` is used when applying $\otimes$R and $\multimap$L (figure 7). The function `verifyEmpty` is used to check the condition $\forall e \in \mathit{exp}(\Delta).e = 0$ in rule init:

$$\frac{\forall e \in \mathit{exp}(\Delta).e = 0}{\Gamma; \Delta; A \Downarrow\!\overset{RM}{\Longrightarrow} A}\,\mathsf{init}$$

Other places where this is used are in rules decideR, decideL, decideL!, **1**R, and !R. The function `unsatisfiable` c returns `true` if there exists no satisfying assignment for the boolean variables in c or `false` otherwise. The implementation of the signature described above is straightforward by using OBDDs for both boolean expressions and constraints.

3.4.3 Focusing

Since each version of the theorem prover will share much of the implementation of the focusing phases as described in section 3.2, we attempt to separate the core focusing phases from other parts of the theorem prover implementation. We also seek to implement isolate each phase from every other phase so that so phases can be re-implemented independently. The datatype `sequent` distinguishes the five focusing phases, which is used by the `dispatcher` to invoke the correct phase. The `functor FocusingFn(...) :> FOCUSING` simply implements the focusing phases as described in 3.2 in a purely sequential manner using SML with no additional extensions.

3.4.4 ConCert Simulator Version

To test to the programming interface discussed in section 2.2, we develop a version of the theorem prover for the ConCert simulator as described in section 2.5. We set up the simulator as follows fixing type `env` and type `result`:

```
(* ConCert Simulator Interface *)
structure Task =
struct
  type env = F.sequent * C.constraints
  datatype result =
    R of (C.constraints * ((env -> result) * env)) option
end
structure CCTasks = CCTasksFn(structure T = Task)
```

where `F :> FOCUSING` and `C :> CONSTRAINTS`. Type `env` specifies the sequent to start proving and the current set of constraints. Type `result` indicates whether or not it is successful; if it is successful, it gives the new constraints, which indicates which resources were consumed, and the code for a task to execute if the overall goal cannot be proven. Ideally, we would like `type result` to contain a suspended task

```
type result = (C.constraints * (unit -> CCTasks.task)) option,
```

but this is not possible as SML does not (yet) have mutually recursive modules.

As described in section 3.3, we inject a new task for each focus formula in the decision phase. Thus, we write an implementation for the decision phase specifically for the simulator that starts new subtasks and waits for the results. To gather the results, we use `relax` to continue proving as soon as we get a successful result. The other subtasks remain running as we may need their results if the overall goal cannot be accomplished. The following code snippet illustrates this use of `relax`:

```
let
  val tsks = (* ... start subtasks ... *)

  fun resultLoop (nil) = fc ()
    | resultLoop (ts)  =
      let
        open CCTasks
        val (Task.R result, ts') =  relax ts
      in
        case result of
          NONE => resultLoop ts'
        | SOME (c,next) =>
          let
            fun fc' _ = let val t = injectTask true next
                        in  resultLoop (t::ts') end
          in
            sc ((c,()), fc')
          end
      end
in
  resultLoop tsks
end
```

where `sc` and `fc` are the success and failure continuations, respectively.

It should be noted that in the current implementation, there is a little cheat in that the code that is "shipped-back" as part of a successful result is not closed (i.e. we rely on the SML compiler to build that closure). This could be remedied by modifying the type of the failure continuations and restructuring the code, but it is evident that some form of automation in closure converting code is necessary.

