Efficient Logic Variables for Distributed Computing

SEIF HARIDI
Swedish Institute of Computer Science (SICS)
PETER VAN ROY
Université Catholique de Louvain and SICS
PER BRAND
Swedish Institute of Computer Science
MICHAEL MEHL and RALF SCHEIDHAUER
German Research Center For Artificial Intelligence (DFKI)
and
GERT SMOLKA
Universität des Saarlandes and DFKI

We define a practical algorithm for distributed rational tree unification and prove its correctness in both the off-line and on-line cases. We derive the distributed algorithm from a centralized one, showing clearly the trade-offs between local and distributed execution. The algorithm is used to realize logic variables in the Mozart Programming System, which implements the Oz language (see `http://www.mozart-oz.org`). Oz appears to the programmer as a concurrent object-oriented language with dataflow synchronization. Logic variables implement the dataflow behavior. We show that logic variables can easily be added to the more restricted models of Java and ML, thus providing an alternative way to do concurrent programming in these languages. We present common distributed programming idioms in a network-transparent way using logic variables. We show that in common cases the algorithm maintains the same message latency as explicit message passing. In addition, it is able to handle uncommon cases that arise from the properties of latency tolerance and third-party independence. This is evidence that using logic variables in distributed computing is beneficial at both the system and language levels. At the system level, they improve latency tolerance and third-party independence. At the language level, they help make network-transparent distribution practical.

This research is funded in Sweden by the Swedish national board for industrial and technical development (NUTEK) and SICS. This research is partially funded in Belgium by the Walloon Region. The development of Mozart at DFKI is supported by the BMBF through Project PERDIO (FKZ ITW 9601).
Author's addresses: S. Haridi and P. Brand, Swedish Institute of Computer Science, S-164 28 Kista, Sweden; email: {seif; perbrand}@sics.se; P. Van Roy, Department of Computing Science and Engineering, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium; email: pvr@info.ucl.ac.be; M. Mehl, current address: T-Nova, Deutsche Telekom Innovationsgesellschaft mbH, Entwicklungszentrum Süd-West, BU TMN, Postfach 10 08 33, D-66008 Saarbrücken, Germany; email: Michael.Mehl@telekom.de; R. Scheidhauer, current address: IDS Scheer AG, Postfach 10 15 34, D-66015 Saarbrücken, Germany; email: R.Scheidhauer@IDS-Scheer.de; G. Smolka, current address: Universität des Saarlandes, D-66123 Saarbrücken, Germany; email: smolka@ps.uni-sb.de.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; *constraint and logic languages*; *data-flow languages*; *multiparadigm languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; *constraints*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*online computation*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Distributed algorithms, Oz, Mozart

1. INTRODUCTION

Logic variables were first studied in the context of logic programming [Robinson 1965; Warren 1977]. They remain an essential part of logic programming and constraint programming systems [Van Roy 1994; Jaffar and Maher 1994]. In the context of the Distributed Oz project, we have come to realize their usefulness to distribution [Haridi et al. 1997; Smolka et al. 1995]. Logic variables express dependencies between computations without imposing an execution order. This property can be exploited in distributed computing:

—Two basic concerns in distributed computing are latency tolerance and third-party independence. We say a program is *third-party independent* if its execution is unaffected by sites that are not currently involved in the execution. We show that using logic variables instead of explicit message passing can reduce the effect of both concerns with little programming effort.

—With logic variables we can express common distributed programming idioms in a network-transparent manner that results in optimal or near-optimal message latency. That is, the same idiom that works well in a centralized setting also works well in a distributed setting.

The main contribution of this article is a practical algorithm for distributed rational tree unification that realizes these benefits. The algorithm is used to implement logic variables in the Mozart system. We formally define the algorithm and prove that it satisfies safety and liveness properties in both the off-line and on-line cases.

From the programmer's point of view, the use of logic variables adds a dataflow component to program execution. In a first approximation, this component can be completely ignored. That is, it is invisible to the programmer whether or not a thread temporarily blocks while waiting for a variable's value to arrive. Programs can be developed using well-known techniques of concurrent object-oriented programming [Lea 1997]. In a second approximation, the dataflow component greatly simplifies many concurrent programming tasks [Haridi and Franzén 1999; Bal et al. 1989].

This article consists of two parts that may be read independently of each other. The first part, Section 2, motivates and discusses in depth the use of logic variables in concurrent and distributed programming. Section 2.1 introduces a general execution model, its distributed extension, and the concept of the logic variable.

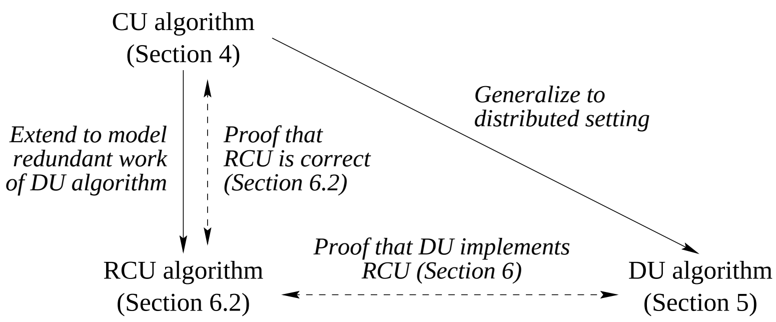

Fig. 1. Defining the algorithm and proving it correct.

Section 2.2 gives the key ideas of the distributed unification algorithm. Section 2.3 shows how to express basic concepts in concurrent programming using logic variables. Section 2.4 expresses common distributed programming idioms in a network-transparent manner with logic variables. We show that the algorithm provides good network behavior for these examples. Finally, Section 2.5 shows how to add logic variables in an orthogonal way to Java and ML, taken as representative examples of object-oriented and functional languages.

The second part, Section 3 and following, defines the distributed unification algorithm, proves its total correctness (see Figure 1), and discusses its implementation. Section 3 defines the formal representation of logic variables and data structures. This section also defines configurations and executions and introduces the reduction rule notation used to define algorithms. Section 4 defines the CU algorithm, which implements off-line centralized unification, and summarizes well-known results about its correctness. By *off-line* we mean that the set of equations is finite and initially known. Section 5 defines the DU algorithm, which implements off-line distributed unification. Section 6 defines the RCU algorithm, which modifies the centralized algorithm to reflect the redundant work done by the DU algorithm. The section then proves that the DU algorithm is a correct implementation of the CU and RCU algorithms. Section 7 defines on-line versions of the CU and DU algorithms. By *on-line* we mean that new equations can nondeterministically be introduced at any moment. We define the *finite-size* property and prove, that, given weak fairness, every introduced equation that satisfies this property is eventually entailed by the store for both algorithms. Section 8 defines the algorithm used in the Mozart system, which refines, optimizes, and extends the on-line DU algorithm in many ways.

2. LOGIC VARIABLES IN CONCURRENT AND DISTRIBUTED SETTINGS

This section motivates our unification algorithm by showing its usefulness to distributed programming. First Section 2.1 introduces our execution model and its notation. Then Section 2.2 gives the key ideas of the algorithm. This is followed by Section 2.3 which gives programming examples showing the usefulness of logic variables for basic tasks in concurrent programming. Section 2.4 continues with

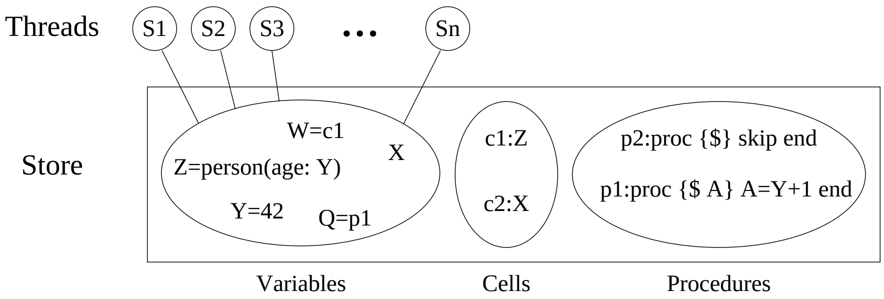

Fig. 2. The Oz execution model.

tasks in distributed programming. We explain in detail the network behavior of our algorithm for these tasks. Finally, Section 2.5 shows how to add logic variables to other languages including the Java and ML families.

2.1 Basic Concepts and Notation

As a framework for logic variables, we introduce a general execution model that can accommodate most programming languages. Underlying it is a formal model called *concurrent constraints* [Smolka 1995; Saraswat 1993] that contains logic variables as a basic ingredient. Some uses of logic variables, e.g., synchronization and communication, already appear in this model. The general execution model, called the Oz execution model, extends the concurrent constraint model with explicit state and higher-orderness. Other uses of logic variables, e.g., locks, become possible when explicit state is added.

This section gives the essential characteristics of the Oz execution model and how it is distributed. Later on, we show how to add logic variables to the more restricted models of Java and ML. The advantage of using a general formal model is that it allows us to define precisely what the examples do. It is straightforward to compile Java or ML to Oz; the converse is not as easy.

The Oz language has advanced support for logic programming and constraint programming [Smolka 1996; Schulte 1997]. This support shows up in both the Oz execution model and kernel language. We do not present this support here since it is outside the scope of the article.

2.1.1 *The Oz Execution Model.* The Oz execution model consists of a store and a set of dataflow threads that reference logic variables in the store (see Figure 2). Threads contain statement sequences S_i and communicate through shared references. A thread is a *dataflow* thread if it only executes its next statement when all the values the statement needs are available. Data availability is implemented using logic variables. If the statement needs a value that is not yet available, then the thread automatically blocks until the value is available. We add the fairness condition that if all values are available then the thread will eventually execute its next statement.

The shared store is not physical memory; rather it is an abstract store that only allows legal operations for the entities involved, i.e., there is no direct way to

S ::=	S S	Sequence
|	X=f(l_1:Y_1 ... l_n:Y_n) |	Value
	X=<number> | X=<atom> | {NewName X}	
|	**local** X_1 ... X_n **in** S **end** | X=Y	Variable
|	**proc** {X Y_1 ... Y_n} S **end** | {X Y_1 ... Y_n}	Procedure
|	{NewCell Y X} | {Exchange X Y Z} | {Access X Y}	State
|	**if** X **then** S **else** S **end**	Conditional
|	**thread** S **end** | {GetThreadId X}	Thread
|	**try** S **catch** X **then** S **end** | **raise** X **end**	Exception

Fig. 3. The Oz kernel language.

inspect their internal representations. The store consists of three compartments, namely logic variables (with optional bindings), cells (named mutable pointers, i.e., explicit state), and procedures (named lexically scoped closures, i.e., higher-orderness). Variables can reference the names of procedures and cells. Cells point to variables. The external references of threads and procedures are variables. When a variable is bound, it disappears, i.e., all threads that reference it will automatically reference the binding instead. Variables can be bound to any entity, including other variables. The variable and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Because of monotonicity, a thread that is not blocked is guaranteed to stay not blocked until it executes its next statement.

2.1.2 *The Oz Language.* All Oz execution can be defined in terms of a kernel language whose semantics are outlined in Haridi and Franzén [1999] and Van Roy et al. [1997]. The current Oz language is called Oz 2 to distinguish it from an earlier language, Oz 1, whose kernel language is called the Oz Programming Model (OPM) [Smolka 1995]. Oz 1 was designed for fine-grained concurrency and implicit exploitation of parallelism. Oz 2 abandons this model in favor of explicit control over concurrency by means of a thread creation construct. We do not discuss Oz 1 further in this article.