4 Conclusion

We have discussed the aim of the ConCert project and how the development process of an application, namely a theorem prover for linear logic, contributes to ConCert. In this paper, we have presented a high-level view of what we believe is a reasonable programming model for the ConCert framework, some considerations in mapping this programming model to the ConCert grid software, and an implementation of a simulator for the programming model. We have then described in more detail the theorem proving algorithm we have implemented in Iktara and how the algorithm can be parallelized. This then gives a basis to motivate certain design choices in our programming model. Finally,

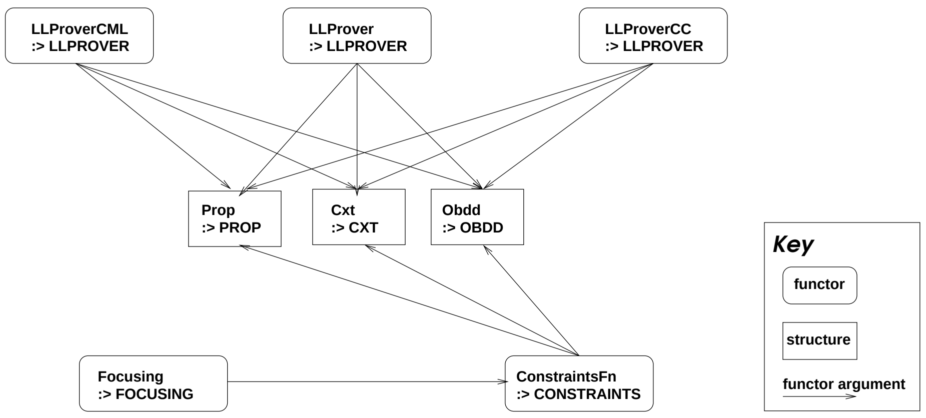

Figure 8: Iktara modules. The round-edged boxes denote functors, the sharp-edged boxes denote structures, and the arrows indicate some instantiation of the functors.

we have presented the actual implementation details of Iktara and how it uses the simulator.

In recent years, several workshops and conferences have emerged in the area of grid computing such as GRID [Lee01] and CCGrid [BMR01]. Similarly, large corporations have become involved with the development of grid architectures, such as the endorsement of the Globus project [Glo02] by IBM and the creation of the JXTA project [JXT01] by Sun Microsystems. The focus of such projects has been the creation of large-scale architectures and toolkits with security based on almost exclusively on authentication. Contrast this to ConCert's view on security that is based at least in part on verifiable properties of code rather than the identity of code producers and/or resource donors. A current project that shares a similar view to ConCert is the Mobile Resource Guarantees project taking place at the University of Edinburgh and Ludwig-Maximilians-Universität München [MRG02].

Future Work

In this section, we discuss some outstanding issues and some ideas for future work.

Programming Model

- Clearly, we need to implement a compiler for the proposed task programming interface.
- It is still open question what are the "right" semantics and the right

properties to be checked by the type system for tasks and the programming constructs proposed in section 2.2 . For example, should the usage of tasks be linear (i.e. should we ensure that a task is only `sync`ed on once)?

- Many open questions concerning the marshaling of data across the network and how that affects the programming model remain. How do we verify that or determine if a piece of data or code is marshalable? How much automation should there be in closure conversion for marshaling? Since communication is at a premium, presumably, the user would like to have precise control what gets sent across the network. How do we instruct the system to marshal modules? When do we cache data or code that has been transmitted across the network?

- In the interface described in section 2.2, we present two constructs `syncall` and `relax` that seem to have a conjunctive meaning. Should there be a disjunctive primitive? Note that a notion of disjunction can be implemented in terms of `relax` and `forget` as follows:

  ```
  (* select : 'r task list -> 'r *)
  fun select tl =
      let
        val (r,tl') = relax tl
      in
        app forget tl';
        r
      end
  ```

  Noting this, it seems clear that though the constructs we have proposed (`syncall` and `relax`) are motivated by pragmatic reasons as discussed in this paper, we should be able to decompose these into more primitive constructs with a possibly better logical interpretation. We have some initial ideas on this line, but they are still very preliminary.

- One might notice that in the description of our programming model, it appears that tasks are never explicitly destroyed. A failed task may be restarted if anyone desires its result. Implicit in our assumptions about the network has been that tasks have some "natural" lifespan in that due to the turbulent nature of the network, tasks will simply be forgotten by everyone. More realistically, tasks need to be garbage-collected in some manner. It is an open problem whether this can be done.