Figure 3 defines the abstract syntax of a statement S in the Oz kernel language. Statement sequences are reduced sequentially inside a thread. All variables are logic variables, declared in an explicit scope defined by the **local** statement. Values (records, numbers, names, etc.) are introduced explicitly and can be equated to variables. A *name* is a unique unforgeable constant that has no external representation. A new name is created by calling `NewName`. Procedures are defined at run-time with the **proc** statement and referred to by a variable. Procedure applications block until the first argument references a procedure name. State is created explicitly by `NewCell`, which creates a *cell*, a mutable pointer into the variable store. Cells are updated by `Exchange` and read by `Access`. The **if** statement defines a conditional that blocks until its condition is true or false in the variable store. Threads are created explicitly with the **thread** statement. Each thread has a unique identifier that is used for thread-related operations. Exception handling is dynamically scoped. The **try** statement defines a scope, and the **raise** statement raises an exception that is caught by the innermost enclosing scope.

The full Oz language is defined by transforming all its statements into this kernel language. Oz supports idioms such as objects, classes, reentrant locks, and a variety of channels called "ports" [Smolka 1995; Van Roy et al. 1997]. The system implements them efficiently while respecting their definitions. We give a brief summary of each idiom's definition. For clarity, we have made small conceptual simplifications. Full definitions are given in Haridi and Franzén [1999].

—**Object**: An object is essentially a one-argument procedure `{Obj M}` that references a cell, which is hidden by lexical scoping. The cell holds the object's state. The argument `M` indexes into the method table. A method is a procedure that is given the message and the object state and calculates the new state.

—**Class**: A class is essentially a record that contains the method table and attribute names. A class is defined through multiple inheritance, and any conflicts are resolved at definition time when building its method table.

—**Reentrant lock**: A reentrant lock is essentially a one-argument procedure `{Lck P}` used for explicit mutual exclusion, e.g., of method bodies in objects used concurrently. `P` is a zero-argument procedure defining a critical section. Reentrant means that the same thread is allowed to reenter the lock. Calls to the lock may therefore be nested. The lock is released automatically if the thread in the body terminates or raises an exception that escapes the lock body.

—**Port**: A port is an asynchronous channel that supports many-to-one communication. A port `P` encapsulates a stream `S`. A *stream* is a list with an unbound tail. The operation `{Send P M}` adds `M` to the end of `S`. Successive sends from the same thread appear in the order they were sent.

2.1.3 *The Distribution Model.* The Mozart system implements Distributed Oz, which is a conservative extension to the centralized Oz language [DFKI Oz 1998] that completely separates functionality from distribution structure. That is, Oz language semantics are unchanged,[1] while adding predictable and programmable control over network communication patterns. Porting existing Oz programs to Distributed Oz requires essentially no effort.

Allowing a successful separation of the functionality from the distribution structure puts severe restrictions on a language. It would be almost impossible in C++ because of its complex, informal semantics and because the programmer has full access to all underlying representations [Stroustrup 1997]. It is possible in Oz because of the following properties:

—Oz has a simple formal foundation that does not sacrifice expressiveness or efficient implementation. Oz appears to the programmer as a concurrent object-oriented language whose basic functionality is comparable to modern languages such as Java. The current emulator-based implementation is competitive with Java emulators [Henz 1997b; 1997a]. Standard techniques for concurrent object-oriented design apply to Oz [Lea 1997]. Furthermore, Oz introduces powerful new techniques that are not supported by Java [Haridi and Franzén 1999]. Some of these techniques are presented in this article.

—Oz is both a state-aware and dataflow language. That is, language entities can be classified naturally into stateless, single assignment, and stateful. This helps

[1] Only ports are changed slightly to better model asynchronous FIFO communication between sites [Van Roy et al. 1997].

give the programmer control over network communication patterns in a natural manner. Stateless data include procedures and values, which can safely be copied to many sites [Alouini and Van Roy 1999]. Stateful data include objects, which at any instant must reside on just one site [Van Roy et al. 1997]. Single-assignment data include logic variables, whose dataflow synchronization allows us to decouple calculating a value from sending it across the network.

—Oz is a fully dynamic and compositional language. That is, Oz is dynamically typed, and all entities are first-class. By *dynamically typed* we mean that its type structure is checked at run-time. This makes it easy to implement fully open distribution, in which independent computations can connect and disconnect at will. When connected they can communicate as if they were in the same centralized process. For example, it is possible to define a class `C` in one computation, pass `C` to an *independent* computation that has never before heard of `C`, let the independent computation define a class `D` inheriting from `C`, and pass `D` back to the original computation [Van Roy et al. 1999; Haridi et al. 1998].

—Oz provides language security. That is, all references to language entities are created and passed explicitly. An application cannot forge references nor access references that have not been explicitly given to it. The underlying representation of language entities is inaccessible to the programmer. This is a consequence of the abstract store and a kernel language with lexical scoping and first-class procedures. These are essential properties to implement a capability-based security policy, which is important in open distribution.

The Distributed Oz execution model extends the Oz execution model by giving a distributed semantics to each language entity. The distributed semantics defines the network behavior when language entities are shared between sites. The semantics is chosen carefully to give predictable control over network communication patterns. The centralized semantics is unchanged: we say the model is *network-transparent* [Cardelli 1995]. In the current system, language entities are put in four categories. Each category is implemented by a family of distributed protocols:

—**Stateless**: records, numbers, procedures, and classes. Since they do not change, these entities can be copied at will.[2] There is a trade-off between when to copy, how many times to copy to a site, and access time. This gives a family of protocols to define their distributed behaviors [Alouini and Van Roy 1999].

—**Single assignment**: logic variables. Assignment is done by a distributed unification algorithm, which is the subject of this article. To be precise, logic variables provide *consistent multiple assignment*, i.e., there can be multiple assignments as long as they are unifiable. We keep the phrase "single assignment" to avoid multiplying terminology.

—**Stateful**: cells, objects, reentrant locks, ports, and threads. For efficiency reasons, these entities' state pointers are localized to a site. If the state pointer's site can change, we say that the entity is mobile. Currently there are two mobility behaviors: a mobile state protocol (cells, objects, locks, ports) and a stationary

[2]This is true only for the entity, not for its external references. An external reference has its own protocol that corresponds to its category.

access protocol (threads). The mobile state protocol ensures coherent state updates with controlled mobility of the state pointer [Van Roy et al. 1997]. The stationary access protocol is used for entities that cannot move.

—**Resource**: entities external to the shared store. References to resources can be passed around the network at will, but the resource can only be executed on its home site [Van Roy et al. 1999]. This includes computational and memory resources, which can be made visible in the language, e.g., by means of *virtual sites* [Haridi et al. 1998].

The single-assignment category can be seen as an optimization of the stateful category in which a variable is bound to only one value, instead of repeatedly to different values. That is, the distributed unification algorithm is more efficient than the mobile state protocol. However, it turns out that logic variables have many more uses than simply as an optimization of stateful entities. These uses are explained below.

2.1.4 *Logic Variables.* A logic variable conceptually has a fixed value from the moment of its creation. The value is unknown at first, and it remains unknown until the variable is bound. At all times, the variable can be used as if it were the value. If the value is needed, then the thread requiring the value will block until the variable is bound. If the value is not needed then execution continues.

A logic variable can be passed among sites arbitrarily. At all times, it "remembers its origins," i.e., when the value becomes known then the variable will receive it. The communication needed to bind the variable is part of the variable and not part of the program manipulating the variable. This means that the variable can be passed around at will, and the value will always arrive at the variable. This is one key reason why logic variables are useful in distributed computing.

Logic variables can replace standard (assignable) variables in all cases where they are assigned only one value, i.e., where they are used as placeholders for values. The algorithm used to bind logic variables must ensure that the result is independent of binding order. In a centralized system, the algorithm is called *unification* and is usually implemented as an extension of a union-find algorithm. Union-find handles only the binding of variables with variables [Mehlhorn and Tsakalidis 1990]. Unification generalizes this to handle nonvariable terms as well. In a good implementation, binding a new variable to a nonvariable (the common case) compiles to a single register move or store operation [Van Roy 1994].

A logic variable may be bound to another logic variable. A legitimate question is whether variable-variable binding is useful in a practical system. As we shall see, one reason that variable-variable binding is important is that it allows us to maintain maximum latency tolerance and third-party independence when communicating among more than two sites, independent of fluctuating message delays. A second reason is that it has a very simple logical semantics.

It is possible to disallow variable-variable binding to obtain a slightly simpler implementation. The simpler implementation blocks any attempt to do variable-variable binding until at least one of the variables is bound to a value. The price of the simpler implementation is that third-party dependencies are not removed in all cases. Futures [Halstead 1985] and I-structures [Arvind and Thomas 1980; Veen 1986; Iannucci 1990] resemble this weaker version of logic variables (see Section 9.2.1). There remains a crucial difference with logic variables, namely that

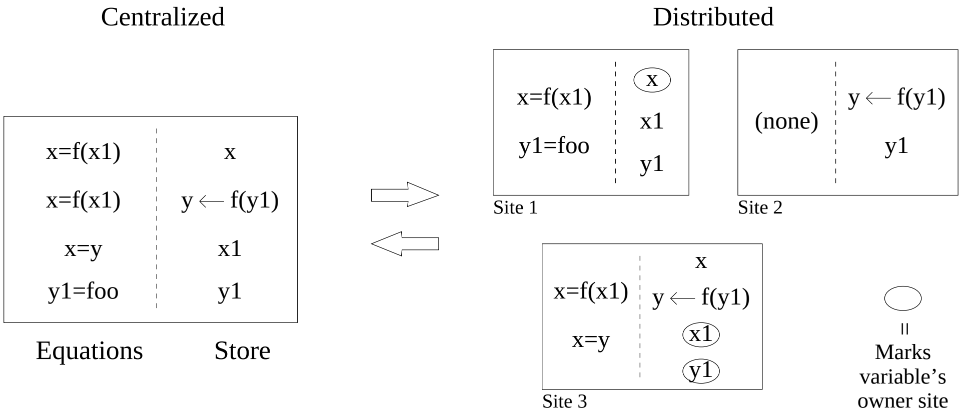

Fig. 4. Initial configuration of example.

futures and I-structures can be assigned only once, whereas logic variables can be assigned more than once, as long as the assignments are consistent with each other.

The efficiency difference between full and weak logic variables is small. The distributed binding algorithm is almost identical for the full and weak versions. Furthermore, the full version has a simple logical semantics. For these three reasons we have implemented the full version in the Distributed Oz implementation.

2.2 Distributed Unification

For logic variables to be practical in a distributed setting they must be efficiently implementable. This section gives the key ideas of the distributed algorithm that realizes this goal. We explain the algorithm in just enough detail so that its network behavior becomes clear. This will allow us to infer the algorithm's behavior in the programming examples that follow. A formal definition of the algorithm and proofs of its correct behavior are given starting from Section 3.

The two basic operations on logic variables are binding and waiting until bound. Waiting until bound is easy: the variable's implementation has a list of threads that need its value. These threads are blocked. When the value arrives, the threads are awoken. Binding is harder: it requires cooperation between sites. If a variable exists on several sites, then it must be bound to the same value on all sites, despite concurrent binding attempts. Unification implements the binding operation. At any instant there can be any number of bindings in various stages of completion. Both the centralized and distributed algorithms cause each binding request to be eventually incorporated into the store, if it is consistent with the store.

The basic distributed operation is binding a variable to a value. This is implemented by making one site the "owner" of the variable. In the current system, the site that declares the variable is its owner. A binding request is sent to the owner, and the owner forwards the binding to each site that knows the variable. In terms of network behavior, one message is sent to the owner, and one message is sent by the owner to each site that knows the variable. The owner's sends can be done by a reliable multicast, if the network supports it efficiently. The owner accepts the first binding request and ignores all subsequent binding requests. An ignored request

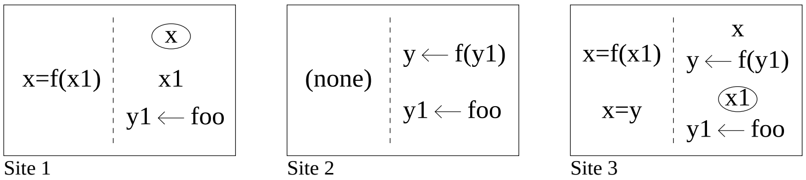

Fig. 5. Configuration after executing the equation Y1=foo.

will be retried by its initiating site after it receives the binding. As we will see in the programming examples, in the majority of cases a variable is declared either on a site that will need its value or on the site that will bind the variable. In both of these cases, the network behavior of the algorithm is very good.

A logic variable X can be bound to a data structure or to another variable. The algorithm is the same in both cases. By default the binding is *eager*, i.e., the new value is immediately sent to all sites that know about X. This means that a bound variable is guaranteed to eventually disappear from the system. The same binding eventually appears on each site that has the variable. For example, executing the equation X=f(X1) causes the binding X ← f(X1) to appear in the store on all sites containing X. Furthermore, X1 is added to the known variables of all of these sites that did not know X1.

2.2.1 *An Example.* We illustrate the algorithm with an example. Figure 4 shows a centralized configuration (on the left) and one way to distribute it (on the right). Each configuration has a set of equations and a store. For the algorithm, an *equation* is simply a request to perform a binding. In the formal discussion (Section 3 and following), we need more kinds of requests than just equations. We call all the requests *actions*. The same equation may exist more than once. The store contains the variables and their bindings, if the latter exist.

In the distributed case, each site has a set of equations and a store. The centralized equations are distributed among the sites. Each variable is visible on a subset of the sites. If there is only one site, then the distributed algorithm is identical to the centralized algorithm. Each variable occurrence on a site is called a "proxy." One of the sites is the variable's owner. In Figure 4, site 1 is the owner of X, and site 3 is the owner of both X1 and Y1. If the variable is bound, then the binding will eventually arrive on each site that sees the variable. Variable Y is bound to f(Y1) on sites 2 and 3.

Site 1 requests the binding Y1=foo. This sends a message to site 3, the owner of Y1. The owner sends a message to all proxies of Y1. That is, the owner sends three messages, to sites 1, 2, and 3. When a message arrives on a site, then the binding Y1 ← foo appears on that site (see Figure 5). Since the owner is on site 3, its message to site 3 does not need any network operations.

2.2.2 *Lazy and Eager Variables.* Logic variables can have different distributed behaviors, as long as network transparency is satisfied in each case. As explained above, by default a logic variable is *eager* on all sites, i.e., its binding is sent immediately to all sites that reference the variable. This gives maximal latency tolerance and third-party independence. However, this may cause the binding to be sent to sites that do not need it. We say that a logic variable is *lazy* on a site if its value is only sent to that site when the site needs it, e.g., when a thread is waiting for the variable. Binding a lazy variable typically needs fewer messages, since not all sites that know the variable need its value. Both eager and lazy variables are implemented by the on-line DU algorithm of Section 5. They differ only in the scheduling of one reduction rule. The Mozart implementation currently only provides eager variables; with a minor change it can provide both. A programmer annotation can then decide whether a variable is eager or lazy. The implementation issues of laziness are further explored in Section 8.5.6.

2.3 Examples of Concurrent Programming

It turns out that logic variables suffice to express most concurrent programming idioms in an intuitive and concise manner. Additional concepts such as semaphores, critical sections, or monitors are not needed. Bal et al. [1989] conclude that logic variables are "spectacularly expressive" in concurrent programming even without explicit state. We give examples of four important idioms, namely synchronization, communication, mutual exclusion, and first-class channels. Many other idioms can be found in the concurrent logic programming literature [Shapiro 1989; Haridi and Franzén 1999].

2.3.1 *Synchronization and Communication.* The following fragment creates two threads and synchronizes their execution:

```
local X in
   thread {Print a} X=unit end
   thread {Wait X} {Print b} end
end
```

The statement `{Wait X}` blocks until `X`'s value is known. Therefore "a" is always printed before "b." The value of `X` is communicated from the first to the second thread. `{Wait X}` is not a new notion; it can be defined as **if** `X=1` **then skip else skip end**.

2.3.2 *Mutual Exclusion.* A critical section can be defined by means of logic variables and one cell. The cell is used to manage access to a token, which is passed from one thread to the next. Assume that a cell `C` exists with initial content **unit**, e.g., defined by `{NewCell` **unit** `C}`. Then the following fragment defines a critical section:

```
local X Y in
   {Exchange C X Y} {Wait X} % Enter
   ...                       % Body
   Y=unit                    % Exit
end
```

We show that only one thread at a time can be executing the body. A thread that tries to enter is given `C`'s previous state X_n and current state Y_n. The thread then waits on X_n. When the previous thread leaves, it binds Y_{n-1}=**unit**. Since $Y_{n-1} = X_n$, this allows the next thread to enter. This works even if many threads try to enter concurrently, since the exchanges are serialized. Section 2.4.6 uses this idea to define a procedure `NewSimpleLock` that can create any number of locks.

2.3.3 *First-Class Channels.* A simple kind of FIFO channel is the *stream*, a list with an unbound tail. Reading from the stream is receiving from the channel. Appending to the stream is sending through the channel. Each element of the stream can be a record containing both the query and an answer channel. For example, here is a fragment that handles queries appearing on the stream `X0`:

```
case X0 of query(Q1 A1)|X1 then % Wait for query Q1 and channel A1
   A1={CalcAnswerStream Q1}     % Calculate answer stream on A1
   case X1 of query(Q2 A2)|X2 then % Wait for Q1 and A2
      A2={CalcAnswerStream Q2}     % Calculate answer stream on A2
      ...
   end
end
```

We assume that `Q1` is a database query that gives multiple answers, which appear incrementally on `A1`. The **case** statement is a useful idiom: it waits until `X0` is sufficiently bound to read the pattern `query(Q1 A1)|X1`. The pattern variables `Q1`, `A1`, `X1` are declared implicitly. Typically, the above fragment would be written as part of a loop:

```
local P in
   proc{P M}
      case M of query(Q A) then A={CalcAnswerStream Q} end
   end
   {ForAll X0 P}
end
```

`ForAll` takes a list `X0` and a one-argument procedure, and applies the procedure to all the list's elements. The above example can be written more compactly as a nested statement with exactly the same meaning:

```
{ForAll X0
   proc{$ M}
      case M of query(Q A) then A={CalcAnswerStream Q} end
   end}
```

The "`$`" is used as a nesting marker; it implicitly declares a logic variable. This is a syntactic short-cut that avoids explicitly declaring `P`. Using `ForAll` is efficient; there are no memory leaks, and the stream is not consumed faster than it is produced. It may be produced faster than it is consumed, however. Usually, a stream is associated to an Oz *port*, and one writes to the port (see Section 2.1.2).

2.4 Examples of Distributed Programming

The purpose of this section is to show the usefulness of logic variables when extending concurrent object-oriented programming to a distributed setting. Sections 2.4.1–2.4.7 present a series of common programming idioms in distributed programming. We show how to express them in a concurrent object-oriented language that contains logic variables. The resulting solutions have two properties:

—The solutions perform correctly, independently of how they are partitioned among sites. That is, the programming idioms underlying the communication patterns can be expressed in a network-transparent manner.

—No matter how the solutions are partitioned among sites, the resulting message traffic is optimal or nearly optimal (in the common cases) or at least reasonable (in the uncommon cases). That is, given logic variables, the same programming idioms perform well in both centralized and distributed settings.

This shows, that, at least in the cases presented here, using logic variables allows us to keep useful programming idioms of centralized object-oriented programming, while allowing the implementation to extend efficiently to a distributed setting. This is evidence that controlling execution through data availability, which is what logic variables provide, is a natural way to keep good performance while mapping a program to arbitrary distribution structures.

The examples show a variety of distributed programming techniques using logic variables. Some of them, e.g., barrier synchronization and distributed locking, will normally be provided as primitives by the system. Others, e.g., stream communication, will normally be programmed by the user. We do not distinguish between the two cases, since our goal is to show the expressiveness of logic variables.

2.4.1 *Latency Tolerance and Third-Party Independence.* From the viewpoint of execution order of basic language operations, a distributed execution cannot be distinguished from a concurrent execution. Distinguishing them requires looking at the effects of partitioning an execution over several sites. This affects system properties such as network properties (e.g., delays and limited bandwidth) and site resources (e.g., disks and main memory). At the language level, the latter shows up as the restriction of some operations to be local or remote only (such as local memory operations and remote message sends).

Logic variables decouple the declaration of a variable from its binding. Once a variable is declared, it can be passed to other sites, even before it is bound. When it is bound, the binding will be transferred automatically and efficiently to the sites needing it. This decoupling allows programs to provide a degree of *latency tolerance*, i.e., their execution is less affected by changes in network latency. For example, in the following code fragment

```
local Ans in
   thread
      {DataBase query("How far is up?" Ans)}
   end
   thread
      {RemoteClient inform(Ans)}
   end
end
```

the database query and the client transfer are initiated concurrently. Assume that the database and the client are on different remote sites. The initiator site owns `Ans`. As soon as `Ans` is bound, the binding will be sent from the database site to the initiator site, which forwards it to the client site. This is all done independently of the initiator.

A logic variable can be bound to another logic variable. This allows programs to improve third-party independence. For example, assume variable `X` exists on sites 1 and 2, and variable `Y` exists on sites 2 and 3. Assume that `X` and `Y` are bound together on site 2. Then binding `X` to 99 on site 1 should be visible on site

```
proc {Generate N Max L} % Return list of integers from N to Max-1
   if N < Max then L1 in
      L=N|L1
      {Generate N+1 Max L1}
   else L=nil end
end

fun {Sum L A}           % Return (A + sum of elements of list L)
   case L
   of nil then A
   [] X|Ls then {Sum Ls A+X}
   end
end

local CS L S in         % Generate a list and sum its elements
  CS={NewComputeServer ´sinuhe.sics.se´} % Remote compute server
  thread L = {Generate 0 150000} end    % Producer thread (local)
  {CS proc {$} S={Sum L 0} end}         % Consumer thread (remote)
  {Print S}                             % Print result (local)
end
```

Fig. 6. Stream communication.

3 independent of what happens to site 2.

2.4.2 *Stream Communication.* This second example has a producer `Generate` that creates a data stream and a consumer `Sum` that reads this stream (see Figure 6). We first examine the program in a centralized setting. Then we explain what happens when the producer and consumer run on different sites. As we saw before, a stream is a list whose tail is a logic variable. The producer thread repeatedly binds the tail to a pair of an element and a new tail. The consumer thread can start reading the stream while the producer is still creating it. In the program of Figure 6, the producer generates a list of integers from 0 to 149999 and the consumer sums them, giving 11249925000.