Simulator

- As discussed in section 2.5.3, there are limitations to the simulator. While some are unavoidable, we would still like to find a better simulation environment that relies less on programmers' discipline. One possibility may be to implement a simulator where a task or cord is modeled as a UNIX process where the communication is done solely through TCP/IP sockets.

In some sense, injecting a task into the network is more similar to forking a process than spawning a thread. This might more accurately model the network and would require proper marshaling of code/data in order for the program to work.

Theorem Prover

- As described in this paper, the theorem proving is not certifying in that it does not generate proof terms. This is essential because we need to have some method of ensuring that malicious resource donors are not giving bogus answers. Along this front, we also need develop a type checker for these proof terms.
- Currently, no front-end has been implemented for Iktara. Ideally, there should be some concrete syntax for inputting formulas and some method of displaying proofs.
- We would like to extend the theorem prover to first-order linear logic as it appears that standard unification techniques would apply and would likely enable us to feasibly prove some richer theorems.
- Most importantly, as we develop a compiler for the programming interface described in section 2.2 and the ConCert grid software matures, we would like have a version of the theorem prover running on the grid.

Acknowledgements

Foremost, I would like to thank my advisors Prof. Robert Harper and Prof. Frank Pfenning for their insightful advice and teaching me much of what I know today about logic and programming languages. I would also like to thank the rest of the ConCert framework team, Tom Murphy, Margaret DeLap, and Jason Liszka, for their comments and close collaboration. Also, I am grateful for the helpful comments provided by Prof. Pfenning, Prof. Harper, Tom, Margaret, and Mark Stehlik on preliminary drafts of this paper. Lastly, I would like to acknowledge the helpful suggestions provided by other members of the ConCert project.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[And97] Henrik Reif Andersen. An introduction to binary decision diagrams. Course Notes on the WWW, 1997. URL: http://www.itu.dk/people/hra/bdd97.ps.

[BL97] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, 1997.

[BMR01] Rajkumar Buyya, George Mohay, and Paul Roe, editors. *Cluster Computing and the Grid - CCGrid 2001*. IEEE Press, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000.

[Con01a] ConCert. Certified code for grid computing, project webpage, 2001. URL: http://www.cs.cmu.edu/˜concert.

[Con01b] ConCert. Project description, 2001. URL: http://www.cs.cmu.edu/˜concert/papers/proposal/proposal.pdf.

[DeL02] Margaret DeLap. Implementing a framework for certified grid computing. To appear as a technical report, 2002. Undergraduate thesis.

[Fig00] FightAIDS@home, 2000. URL: http://www.fightaidsathome.org.

[GIM96] GIMPS. The great internet mersenne prime search, 1996. URL: http://www.mersenne.org.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Glo02] Globus. The Globus project, 2002. URL: http://www.globus.org.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems*, pages 7(4):501–538, October 1985.

[Har02] Robert Harper. Programming languages: Theory and practice. Course Notes on the WWW, 2002. URL: http://www.cs.cmu.edu/˜rwh/plbook/.

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994. Extended abstract appeared in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15-18, 1991.

[How98] Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics.* PhD thesis, University of St. Andrews, Scotland, 1998.

[HP01] James Harland and David Pym. Resource-distribution via boolean constraints. To appear in *ACM Transactions on Computational Logic*, October 2001.

[JXT01] JXTA. Project JXTA, 2001. URL: http://www.jxta.org.

[Lee01] Craig A. Lee, editor. *Grid Computing - GRID 2001.* Number 2242 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2001.

[MRG02] MRG. Mobile resource guarantees, project webpage, 2002. URL: http://www.dcs.ed.ac.uk/home/dts/mrg/.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Pfe01] Frank Pfenning. Linear logic. Course Notes on the WWW, 2001. URL: http://www.cs.cmu.edu/~fp/courses/linear/linear.ps.

[Rep99] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[SET01] SETI@home. The search for extraterrestrial intelligence, 2001. URL: http://setiathome.ssl.berkeley.edu.