This example will run in a distributed way if the producer thread and consumer thread are started on different sites. This is what the main program of Figure 6 does; it executes the producer locally and the consumer remotely. In the producer, binding `L` to `N|L1` adds one element to the stream. In the distributed execution this will send exactly one message to the consumer.

The consumer `Sum` is run remotely by packaging the call `S={Sum L 0}` into a zero-argument procedure that is passed to the compute server `CS`. The procedure's compiled code is sent across the network. The logic variable `S` is shared between the local and remote sites, and therefore transparently becomes a distributed variable. When `Sum` finishes its calculation then the result is bound to `S`. This sends the result back to the local site.

A compute server is a one-argument procedure that takes work packaged as a zero-argument procedure, and executes it in its own thread on the remote machine. The compute server is created by calling the function `NewComputeServer` with a hostname. This creates a process on the remote machine and returns a compute

```
proc {Generate N L} % Return list L of integers starting with N
   case L of X|Ls then % Wait until the next element is asked for
      X=N
      {Generate N+1 Ls}
   else skip end
end

fun {Sum N L A}      % Return (A + sum of first N elements of L)
   if N>0 then X L1 in
      L=X|L1           % Ask for the next element
      {Sum N-1 L1 A+X}
   else A end
end

local CS L S in
  CS={NewComputeServer ´sinuhe.sics.se´} % Remote compute server
  {CS proc {$} {Generate 0 L} end}      % Producer thread (remote)
  thread S={Sum 150000 L 0} end         % Consumer thread (local)
  {Print S}                             % Print result (local)
end
```

Fig. 7. Stream communication with flow control.

server.

Because of network transparency, all possible code fragments can be used as work to be transferred to the compute server. However, if the work uses resources, then it cannot be packaged in a procedure. This is because resources are site-specific (see Section 2.1.3). Instead, Oz allows such work to be packaged in a *functor*, which describes the resources that the work needs. Just like procedures, functors are stateless and can be passed across the network transparently. Functors are beyond the scope of this article; for more information see [Duchier et al. 1998; Van Roy et al. 1999].

2.4.3 *Stream Communication with Flow Control.* This third example extends the second example by adding end-to-end flow control (see Figure 7). A stream element is only generated when the consumer asks for it. The second example has no flow control, i.e., the producer will create elements eagerly independent of what the consumer needs. Unless the list's maximum size is small, flow control is needed to avoid problems with memory utilization. This is true in both centralized and distributed settings.

The consumer asks for an element by binding the stream's tail to a pair of a logic variable and a new tail. The producer waits until this pair exists and then binds the logic variable to the next element. The consumer can terminate the producer by binding `L` to `nil`, i.e., by replacing the "**`else`** `A` **`end`**" in its definition by "**`else`** `L=nil A` **`end`**." The producer terminates when it detects the end of `L`.

In this example the producer and consumer will execute in lock step. The main program of Figure 7 executes the producer locally and the consumer remotely. Therefore one round-trip message delay is needed for each element of the stream. To relax this tight synchronization, an n-element buffer can be programmed.

```
local CS1 CS2 CS3 L S1 S2 S3 in
   CS1={NewComputeServer ´sinuhe.sics.se´}
   CS2={NewComputeServer ´norge.info.ucl.ac.be´}
   CS3={NewComputeServer ´tinman.ps.uni-sb.de´}
   thread {Generate 0 L} end              % Producer (local)
   {CS1 proc {$} S1={Sum L 150000 0} end} % Consumer 1 (on Site 1)
   {CS2 proc {$} S2={Sum L 150000 0} end} % Consumer 2 (on Site 2)
   {CS3 proc {$} S3={Sum L 150000 0} end} % Consumer 3 (on Site 3)
end
```

Fig. 8. Stream communication with multiple readers.

2.4.4 *Stream Communication with Multiple Readers.* Now let the stream be read by multiple consumers. Figure 8 shows how to do it with consumers on three sites. We assume three compute servers referenced by `CS1`, `CS2`, and `CS3`. Both previous examples of stream communication (with and without flow control) will work with multiple consumers. This is an excellent illustration of the difference between logic variables and I-structures. It is allowed for multiple readers to bind the list's tail, since they bind it in a consistent way. This would not work with ordinary single assignment, e.g., as provided by I-structures.

The example without flow control is straightforward: one message is sent to each consumer per element. The example with flow control is more interesting; it is shown in Figure 8. In this case, each consumer sends a message to request the next element when it is needed. The network behavior is as follows. To make things interesting, we assume a fast, a medium, and a slow consumer. The fast consumer sends a message to the producer, which is the owner of the first stream variable. The message contains two variables: one for the element and one for the next stream variable. Both of these variables are owned by the fast consumer. It follows, that, from this point on, the fast consumer will be the owner of all stream variables. Therefore all further stream elements will be sent by the producer to the fast consumer, who will multicast them to the other consumers. After the first message, the medium consumer will send requests to the fast consumer, since it is the owner. These requests will be ignored, since the fast consumer will already have bound the stream variable. The slow consumer will send no requests at all; it receives the elements before asking for them.

2.4.5 *Barrier Synchronization.* We would like to create a set of concurrent tasks and be informed as soon as all tasks have finished. This should be efficient no matter how the tasks are partitioned over a set of sites. Figure 9 gives a simple solution that works well in both centralized and distributed settings. To explain how it works, we need first of all to understand how to synchronize on the termination of a single thread. This is done as follows, where statement `S` represents a task:

```
local X in thread S X=unit end ... {Wait X} end
```

The main thread creates a new thread whose body is `S X=unit`. The new thread will bind `X` after `S` is finished, and the main thread detects this with a `{Wait X}`. A statement `S` *finishes* when it reduces to **`skip`** in its thread. Other threads may be created during the execution of `S`; these are independent of `S`. If the task is executed

```
proc {BarrierSync Ps}
   proc {Conc Ps L}
      case Ps of P|Pr then X Ls in
         L=X|Ls
         thread {P} X=unit end
         {Conc Pr Ls}
      else
         L=nil
      end
   end
   L
in
   {Conc Ps L}
   {ForAll L proc {$ X} {Wait X} end}
end

{BarrierSync [proc {$} E1 end   % Task 1
              proc {$} E2 end   % Task 2
              proc {$} E3 end]} % Task 3
```

Fig. 9. Barrier synchronization.

remotely, then binding `X` sends a single message to the main site, which owns `X`. This informs the thread of the task's completion. The message sent back to the task's site is a simple acknowledgment that does not affect the barrier's latency, which is one message.

We generalize this idea to multiple tasks. The general scheme is as follows:

```
local X1 ... Xn in
   thread S1 X1=unit end
   thread S2 X2=unit end
   ...
   thread Sn Xn=unit end
   {Wait X1} ... {Wait Xn} S
end
```

The main thread waits until all `Xi` are bound. When `Si` terminates then its thread binds `Xi=`**`unit`**. When all tasks terminate then all `Xi` are bound, so the main thread runs `S`.

Assume now that the tasks are distributed over a set of sites. Each `Xi` is owned by the main thread's site. Therefore binding `Xi=`**`unit`** sends a message from the task site to the main site. When all variables are bound, the main thread resumes execution. Concurrently, the main site sends a message back for each message it received. These messages do not affect the barrier's latency.

2.4.6 *Distributed Locking.* If a program fragment may be executed by many threads, then it is important to be able to guarantee mutual exclusion. A thread that attempts to execute the fragment should block and be queued. Multiple requests should all correctly block and be queued, independent of whether the threads are on the same site or on another site. We show that it is possible to implement this concisely and efficiently in the language. As explained in Section 2.3, Figure 10

```
proc {NewSimpleLock ?Lock}
   Cell = {NewCell unit}
in
   proc {Lock Code}
      Old New in
      try
         {Exchange Cell Old New} {Wait Old}  % Enter
         {Code}                               % Body
      finally New=unit end                    % Exit
   end
end
```

Fig. 10. Distributed locking.

shows one way to implement a lock that handles exceptions correctly.[3] If multiple threads attempt to access the lock body, then only one is given access, and the others are queued. The queue is a sequence of logic variables. Each thread blocks on one variable in the sequence, and will bind the next variable after it has executed the lock body. Each thread desiring the lock therefore references two logic variables: one to wait for the lock and one to pass the lock to the next thread. Each logic variable is referenced by two threads.

If the threads are on different sites, then the queue is distributed. A single message will be sent to transfer the lock from one site to another. This implements distributed token passing, which is a well-known distributed algorithm for mutual exclusion [Chow and Johnson 1997]. We explain how it works. When a thread tries to enter the lock body, the `Exchange` gives it access to the previous thread's `New` variable. The previous thread's site is `New`'s owner. When the previous thread binds `New`, the owner sends the binding to the next thread's site. This requires a single message.

2.4.7 *Remote Method Invocation (RMI).* Let us invoke an object from within a thread on a given site. Where will the object execute? On a network-transparent system there are several possible answers to this question. Here we give just enough information to justify our RMI implementation. For a full discussion of the issues we refer the reader to Van Roy et al. [1997; 1999]. In Mozart, objects synchronously migrate to the invoking site by default. Therefore the object executes locally with respect to the invoking thread. This makes it easy for the object to synchronize with respect to the thread. If the object raises an exception, then it is passed to the thread. Object migration is implemented by a lightweight mobility protocol that serializes the path of the object's concurrent state pointer among the invoking sites.

It is possible in Oz to define a generic procedure that takes any object and returns a stationary object, i.e., such that all its methods will execute on the same site. This works because Oz has first-class messages and dynamic typing [Henz 1997a]. This is not possible in Java [Gosling et al. 1996]. Figure 11 defines `NewStationary`, which, given any object class, creates a stationary object of that class. It works by

[3]A thread-reentrant lock is defined in Van Roy et al. [1997].

```
proc {NewStationary Class Init ?StatObj}
   Obj={New Class Init}
   S P={NewPort S}
   N={NewName}
in
   proc {StatObj M}
      R in
      {Send P M#R}
      if R=N then skip
      else raise R end
      end
   end
   thread
     {ForAll S
      proc {$ M#R}
         thread
            try {Obj M} R=N
            catch X then R=X end
         end
      end}
   end
end
```

Fig. 11. RMI definition: Create a stationary object from any class.

```
% Create class Counter on local site
class Counter
   attr i
   meth init i <- 0 end
   meth inc  i <- @i+1 end
   meth get(X) X=@i end
   meth error raise e(someError) end end
end

% Create object Obj on remote site
{CS proc {$} Obj={NewStationary Counter init} end}

% Invoke object from local site
{Obj inc}
{Obj inc}
local X in {Obj get(X)} {Print X} end
try {Obj error} catch X then {Print X} end
```

Fig. 12. RMI example: A stationary counter object.

```
public class List {
   final unknown int car;
   final unknown List cdr;

   List(unknown int car, unknown List cdr) {
      this.car=:=car;
      this.cdr=:=cdr;
   }
   public void cons(unknown int car, unknown List cdr) {
      this.car=:=car;
      this.cdr=:=cdr;
   }
}
```

Fig. 13. List implementation in CC-Java.

wrapping the object inside a port, which is a stationary entity to which messages can be sent asynchronously. Therefore the object always executes on the same site, namely the site on which it was created. As before, the object synchronizes with respect to the invoking thread, and exceptions are passed to the invoking thread. The logic variable `R` is used both to synchronize and to pass back exceptions.

Figure 12 defines `Obj` remotely and invokes it from the local site. For example, `{Obj get(X)} {Print X}` queries the object and prints the result on the local site. The object binds the variable `X` to the current count. Since the local site owns `X`, the binding request sends one message from the remote site to the local site. With the initial invocation, this gives a total message latency of two for the remote call, just like an RPC. There is a third message back to the remote site that does not affect the message latency.

2.5 Adding Logic Variables to Other Languages

This section shows how to add logic variables in an orthogonal way to Java and ML, representative examples of object-oriented and functional languages.

2.5.1 *Java.* Sundström [1998] has recently defined and implemented a Java variant, CC-Java (Concurrent Constraint Java), which replaces monitors by logic variables and adds statement-level thread creation. Except for these differences, CC-Java has the same syntax and semantics as Java.

CC-Java provides logic variables through a single new modifier, `unknown`, which can be used in declarations of local variables, fields, formal parameters, and functions. For example, a variable `i` declared as `unknown int i;` is initially assigned an unknown value. Standard Java variables can be freely replaced by unknown variables. The result is always a legal CC-Java program. Variables with Java types will never be assigned unknown values—any attempt will block the thread until the value is known.

An `unknown` variable is bound by the new operator "`=:=`", which does unification. Each of the two operands can be known (i.e., be a standard Java variable) or unknown. Doing `i=:=23` binds `i` to 23. For correctness, the assignment operator "`=`" must overwrite (not unify) any reference to an `unknown` variable on the left-hand side. Declaring an `unknown` variable as `final` means that it is only assigned

```
public class StreamExample {
   // Return list of integers from n to max-1
   static List generate(int n, int max) {
      final unknown List l;
      unknown List ptr=l;
      for (int i=n; i<max; i+=1) {
         final unknown List tail;
         ptr=:=new List(i,tail);
         ptr=tail;
      }
      ptr=:=null;
      return l;
   }

   // Return (a + sum of elements of list l)
   static int sum(unknown List l, int a) {
      int sum=a;
      unknown List ptr=l;
      while (ptr!=null) {
         final unknown int x;
         final unknown List ls;
         ptr.cons(x,ls); // Wait until ptr is a list pair
         sum+=x;         // Wait until x is an integer
         ptr=ls;
      }
      return sum;
   }

   // Generate a list and sum its elements
   public static void main(String[] args) {
      unknown List l;
      int sum;
      thread l=:=generate(0,1500); // Using 150000 would overflow
      sum=sum(l,0);
      System.out.println(sum);
   }
}
```

Fig. 14. Stream communication in CC-Java.

once, i.e., when it is declared. A `final unknown` variable is therefore equivalent to an Oz logic variable. An `unknown` variable is equivalent to an Oz cell that points to a logic variable.

Figure 13 shows how to implement lists in CC-Java. Each list pair contains two logic variables, and therefore lists can be partially instantiated just like in Oz. Using logic variables does not imply any memory penalty for lists: when compiled to Oz, a CC-Java list pair uses just two memory words. Threads can synchronize on the instantiation state of lists.

Figure 14 uses these lists to write the stream communication example of Figure 6 in CC-Java (see Section 2.4.2). The `thread` statement of CC-Java is used to generate the list in another thread. The example has been written in a natural style in Oz and CC-Java, where Oz uses recursion, while CC-Java uses iteration to define the `generate` and `sum` functions. Comparing the two examples, we see that there is very little difference in clarity between these two styles. Their run-time efficiencies are comparable.

When examining the CC-Java program, two observations can be made. First, the example has two synchronization points: the statements `ptr.cons(x,ls)` and `sum+=x` inside the `sum` function. The former waits until `ptr` contains a list pair, and the latter waits until `x` is an integer. Second, the example shows that both `final unknown` and `unknown` variables are useful. The former are used as fixed references to data structures. The latter are used in loops that use a different logic variable in each iteration. In the `sum` method, the assignment statement `ptr=ls` makes `ptr` point to `ls` instead of what it pointed to in the previous iteration.

It is straightforward to compile CC-Java to either Oz or Java. A prototype CC-Java-to-Oz compiler has been implemented that understands the full Java syntax and compiles most of the Java language. Benchmarks show that CC-Java and Oz have comparable performance on the Mozart implementation of Distributed Oz. Both CC-Java and Oz on Mozart have performance comparable to Java on JDK 1.1.4, except that threads are much faster in Mozart [Henz 1997a; 1997b].

We outline how to implement a CC-Java-to-Java compiler. All Java code that does not use logic variables is unchanged. For each class `C` of which `unknown` instances are declared, the compiler adds a second class definition `UnknownC` to the Java code. The class `UnknownC` includes all methods of `C` and additional methods to unify the variable and to obtain its value. At each point where the value of an object of class `UnknownC` is needed, the compiler inserts a call to obtain the value. If the value is not yet available, then the calling thread is blocked until the value becomes available through unification.

2.5.2 *ML.* Smolka [1998] has recently shown how logic variables can be added as a conservative extension to a functional language very similar to Standard ML. We outline how the extension is done. Several new operations are added, including the following:

—`lvar: unit -> 'a`. The operation `lvar()` creates a fresh logic variable and returns its address.

—`<-: 'a * 'a -> 'a`. The operation `x <- y` binds `x`, which must be a logic variable, to `y`.

—`==: 'a * 'a -> 'a`. The operation `x == y` unifies `x` and `y`. This raises an exception if `x` and `y` are not unifiable.

—`wait: 'a -> 'a`. The operation `wait x` is an identity function that blocks until its argument is nonvariable.

—`spawn` *e*. This operation spawns a new thread evaluating expression *e* and returns `()`.

Execution states map addresses to any ML entity including primitive values, records, and reference cells. Execution states are extended so that a state may also map an address to a logic variable or to another address. The entity corresponding to an address is obtained by iterating the state function until the result is no longer an address. This iteration is the dereferencing operation. If a thread needs an entity and encounters a logic variable, then it blocks until the entity is available.

With this extension, existing ML programs continue to work, and logic variables may be freely intermixed with ML entities. ML provides explicit stateful entities which are called *references* and behave like typed Oz cells. As in Oz and CC-Java, the combination of logic variables and state makes it easy to express powerful concurrent programming techniques. Smolka outlines the semantics of the extension and illustrates some of these programming techniques.

3. BASIC CONCEPTS AND NOTATION

This section introduces the basic concepts and notation used for the CU and RCU algorithms, which do centralized unification. Most of this notation remains valid for the distributed algorithms. The extra notation they need will be given later on.

3.1 Terms and Constraints

In the usual case, a variable will be bound to a data structure. However, because of unpredictable network behavior, it may also be necessary to bind variables to variables or data structures to data structures. The result should not depend on the order in which the bindings occur. This justifies using a constraint system (D,C) to model the data structures and their bindings [Jaffar and Maher 1994]. The domain D is the set of data structures of interest; for generality we assume these are rational trees, i.e., trees with only finitely many subtrees. A rational tree is a good model for data structures with pointers since the tree can be represented (though not uniquely represented) by a rooted directed graph. Unfolding the graph to remove its cycles yields the tree [Courcelle 1983; Podelski and Smolka 1997].

The constraints C model bindings; we assume they are equalities between terms that describe sets of rational trees. For example, the constraint $x = f(y)$ means that the trees described by the variable x all have a root labeled f and a single subtree, which is a tree described by the variable y. In this way, we express clearly what it means to bind terms that may contain unbound variables. If y is unbound, then nothing is known about the subtree of x.

We introduce a uniform notation for terms. A term can either be a variable or a nonvariable term. Terms are denoted by u, v, w. Variables are denoted by x, y, z. Nonvariable terms are denoted by t, t_1, t_2. A nonvariable term is a record of the form $f(x_1, ..., x_n)$ with arity $n \geq 0$, where x_1, ..., x_n are variables and where the label f is an atomic constant taken from a given set of constants. A constraint has the general form $\bigwedge_i u_i = v_i$ where u_i and v_i are terms. A *basic* constraint has the form $x = u$.

To bind u and v means to add the constraint $u = v$ to the system. This is sometimes called *telling* the constraint. The operation of binding u and v is called *unification*. This is implementable in a time and space essentially equivalent to that needed for manipulating data structures in imperative languages [Van Roy 1994]. For more on the constraint-solving aspects of unification see Jaffar and Maher [1994].

We assume a partial order between terms such that all variables are in a total order and such that all nonvariable terms are less than all variables. That is, we assume a transitive antisymmetric relation $\mathrm{less}(u, v)$ such that for any distinct variables x and y, exactly one of $\mathrm{less}(x, y)$ or $\mathrm{less}(y, x)$ holds. In addition, for any nonvariable term t and any variable x, $\mathrm{less}(t, x)$ holds. The algorithm uses the order to avoid creating binding cycles when binding variables to variables (e.g., x bound to y and y bound to x). This is especially important in a distributed setting.

3.2 Configurations

A *configuration* $c = (\alpha;\sigma;\mu)$ of a centralized execution is a triple containing an action α, a store σ, and a memo table μ:

$$\alpha = \bigwedge_i u_i = v_i \wedge \bigwedge_i \text{false} \wedge \bigwedge_i \text{true} \qquad \sigma = \bigcup_i x_i \leftarrow u_i \qquad \mu = \bigcup_i x_i = y_i$$

The action α is a multiset of three kinds of primitive actions, of the form $u = v$, false, and true. The equation $u = v$ is one kind of primitive action. The notation $x \leftarrow u$ represents the binding of x to u. The store is a set of bindings. All variables x_i in the store are distinct, and there are no cycles $x_{a_1} \leftarrow x_{a_2}$, ..., $x_{a_{n-1}} \leftarrow x_{a_n}$, $x_{a_n} \leftarrow x_{a_1}$. It is easy to show that configurations always have this form in the CU algorithm.

The notation $x \leftarrow u, \sigma$ will be used as shorthand for $\{x \leftarrow u\} \cup \sigma$. The function $\text{lhs}(\sigma) = \bigcup_i x_i$ gives the set of bound variables in σ, which are exactly the variables on the left-hand sides of the binding arrows.

The memo table μ is used to store previously encountered variable-variable equalities so that the algorithm does not go into an infinite loop when unifying terms representing rational trees with cycles. For example, consider the equation $x = y$ with store $x \leftarrow f(x) \wedge y \leftarrow f(y)$. Dereferencing x and y and decomposing the resulting equation $f(x) = f(y)$ gives $x = y$ again (see Section 4). This loop is broken by putting $x = y$ in the memo table and testing for its presence. The memo table has two operations, *ADD* and *MEM*, defined as follows:

$$\begin{array}{rcl}
\mathit{ADD}(x = y, \mu) & = & \mu \cup \{x = y\} \\
\mathit{ADD}(x = t, \mu) & = & \mu \\
\mathit{MEM}(x = u, \mu) & = & \text{true if } x = u \in \mu \\
\mathit{MEM}(x = u, \mu) & = & \text{false otherwise}
\end{array}$$

If the number of variables is finite, then the number of possible variable-variable equations in the memo table is finite also. Therefore all possible loops are broken. In the off-line case this is always true. In the on-line case this is true if the finite-size property holds (see Section 7).