A Dynamic Semantics of Tasks

In this section, we give dynamic semantics for the programming model described in section 2.2 to aid in the understanding of the proposed language extensions. We elide the static semantics (i.e. the type system) because we do not (yet) assert anything stronger than what is suggested by the signature given in figure 1. To specify the dynamic semantics, we use structured operational semantics (SOS) in the style presented in Harper's course notes [Har02]. To present a reasonably realistic account in that some surface-level issues related to marshaling are drawn out, we present this as an extension to an effect-free core of ML plus references.

The language of expressions are extended as follows:

Expressions	e	$::=$	$\ldots$	
		$\mid$	l	locations
		$\mid$	$\texttt{ref}(e)$	create new cell
		$\mid$	$\texttt{!}e$	dereference
		$\mid$	$e_1 \texttt{ := } e_2$	update cell
		$\mid$	d_t	result destination for task t
		$\mid$	$\texttt{inject}\,(e_1, e_2)$	inject task
		$\mid$	$\texttt{enable}\ e$	enable task
		$\mid$	$\texttt{sync}\ e$	sync task
		$\mid$	$\texttt{syncall}\ e$	sync all tasks
		$\mid$	$\texttt{relax}\ e$	relax on tasks
		$\mid$	$\texttt{forget}\ e$	forget task
		$\mid$	$\texttt{status}\ e$	status of task
		$\mid$	sv	status values

where l ranges over *locations*, d ranges over *destinations* for the results of tasks, t ranges over *task handles*, and sv ranges over *statuses*. Both l and t are infinite sets of identifiers disjoint from variables, and as in standard practice, the identifiers are allowed to α-vary (i.e. changing the name does not change its identity).

Now, we extend the set of values to include locations, destinations, and statuses. For convenience, we separate values into another category, *resulting values*.

Resulting Values	rv	$::=$	$\ldots$	
		$\mid$	l	locations
		$\mid$	sv	status values
Values	v	$::=$	rv	
		$\mid$	d_t	result destination for task t
Destinations	d	$::=$	$\boxed{}$	waiting for result
		$\mid$	$\boxed{M, rv}$	filled with result
Statuses	sv	$::=$	`Disabled`	
		$\mid$	`Failed`	
		$\mid$	`Finished`	
		$\mid$	`Running`	

We choose to leave out the status `Waiting`, for it clutters the specification of the dynamic semantics without altering any behavior except the result of `status`. In this formulation, rather than changing status from between `Running` and `Waiting`, we leave the status at `Running`. We could imagine the status of tasks changing implicitly as appropriate.

A *memory* M is a finite mapping from locations to values. We write $M[l{=}v]$ for the addition or update of location l to v in memory M.

Memories	M	$::=$	$\cdot$
		$\mid$	$M, l{=}v$

We define the *grid* G as a finite mapping from task handles t to abstract tasks. An abstract task is defined as $(\dot{M}, \dot{e}) \rhd (sv, M, e)$ where the current state of the task is (sv, M, e) and the initial memory and code is $(\dot{M}, \dot{e})$. We write the tasks on the grid as follows:

$$G = \langle t_1{:}(\dot{M_1}, \dot{e_1}) \rhd (sv_1, M_1, e_1)\rangle, \ldots, \langle t_n{:}(\dot{M_n}, \dot{e_n}) \rhd (sv_n, M_n, e_n)\rangle$$

This yields the following stepping judgment:

$$G \mapsto G'$$

which says that grid state G steps to grid state G'. To not clutter the rules unnecessarily, we only write the tasks that change. Also, as the initial memory and code does not change during the execution of a task, we write it only when a task is created or restarted. For example, we write

$$G{::}\langle t{:}(sv, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle$$

as an abbreviation for the stepping judgment where task t makes a step and all other tasks in G stay the same. We can view the grid making a step by non-deterministically choosing some task t in G and then making a step in t.

Consider a simple substitution semantics for the effect-free core of ML augmented with G, M, sv, and t in a straightforward manner. We now present the additional rules for references and tasks. First, we give the rules for the reference-related constructs.