3.3 Algorithms

We define an algorithm as a set of reduction rules, where a rule defines a transition relation between configurations. The algorithms in this article all have a straightforward translation into an efficient imperative pseudocode. We do not define the algorithms in this way, since this would complicate reasoning about them. Rule reduction is an atomic operation. If more than one rule is applicable in a given configuration, then one is chosen nondeterministically. A rule is defined according to the following diagram:

$$\begin{array}{c||c} \alpha & \alpha' \\ \hline \sigma;\ \mu & \sigma';\ \mu' \end{array} \quad C$$

A rule becomes applicable for a given action α when (1) the actual store matches the store σ given in the rule and (2) the optional condition C is satisfied. The rule's reduction atomically replaces the current configuration $(\alpha;\sigma;\mu)$ by the result configuration $(\alpha';\sigma';\mu')$.

Both the centralized and the distributed algorithms are defined in the context of a structure rule and a congruence:

$$\text{Structure} \qquad \frac{\alpha_1 \wedge \alpha_2}{\sigma;\, \mu} \left\| \frac{\alpha_1' \wedge \alpha_2}{\sigma';\, \mu'} \right. \quad \textbf{if} \quad \frac{\alpha_1}{\sigma;\, \mu} \left\| \frac{\alpha_1'}{\sigma';\, \mu'} \right.$$

$$\text{Congruence} \qquad \begin{cases} \alpha_1 \wedge \alpha_2 \equiv \alpha_2 \wedge \alpha_1 \\ \text{true} \wedge \alpha \equiv \alpha \end{cases}$$

Because of the congruence, the number of occurrences of the true action is irrelevant. This is not true of the other primitive actions, which form a multiset.

3.4 Executions

An *execution* e of a given algorithm is a (possibly infinite) sequence of configurations such that the first configuration is an initial configuration and such that each transition corresponds to the reduction of one rule:

$$c_1 \xrightarrow{R_1} c_2 \xrightarrow{R_2} \cdots \xrightarrow{R_{n-1}} c_n$$

In any execution, distributed or centralized, we assume that the rules are reduced in some total order. This is done without loss of generality, since the results we prove in this article will hold for *all* possible executions. Therefore, reductions that are not causally related may be assumed to execute in any order.

An *initial* configuration of the CU algorithm is of the form $(\alpha_1;\emptyset;\emptyset)$, where α_1 is a finite conjunction of equations, while the store and memo table are both empty. A *terminal* configuration (if it exists) is a configuration where no rules are applicable. The last configuration of a finite execution is not necessarily a terminal configuration, since rules may still be applicable. A *valid* configuration is one that is reachable by an execution of a given algorithm. We speak of centralized executions (using the CU or RCU algorithms; see Sections 4 and 6.2) and distributed executions (using the DU algorithm; see Section 5).

3.5 Adapting Unification to Reactive Systems

A store represents a logical conjunction of constraints. Adding a constraint that is inconsistent with the store results in the conjunction collapsing to a "false" state. This behavior is incompatible with a long-lived reactive system. Furthermore, it is expensive in a distributed system, since it requires a global synchronization. Rather, we want an inconsistent constraint to be flagged as such (e.g., by raising an exception), without actually being put in the store. This requires two modifications to the unification algorithm:

—**Incremental tell**, i.e., information that is inconsistent with the store is not put in the store [Smolka 1995]. The CU and DU algorithms both implement incremental tell by decomposing a rational tree constraint into the basic constraints $x = y$ and $x = t$ and by not collapsing the store when an inconsistency is detected with a basic constraint. Inconsistency is represented as a new action "false" instead of being incorporated into the store. This new action can be used to inform the program, e.g., by raising an exception.

—**Full failure reporting**, i.e., every case of inconsistency is flagged to the program. Neither the CU nor the DU algorithms do this. If the inconsistent equation

$x = y$ is present more than once, then the CU algorithm flags this only once. The DU algorithm flags the inconsistency of $(x = y)_s$ only once per site s. That is, both algorithms will flag an inconsistent equation $x = y$ with given variables x and y only once per memo table. Inconsistencies can be flagged more often by introducing more memo tables. This may sometimes do redundant work. For example, if equations reduce within a thread and each thread is given its own memo table, then multiple occurrences of an inconsistent equation will be flagged once per thread. The Mozart implementation will flag an inconsistency at least once per program-invoked unification (see Section 8).

4. CENTRALIZED UNIFICATION (CU) ALGORITHM

This section defines a centralized algorithm for rational tree unification. The definition is given as a set of transition rules with an interleaving semantics. That is, rule reductions do not overlap in time. Only their order is important. Nothing is lost by using an interleaving semantics, since we prove properties that are true for all interleavings that are valid executions [Alford et al. 1985].

The CU algorithm is the base case for the DU algorithm of Section 5. There are many possible variant definitions of unification. The definition presented here is deterministic and constrained by the requirement that it must be extensible to a practical distributed algorithm. That is, the use of global conditions is minimized (see Section 5.1).

4.1 Definition

We define the CU algorithm by the following seven rules.

$$\textsc{Interchange}\qquad \frac{u = x}{\sigma;\ \mu} \Bigg\| \frac{x = u}{\sigma;\ \mu} \qquad \mathrm{less}(u, x)$$

$$\textsc{Bind}\qquad \frac{x = u}{\sigma;\ \mu} \Bigg\| \frac{\mathrm{true}}{x \leftarrow u, \sigma;\ \mu} \qquad \mathrm{less}(u, x), x \notin \mathrm{lhs}(\sigma)$$

$$\textsc{Memo}\qquad \frac{x = u}{x \leftarrow v, \sigma;\ \mu} \Bigg\| \frac{\mathrm{true}}{x \leftarrow v, \sigma;\ \mu} \qquad \begin{array}{l} \mathrm{less}(u, x), \\ \mathit{MEM}(x = u, \mu) \end{array}$$

$$\textsc{Dereference}\qquad \frac{x = u}{x \leftarrow v, \sigma;\ \mu} \Bigg\| \frac{v = u}{x \leftarrow v, \sigma;\ \mathit{ADD}(x = u, \mu)} \qquad \begin{array}{l} \mathrm{less}(u, x), \\ \neg \mathit{MEM}(x = u, \mu) \end{array}$$

$$\textsc{Identify}\qquad \frac{x = x}{\sigma;\ \mu} \Bigg\| \frac{\mathrm{true}}{\sigma;\ \mu}$$

$$\textsc{Conflict}\qquad \frac{t_1 = t_2}{\sigma;\ \mu} \Bigg\| \frac{\mathrm{false}}{\sigma;\ \mu} \qquad \begin{array}{l} t_1 = f_1(x_1, \ldots, x_m),\ t_2 = f_2(y_1, \ldots, y_n), \\ (f_1 \neq f_2 \vee m \neq n) \end{array}$$

$$\textsc{Decompose}\qquad \frac{t_1 = t_2}{\sigma;\ \mu} \Bigg\| \frac{\bigwedge_{1 \le i \le n} x_i = y_i}{\sigma;\ \mu} \qquad \begin{array}{l} t_1 = f(x_1, \ldots, x_n), \\ t_2 = f(y_1, \ldots, y_n) \end{array}$$

4.2 Properties

This section presents a basic result from unification theory, namely the total correctness of the CU algorithm. With this result, we can prove correctness of the distributed unification algorithms by reducing them to centralized unification.

Logical Formula of a Configuration. A configuration $c = (\alpha;\sigma;\_)$ has an associated logical formula $\varepsilon(c) = \varepsilon_a(\alpha) \wedge \varepsilon_s(\sigma)$, where

$$\begin{array}{lcl}
\varepsilon_a(\alpha_1 \wedge \alpha_2) & = & \varepsilon_a(\alpha_1) \wedge \varepsilon_a(\alpha_2) \\
\varepsilon_a(u = v) & = & u = v \\
\varepsilon_a(\text{true}) & = & \text{true} \\
\varepsilon_a(\text{false}) & = & \text{false} \\
 & & \\
\varepsilon_s(\sigma_1 \cup \sigma_2) & = & \varepsilon_s(\sigma_1) \wedge \varepsilon_s(\sigma_2) \\
\varepsilon_s(\{x \leftarrow u\}) & = & x = u \ .
\end{array}$$

THEOREM (LOGICAL EQUIVALENCE PROPERTY). *In every transition $c_i \rightarrow c_{i+1}$ of every execution of the CU algorithm, the logical equivalence $\varepsilon(c_i) \leftrightarrow \varepsilon(c_{i+1})$ holds under the standard equality theory.*

PROOF. By standard equality theory we mean the theory $\mathcal{E}$ given by Lloyd [1987, p. 79], minus the acyclicity axioms (i.e., rule 4). This theory has the usual axioms implying nonequality of distinct symbols; term equality implies argument equality and vice versa; substitution by equals is allowed; and identity. What we want to prove is $\mathcal{E} \models \bar{\forall}\varepsilon(c_i) \leftrightarrow \varepsilon(c_{i+1})$, where the quantification is over all free variables. This is a standard result in unification theory [Haridi 1981; Colmerauer 1982; Martelli and Montanari 1982; Haridi and Sahlin 1984]. □

COROLLARY (ENTAILMENT PROPERTY OR CU TOTAL CORRECTNESS). *Given any initial configuration $c_1 = (\alpha_1;\emptyset;\emptyset)$ of the CU algorithm, then the algorithm always reaches a terminal configuration $c_n = (\alpha_n;\sigma_n;\_)$ with $\varepsilon(c_n) \leftrightarrow \varepsilon(c_1)$. Furthermore, α_n consists of zero or more false actions, and if there are zero, then $\varepsilon_s(\sigma_n) \leftrightarrow \varepsilon_a(\alpha_1)$.*

PROOF. Again, this is a standard result in unification theory. The equivalence follows from the previous theorem. □

5. DISTRIBUTED UNIFICATION (DU) ALGORITHM

This section defines a distributed algorithm for rational tree unification. The section is organized as follows. Section 5.1 explains how to generalize the CU algorithm to a distributed setting. Section 5.2 sets the stage by giving the basic concepts and notation needed in the DU algorithm. Section 5.3 gives an example execution. Section 5.4 defines the DU algorithm in two parts: the nonbind rules, which are the local part of the algorithm, and the bind rules, which are the distributed part. Finally, Section 5.5 compares the CU and DU algorithms from the viewpoint of the dereference operation.

5.1 Generalizing CU to a Distributed Setting

A distributed algorithm must be defined by reduction rules that do local operations only, since these are the only rules we can implement directly. To be precise, two conditions must be satisfied. First, testing whether a rule is applicable should require looking only at one site. Second, reducing the rule should modify only that site, except that the rule is allowed to create actions annotated with other sites. In the distributed system these actions correspond to messages. Rules that satisfy these two conditions are called *local* rules. A distributed algorithm defined in terms of local rules is a *transition system* in an asynchronous non-FIFO network [Tel 1994].

We would like to extend each CU rule to become a local rule in the distributed setting. In this way, we maintain a close correspondence between the centralized and distributed algorithms, which simplifies analysis of the distributed case. Furthermore, this minimizes costly communication between sites.

The first step is to annotate the rule's actions and bindings with sites. Each CU rule reduces an input action and may inspect a binding in the store. We annotate the input action by its site and the binding by the same site. This is correct if we assume that a binding will eventually appear on each site that references the variable. We annotate the output action by the same site as the input action. A "true" output action does not need a site. Actions may remain unannotated, in which case the DU algorithm does not specify where they are reduced. This set of annotations suffices for the rules INTERCHANGE, IDENTIFY, CONFLICT, and DECOMPOSE to become DU rules. An important property of CONFLICT is that an inconsistency is always flagged on the site that causes it.

The three remaining CU rules cannot be so easily extended, since they have global conditions. To be precise, BIND has the unboundness condition $x \notin \text{lhs}(\sigma)$,[4] and MEMO and DEREFERENCE both have the memoization condition $MEM(x = u, \mu)$. It turns out that the memoization condition can be relaxed in the distributed algorithm, so that it becomes a local condition there. In this way, the MEMO and DEREFERENCE rules become local rules. The idea is to give each site its own memo table, which is independent of the other memo tables. Section 6.2 proves that this relaxation is correct, but that redundant local work may be done.

The unboundness condition of BIND cannot be eliminated in this way. Implementing it requires communication between sites. The single BIND rule therefore becomes *several* local rules in the distributed setting. The BIND rule is replaced by four rules that exchange messages to implement a coherent variable elimination algorithm.

The resulting DU algorithm consists of 10 local rules, namely six nonbind rules (Section 5.4.1) and four bind rules (Section 5.4.2). The six nonbind rules do not send any messages. Of the four bind rules, only INITIATE and WIN send messages. All rules test applicability by looking at one site only, except for WIN and LOSE, which use information tied to a variable but not tied to any particular site, namely a flag unbound(x)/bound(x) and a binding request $(x \sim u)$.

5.2 Basic Concepts and Notation

We introduce a set $S = \{1, ..., k\}$ of k sites, where $k \geq 1$. We model distributed execution by *placing* each action and each binding on a site. A primitive action or binding ξ is placed on site s by adding parentheses and a subscript $(\xi)_s$. The same ξ may be placed on several sites, in which case the resulting actions or bindings are considered separately. A configuration of a distributed execution is a triple $(A;\Sigma;M)$ consisting of an action A, a store Σ, and a memo table M. We denote the action, store, and memo table on site s by A_s, Σ_s, and M_s, respectively.

[4]The opposite condition, confirming the existence of a binding, is local.

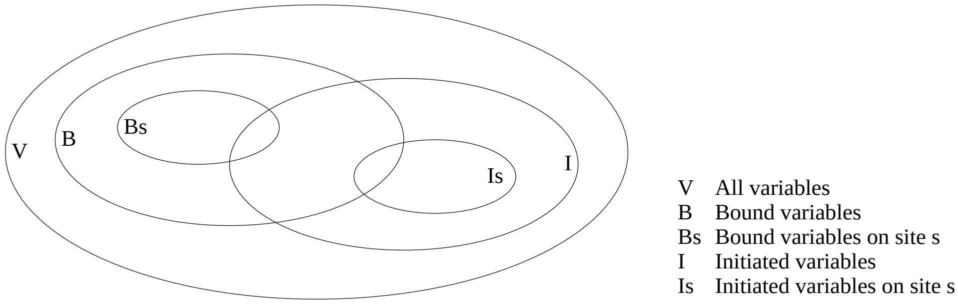

Fig. 15. Bound variables and initiated variables.

5.2.1 *Store.* The store Σ contains sited bindings $(x \leftarrow u)_s$, sited binding initiations $(x \leftarrow \bot)_s$, and flags to denote whether a variable is bound or not ($\text{bound}(x)$ or $\text{unbound}(x)$). A store has the form

$$\begin{aligned}
\Sigma &= \Gamma \cup \bigcup_{s \in S} \Sigma_s \\
\Sigma_s &= \bigcup_{x_i \in B_s} (x_i \leftarrow u_i)_s \cup \bigcup_{x_i \in I_s} (x_i \leftarrow \bot)_s \\
\Gamma &= \bigcup_{x_i \in B} \text{bound}(x_i) \cup \bigcup_{x_i \in V-B} \text{unbound}(x_i).
\end{aligned}$$

It is easy to show that configurations always have this form in the DU algorithm. The set V consists of all variables in A and Σ. The set $B \subseteq V$ contains all the *bound* variables. The set $B_s \subseteq B$ contains all the bound variables whose binding is known on site s. The set $I \subseteq V$ contains all the variables whose binding has been *initiated* on some site but whose binding (if it exists) is not yet known on that site. The set $I_s \subseteq I$ contains all the variables whose binding has been initiated on site s but is not yet known on that site. In terms of the bind rules of Section 5.4.2, B corresponds to those variables for which the Win rule has reduced. I corresponds to those variables for which the Initiate rule has reduced but the corresponding Arrive rule has not yet reduced. Figure 15 illustrates the relationship between these five sets.

Two utility functions are used in the algorithm definition:

$$\begin{aligned}
\text{lhs}(\Sigma_s) &= \{x | \exists u.(x \leftarrow u)_s \in \Sigma_s \vee (x \leftarrow \bot)_s \in \Sigma_s\} \;=\; B_s \cup I_s \\
\text{var}(\Sigma_s) &= \text{lhs}(\Sigma_s) \cup \{x | \exists y, u.((y \leftarrow u)_s \in \Sigma_s \wedge (u \equiv f(..., x, ...) \vee u \equiv x))\}
\end{aligned}$$

The function $\text{lhs}(\Sigma_s)$ returns all bound and initiated variables of Σ_s. It generalizes the function $\text{lhs}(\sigma)$ defined in Section 3.2. The function $\text{var}(\Sigma_s)$ returns all variables mentioned in Σ_s, including variables that are neither bound nor initiated.

5.2.2 *Initial Configuration.* The initial configuration is $(A^{\text{init}};\Sigma^{\text{init}};\emptyset)$, with initial actions A^{init} that are all equations and $\Sigma^{\text{init}} = \{\text{unbound}(x_i) \mid x_i \in V\}$. We have initially $B = \emptyset$ and $I = \emptyset$.

5.2.3 *Action.* An action A is a multiset containing two new primitive actions in addition to the three primitive actions of the centralized setting (see Table I). The

Table I. Actions in Distributed Configurations

Same as centralized setting	
true	Null action
false_s	Failure notification on site s
$(u = v)_s$	Equation on site s
New for distributed setting	
$x \sim u$	Binding request
$(x \Leftarrow u)_s$	Binding in transit to site s

new actions are needed to implement the distributed operations of the algorithm. The exact meaning of these actions is defined by the reduction rules that manipulate them. Intuitively, the action $x \sim u$ represents a message requesting the binding of x to u. For a given x, exactly one such action will cause a binding to be made; all others are discarded. The algorithm does not specify where the binding decision is made. An actual implementation can make the decision in a centralized or distributed way. In the Mozart implementation, the decision is centralized; it is made on the site where the variable was initially declared (see Section 8). The action $(x \Leftarrow u)_s$ represents a message containing a binding to x that may eventually become visible in Σ_s. As long as x is not in $\text{var}(\Sigma_s)$ then the binding stays "in the network."

5.2.4 *Memo Table.* The global memo table M is the juxtaposition of all the local memo tables. That is, $M = \{(x = y)_s | x = y \in M_s\}$. Each local memo table M_s is a set of variable-variable equalities that has identical structure to the centralized memo table μ.

5.3 An Example

Figure 16 gives an example execution that does distributed variable elimination. In this figure, thin solid arrows represent actions or bindings. Vertical bars "**|**" denote rule reductions, which happen in the numbered order shown. Thin dotted arrows represent causal links.

Initially, site 1 has equation $(x = a)_1$, and site 2 has equation $(x = b)_2$. Both sites do an Initiate rule, which puts binding initiations $(x \leftarrow \perp)_s$ in both local stores. This ensures that the two equations cannot reduce until the binding arrives. We say that the equations are *suspended*. Equation $(x = b)_2$ is the first to do a Win rule, and b therefore becomes the global binding of x. The other equation is discarded by the Lose rule. The binding $(x \Leftarrow b)$ is sent to all sites. It arrives at each site through the Arrive rule. At that point, the suspended equations $(x = a)_1$ and $(x = b)_2$ become reducible again. The equation $(x = a)_1$ will cause an inconsistency to be flagged on site 1.

5.4 Definition

The DU algorithm has a close relationship to the CU algorithm. The structure and congruence rules continue to hold in the distributed setting. The only centralized rule that is changed is the Bind rule; it is replaced by the four bind rules below. It is clear from inspection that all six nonbind DU rules have identical behavior to the corresponding CU rules, if the CU rules are considered as acting on a single site.

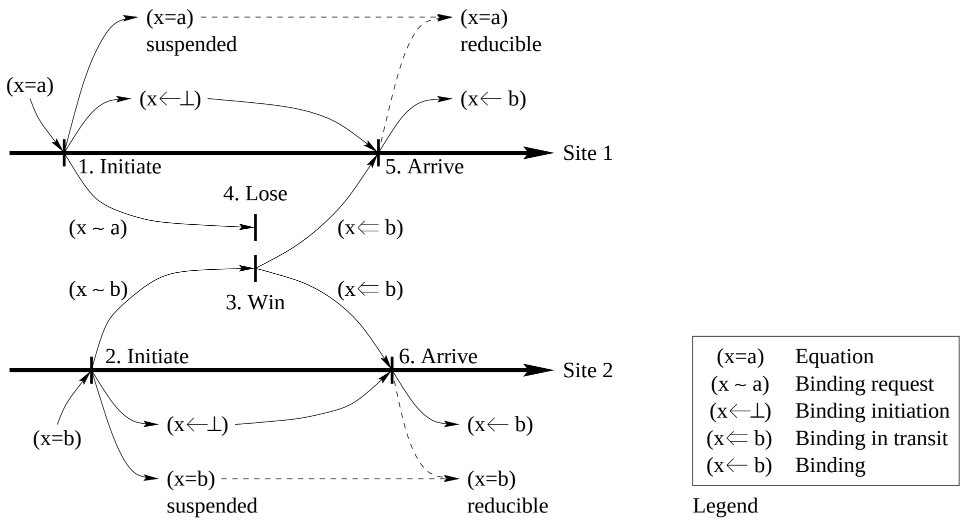

Fig. 16. Distributed unification with $(x = a)_1$ and $(x = b)_2$.

5.4.1 *Nonbind Rules.* These rules correspond exactly to the nonbind rules of the centralized algorithm. An inconsistency is flagged on the site that causes it by the action false_s.