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{ref}(e))\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{ref}(e'))\rangle} \quad (1)$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{!}e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{!}e')\rangle} \quad (2)$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e_1)\rangle \mapsto G'{::}\langle t{:}(sv', M', e_1')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, e_1 \texttt{ := } e_2)\rangle \mapsto G'{::}\langle t{:}(sv', M', e_1' \texttt{ := } e_2)\rangle} \quad (3)$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e_2)\rangle \mapsto G'{::}\langle t{:}(sv', M', e_2')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, v_1 \texttt{ := } e_2)\rangle \mapsto G'{::}\langle t{:}(sv', M', v_1 \texttt{ := } e_2')\rangle} \quad (4)$$

$$\frac{l \notin \mathrm{dom}(M)}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{ref}(v))\rangle \mapsto G'{::}\langle t{:}(\texttt{Running}, M[l{=}v], l)\rangle} \quad (5)$$

$$\frac{l \in \mathrm{dom}(M)}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{!}l)\rangle \mapsto G'{::}\langle t{:}(\texttt{Running}, M, M(l))\rangle} \tag{6}$$

$$\frac{l \in \mathrm{dom}(M)}{G{::}\langle t{:}(\texttt{Running}, M, l \texttt{ := } v)\rangle \mapsto G'{::}\langle t{:}(\texttt{Running}, M[l{=}v], \langle\rangle)\rangle} \tag{7}$$

Now we consider the task-related constructs. We first present the compatibility rules.

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e_1)\rangle \mapsto G'{::}\langle t{:}(sv', M', e_1')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{inject }(e_1, e_2))\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{inject }(e_1', e_2))\rangle} \tag{8}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e_2)\rangle \mapsto G'{::}\langle t{:}(sv', M', e_2')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{inject }(v_1, e_2))\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{inject }(v_1, e_2'))\rangle} \tag{9}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{enable } e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{enable } e')\rangle} \tag{10}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{sync } e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{sync } e')\rangle} \tag{11}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{syncall } e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{syncall } e')\rangle} \tag{12}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{relax } e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{relax } e')\rangle} \tag{13}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{forget } e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{forget } e')\rangle} \tag{14}$$

$$\frac{G{::}\langle t{:}(\texttt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(sv', M', e')\rangle}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{status } e)\rangle \mapsto G'{::}\langle t{:}(sv', M', \texttt{status } e')\rangle} \tag{15}$$

To `inject` a task into the network, we set its initial status as appropriate, copy the memory, and apply the function to its argument. Note that once the task is injected, the contents of t_o's memory diverges from the contents of t's memory. The copying of memory is, in some sense, a kind of marshaling we may want to consider. Considering this memory copying operation would be intensive in implementation, it is not clear at this point whether or not we want to do this.

$$\frac{t_o \notin \mathrm{dom}(G) \qquad v = (v_1, v_2)}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{inject (true}, v\texttt{)})\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, \boxed{}_{t_o})\rangle, \langle t_o{:}(M, v_1\ v_2) \rhd (\texttt{Running}, M, v_1\ v_2)\rangle \end{array}} \tag{16}$$

$$\frac{t_o \notin \mathrm{dom}(G) \qquad v = (v_1, v_2)}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{inject (false}, v\texttt{)})\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, \boxed{}_{t_o})\rangle, \langle t_o{:}(M, v_1\ v_2) \rhd (\texttt{Disabled}, M, v_1\ v_2)\rangle \end{array}} \tag{17}$$

We only `enable` disabled tasks; otherwise, we do nothing.

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{enable}\ \boxed{}_{t_o})\rangle, \langle t_o{:}(\texttt{Disabled}, M_o, e_o)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, \langle\rangle)\rangle, \langle t_o{:}(\texttt{Running}, M_o, e_o)\rangle \end{array}} \tag{18}$$

$$\frac{sv_o \neq \texttt{Disabled}}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{enable}\ \boxed{}_{t_o})\rangle, \langle t_o{:}(sv_o, M_o, e_o)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, \langle\rangle)\rangle, \langle t_o{:}(sv_o, M_o, e_o)\rangle \end{array}} \tag{19}$$

$$\frac{}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{enable}\ \boxed{M_o, rv}_{t_o})\rangle \mapsto G'{::}\langle t{:}(\texttt{Running}, M, \langle\rangle)\rangle} \tag{20}$$

To receive the result from task t_o, we need to copy the memory from t_o (i.e. M_o) to the memory of task t (i.e. M). We write $[l'_1/l_1, \ldots, l'_n/l_n]v$ as the simultaneous substitution of l'_1 for l_1, l'_2 for l_2, …, and l'_n for l_n in v.

$$\frac{M_o = (l_1{=}v_1, \ldots, l_n{=}v_n) \quad l'_1, \ldots, l'_n \notin \mathrm{dom}(M) \quad \theta = [l'_1/l_1, \ldots, l'_n/l_n]}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{sync}\ \boxed{M_o, rv}_{t_o})\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M[l'_1{=}\theta v_1, \ldots, l'_n{=}\theta v_n], \theta rv)\rangle \end{array}} \tag{21}$$

$$\frac{}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{syncall nil})\rangle \mapsto G'{::}\langle t{:}(\texttt{Running}, M, \texttt{nil})\rangle} \quad (22)$$

$$\frac{M_o = (l_1{=}v_1, \ldots, l_n{=}v_n) \quad l'_1, \ldots, l'_n \notin \mathrm{dom}(M) \quad \theta = [l'_1/l_1, \ldots, l'_n/l_n]}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{syncall}\ \boxed{M_o, rv}_{t_o}\ \texttt{::}\ r)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M[l'_1{=}\theta v_1, \ldots, l'_n{=}\theta v_n], \theta rv\ \texttt{::}\ \texttt{syncall}\ r)\rangle \end{array}} \quad (23)$$

Note that there is no possible step from a state evaluating `relax nil`.

$$\frac{M_o = (l_1{=}v_1, \ldots, l_n{=}v_n) \quad l'_1, \ldots, l'_n \notin \mathrm{dom}(M) \quad \theta = [l'_1/l_1, \ldots, l'_n/l_n]}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{relax}\ \boxed{M_o, rv}_{t_o}\ \texttt{::}\ r)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M[l'_1{=}\theta v_1, \ldots, l'_n{=}\theta v_n], (\theta rv, r))\rangle \end{array}} \quad (24)$$

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{relax}\ \boxed{}_{t_o}\ \texttt{::}\ r)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, \texttt{let}\ (v, r') = \texttt{relax}\ r\ \texttt{in}\ (v, \boxed{}_{t_o}\ \texttt{::}\ r'))\rangle \end{array}} \quad (25)$$

Calling `forget` is simply an explicit indication of the failure of a task. When a task fails, the memory and expression is lost. If the task has already completed, then we simply forget the answer.

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{forget}\ \boxed{}_{t_o})\rangle, \langle t_o{:}(sv, M_o, e_o)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, \langle\rangle)\rangle, \langle t_o{:}(\texttt{Failed}, \cdot, \langle\rangle)\rangle \end{array}} \quad (26)$$

$$\frac{}{G{::}\langle t{:}(\texttt{Running}, M, \texttt{forget}\ \boxed{M_o, rv}_{t_o})\rangle \mapsto G'{::}\langle t{:}(\texttt{Running}, M, \langle\rangle)\rangle} \quad (27)$$

The `status` is given by the state of the task.

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{status}\ \boxed{}_{t_o})\rangle, \langle t_o{:}(sv, M_o, e_o)\rangle \\ \mapsto \\ G'{::}\langle t{:}(\texttt{Running}, M, sv)\rangle, \langle t_o{:}(sv, M_o, e_o)\rangle \end{array}} \quad (28)$$

$$\frac{}{\langle t{:}(\mathtt{Running}, M, \mathtt{status}\ \boxed{M_o, rv}_{t_o})\rangle \mapsto G'{::}\langle t{:}(\mathtt{Running}, M, \mathtt{Finished})\rangle} \quad (29)$$

Finally, when a task completes, the result is propagated and it disappears from G. For notational convenience, we define lift substitutions θ over memories and abstract tasks as follows:

$$\begin{aligned}
\theta(\cdot) &\overset{def}{=} \cdot \\
\theta(M, l{=}v) &\overset{def}{=} (\theta M), l{=}\theta v \\
\theta\langle t{:}(\dot{M}, \dot{e}) \rhd (sv, M, e)\rangle &\overset{def}{=} \langle t{:}(\theta\dot{M}, \theta\dot{e}) \rhd (sv, \theta M, \theta e)\rangle
\end{aligned}$$

$$\frac{\begin{array}{l} G = \langle t_1{:}\cdots\rangle, \ldots, \langle t_{i-1}{:}\cdots\rangle, \langle t_i{:}(sv, M_i, rv)\rangle, \langle t_{i+1}{:}\cdots\rangle, \ldots, \langle t_n{:}\cdots\rangle \\ \theta = [\ \boxed{M_i, rv}_{t_i} / \boxed{}_{t_i}\ ] \end{array}}{G \mapsto \theta\langle t_1{:}\cdots\rangle, \ldots, \theta\langle t_{i-1}{:}\cdots\rangle, \theta\langle t_{i+1}{:}\cdots\rangle, \ldots, \theta\langle t_n{:}\cdots\rangle} \quad (30)$$

With the rules given above, failure is non-existent (except for user-specified failure with `forget`). Also, there is no failure recovery. If we want to also model random failure on the network, we can add the following:

$$\frac{}{G{::}\langle t{:}(\mathtt{Running}, M, e)\rangle \mapsto G'{::}\langle t{:}(\mathtt{Failed}, \cdot, \langle\rangle)\rangle} \quad (31)$$

assuming there is some way to effectively detect failures. To specify failure recovery, we add the following rules that upon `sync`ing restart tasks when necessary.

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\mathtt{Running}, M, \mathtt{sync}\ \boxed{}_{t_o})\rangle, \langle t_o{:}(\dot{M}_o, \dot{e}_o) \rhd (\mathtt{Failed}, \cdot, \langle\rangle)\rangle \\ \mapsto \\ G{::}\langle t{:}(\mathtt{Running}, M, \mathtt{sync}\ \boxed{}_{t_o})\rangle, \langle t_o{:}(\dot{M}_o, \dot{e}_o) \rhd (\mathtt{Running}, \dot{M}_o, \dot{e}_o)\rangle \end{array}} \quad (32)$$

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\mathtt{Running}, M, \mathtt{syncall}\ \boxed{}_{t_o} :: r)\rangle, \langle t_o{:}(\dot{M}_o, \dot{e}_o) \rhd (\mathtt{Failed}, \cdot, \langle\rangle)\rangle \\ \mapsto \\ G{::}\langle t{:}(\mathtt{Running}, M, \mathtt{syncall}\ \boxed{}_{t_o} :: r)\rangle, \langle t_o{:}(\dot{M}_o, \dot{e}_o) \rhd (\mathtt{Running}, \dot{M}_o, \dot{e}_o)\rangle \end{array}} \quad (33)$$

$$\frac{}{\begin{array}{c} G{::}\langle t{:}(\texttt{Running}, M, \texttt{relax}\ \boxed{}_{t_o}\ {::}\ r)\rangle, \langle t_o{:}(\dot{M}_o, \dot{e}_o) \rhd (\texttt{Failed}, \cdot, \langle\rangle)\rangle \\ \mapsto \\ G{::}\langle t{:}(\texttt{Running}, M, \texttt{relax}\ \boxed{}_{t_o}\ {::}\ r)\rangle, \langle t_o{:}(\dot{M}_o, \dot{e}_o) \rhd (\texttt{Running}, \dot{M}_o, \dot{e}_o)\rangle \end{array}} \tag{34}$$

From this description, the issue related to garbage-collection of tasks as described in the conclusion becomes more clear. We see that tasks that fail may never get removed from the grid state G. Ideally, we would garbage collect a task t_f if it has failed and no destinations for t_f exist; however, as noted before, we do not know if there is a way to make this feasible.