Interchange
$$\frac{(u = x)_s}{\Sigma;\ M} \;\Big\|\; \frac{(x = u)_s}{\Sigma;\ M} \qquad \text{less}(u, x)$$

Memo
$$\frac{(x = u)_s}{\begin{array}{c}(x \leftarrow v)_s, \Sigma;\\ M_s \cup M\end{array}} \;\Big\|\; \frac{\text{true}}{\begin{array}{c}(x \leftarrow v)_s, \Sigma;\\ M_s \cup M\end{array}} \qquad \begin{array}{l}\text{less}(u, x),\\ \mathit{MEM}(x = u, M_s)\end{array}$$

Dereference
$$\frac{(x = u)_s}{\begin{array}{c}(x \leftarrow v)_s, \Sigma;\\ M_s \cup M\end{array}} \;\Big\|\; \frac{(v = u)_s}{\begin{array}{c}(x \leftarrow v)_s, \Sigma;\\ \mathit{ADD}(x = u, M_s) \cup M\end{array}} \qquad \begin{array}{l}\text{less}(u, x),\\ \neg\mathit{MEM}(x = u, M_s)\end{array}$$

Identify
$$\frac{(x = x)_s}{\Sigma;\ M} \;\Big\|\; \frac{\text{true}}{\Sigma;\ M}$$

Conflict
$$\frac{(t_1 = t_2)_s}{\Sigma;\ M} \;\Big\|\; \frac{\text{false}_s}{\Sigma;\ M} \qquad \begin{array}{l}t_1 = f_1(x_1, ..., x_m),\ t_2 = f_2(y_1, ..., y_n),\\ (f_1 \neq f_2 \vee m \neq n)\end{array}$$

Decompose
$$\frac{(t_1 = t_2)_s}{\Sigma;\ M} \;\Big\|\; \frac{\bigwedge_{1 \leq i \leq n}(x_i = y_i)_s}{\Sigma;\ M} \qquad \begin{array}{l}t_1 = f(x_1, ..., x_n),\\ t_2 = f(y_1, ..., y_n)\end{array}$$

5.4.2 *Bind Rules.* These rules replace the Bind rule of the centralized algorithm. The binding initiation $(x \leftarrow \perp)_s$ and the condition $x \notin \text{lhs}(\Sigma_s)$ in the Initiate rule together ensure that only one binding attempt can be made per site.

Initiate
$$\frac{(x = u)_s}{\Sigma;\ M} \;\Big\|\; \frac{x \sim u \wedge (x = u)_s}{(x \leftarrow \perp)_s, \Sigma;\ M} \qquad \text{less}(u, x),\ x \notin \text{lhs}(\Sigma_s)$$

$$\textsc{Win} \quad \frac{x \sim u \;\Big\|\; \bigwedge_{s \in S}(x \Leftarrow u)_s}{\text{unbound}(x), \Sigma;\ M \;\Big\|\; \text{bound}(x), \Sigma;\ M}$$

$$\textsc{Lose} \quad \frac{x \sim u \;\Big\|\; \text{true}}{\text{bound}(x), \Sigma;\ M \;\Big\|\; \text{bound}(x), \Sigma;\ M}$$

$$\textsc{Arrive} \quad \frac{(x \Leftarrow u)_s \;\Big\|\; \text{true}}{\Sigma;\ M \;\Big\|\; (x \leftarrow u)_s, \Sigma - \{(x \leftarrow \bot)_s\};\ M} \quad x \in \text{var}(\Sigma_s)$$

5.5 Dereference Chains

A *dereference chain* in a store σ is a sequence of bindings $x_1 \leftarrow x_2$, ..., $x_{n-1} \leftarrow u_n$ with $n \geq 1$ and u_n unbound or nonvariable. We say the value of x_1 in store σ is u_n. To find u_n given x_1, it is necessary to follow the chain. A major difference between CU and DU is that CU always constructs dereference chains, whereas DU with eager variables forbids dereference chains to cross sites. Instead, DU copies remote terms to make them local. In a centralized setting, pointer dereferencing is fast, so the penalty of using dereference chains is small. This makes sharing terms very cheap. In a distributed setting, pointer dereferencing across sites is slow, and it makes the current site dependent on the other site. This makes copying terms preferable.

Copying terms instead of creating dereference chains introduces redundant work. It is possible to reduce this work at the cost of more network operations. For example, one can eagerly bind to variables and lazily bind to (big) nonvariable terms. This guarantees that a cross-site dereference chain has a maximum length of one.

DU with lazy variables allows dereference chains to cross sites. When the value is needed, the binding is requested from the owner. If the binding is another variable, then the process is repeated. Each iteration of this process corresponds to a dereference operation.

Taylor [1991] presents a centralized binding algorithm that avoids all dereference chains. Variables that are bound together are put into a circular linked list. When one of them is bound to a value, the list is traversed and all members are bound. This makes accessing a variable's value a constant-time operation, at the price of making binding more expensive. Taylor finds no significant performance difference between this algorithm and the standard algorithm when both are embedded in a Prolog system whose performance is comparable to a good C implementation.

6. OFF-LINE TOTAL CORRECTNESS

This section proves that the DU algorithm behaves as expected. We first define a mapping from any distributed to a centralized execution. Then we define a modification of the CU algorithm, the RCU algorithm, that models the redundant work done by the distributed algorithm. We prove safety and liveness properties of the DU algorithm by reducing it to the RCU algorithm. From this we show that the DU algorithm is correct.

We distinguish between the off-line total correctness and the on-line total correctness. In the off-line case, we have to show that the distributed algorithm terminates and gives correct results for any placement of a fixed set of initial equations. This can be done without any fairness assumptions. This is not true for the on-line case,

Table II. Mapping from Distributed to Centralized Configurations

	Distributed	Centralized
Action	true	true
	false_s	false
	$(u = v)_s$	$u = v$
	$x \sim u$	true
	$(x \Leftarrow u)_s$	$x \leftarrow u$
Store	bound(x)	true
	unbound(x)	true
	$(x \leftarrow \bot)_s$	true
	$(x \leftarrow u)_s$	$x \leftarrow u$

which is handled in Section 7.

6.1 Mapping from Distributed to Centralized Executions

The proofs in this section are based on a mapping m from any distributed configuration $(A;\Sigma;M)$ to a corresponding centralized configuration $(\alpha;\sigma;\mu)$. Distributed executions are mapped to centralized executions by mapping each of their configurations. The mapping m was designed following the reasoning of Section 5.1. We show that m has very strong properties that lead directly to a proof that the distributed algorithm implements the centralized algorithm.

A primitive action is mapped to either a primitive action or a binding. A binding is always mapped to a binding. Other store contents map to true. In this way we can map any distributed configuration to a centralized one:

$$(A; \Sigma; M) \xrightarrow{m} (\alpha; \sigma; \mu) = (m_a(A); m_s(A, \Sigma); m_m(M))$$

Table II defines the mappings m_a and m_s for primitive actions and store contents. The mapping for all of A and Σ is the union of these mappings for all primitive actions in A and all store contents in Σ. The centralized memo table $m_m(M)$ is derived from the local memo tables M_s as follows. For each $x = y$ such that $\exists s : x = y \in M_s$, the centralized memo table contains $(x = y, i)$, where i is the number of tables M_s that contain $x = y$:

$$m_m(M) \; = \; \{(x = y, i) | i = \#\{s | x = y \in M_s\} \wedge i > 0\}$$

The following diagram relates a distributed execution e and its corresponding centralized execution $m(e)$:

$$\begin{array}{ccc} (A; \Sigma; M) & \xrightarrow{e} & (A'; \Sigma'; M') \\ m \downarrow & & \downarrow m \\ (\alpha; \sigma; \mu) & \xrightarrow{m(e)} & (\alpha'; \sigma'; \mu') \end{array}$$

To show total correctness, i.e., that the distributed algorithm is an implementation of unification, we need to show both safety and liveness properties. Section 6.3 proves a sufficient safety property: given any distributed execution e, the corresponding $m(e)$ is a correct centralized execution. Section 6.4 proves a sufficient liveness property: given any e, its execution eventually makes progress. That is, if the last configuration of $m(e)$ is nonterminal, then the last configuration of e is nonterminal, and continuing e will always eventually advance $m(e)$. In the distributed execution, the nonbind rules and the WIN rule are called *progressing* rules, since they advance the centralized execution (see Table III). The other rules are called *nonprogressing*.

6.2 Redundant Centralized Unification (RCU) Algorithm

This section defines and justifies a revised version of the CU algorithm, the RCU algorithm, that models the redundant work introduced by distributing rational tree unification. There are two sources of redundant work in the DU algorithm. The first source is due to the decoupling of binding initiation from binding arrival. A binding initiation for $(x = u)_s$ inhibits reduction of all equations of the form $(x = v)_s$. When a binding arrives on a site, these reductions become possible again, including the reduction of the original equation $(x = u)_s$. To make the original equation disappear, several rule reductions are needed including a DEREFERENCE, one or more IDENTIFY, and possibly a DECOMPOSE. This redundant work can be avoided in the implementation (see Section 8.5.4).

The second source of redundant work cannot be avoided in the implementation. It is due to each site having its own local memo table. Memo tables are needed because of rational trees with cycles. However, they are in fact a general caching technique that avoids doing any unification more than once. In the distributed algorithm, the information stored in each site's memo table is not seen by the other sites. Therefore each site has to reconstruct the part of the centralized memo table that it needs.

To model the local memo tables it suffices to weaken the memo table membership check. This affects the two rules MEMO and DEREFERENCE. Assume there are k sites. We introduce a weaker membership check MEM_k that is true if and only if the equation has been entered at least k times. This is implemented by extending the memo table to store pairs of an equation and the number of times the equation has been entered:

$$
\begin{array}{rcl}
ADD(x = y, \mu) & = & \mu \cup \{(x = y, 1)\} \text{ if } (x = y, \_) \notin \mu \\
ADD(x = y, \mu) & = & \mu - \{(x = y, i)\} \cup \{(x = y, i+1)\} \text{ if } (x = y, i) \in \mu \\
ADD(x = t, \mu) & = & \mu \\
MEM(x = u, \mu) & = & \text{true if } (x = u, \_) \in \mu \\
MEM(x = u, \mu) & = & \text{false otherwise} \\
MEM_k(x = u, \mu) & = & \text{true if } (x = u, i) \in \mu \wedge i \geq k \\
MEM_k(x = u, \mu) & = & \text{false otherwise}
\end{array}
$$

The R-MEMO rule uses the new definition of *MEM*. The R-DEREFERENCE rule uses MEM_k and the new definition of *ADD*. If an equation has been entered from 1 to $k-1$ times then both rules are applicable. This is an example of using nondeterminism to model distributed behavior in a centralized setting.

Now we can update the CU algorithm to model the two sources of redundant work. We model memo table redundancy by replacing MEMO and DEREFERENCE by R-MEMO and R-DEREFERENCE. We model bind redundancy by replacing BIND by R-BIND, as defined below. The three new rules are as follows:

R-BIND $\quad \dfrac{x = u}{\sigma;\ \mu} \Bigg\| \dfrac{x = u}{x \leftarrow u, \sigma;\ \mu} \quad \text{less}(u, x), x \notin \text{lhs}(\sigma)$

R-MEMO	$\dfrac{x = u}{x \leftarrow v, \sigma;\ \mu}$	$\dfrac{\text{true}}{x \leftarrow v, \sigma;\ \mu}$	$\text{less}(u, x)$, $MEM(x = u, \mu)$
R-DEREFERENCE	$\dfrac{x = u}{x \leftarrow v, \sigma;\ \mu}$	$\dfrac{v = u}{x \leftarrow v, \sigma;\ ADD(x = u, \mu)}$	$\text{less}(u, x)$, $\neg MEM_k(x = u, \mu)$

THEOREM (RCU TOTAL CORRECTNESS). *Given any initial configuration, the following two statements hold:*

(*1*) *The RCU algorithm terminates.*

(*2*) *All terminal configurations of the RCU and CU algorithms are logically equivalent to each other according to the definition of Section 4.2.*

PROOF. We handle termination and correctness separately.

(1) We know that CU terminates. The redundant work introduced by RCU has the following effect:
  —**Bind redundancy**. The R-BIND rule introduces extra rule reductions. The number of extra reductions is 2 if u is a variable and $2+a$ if u is a nonvariable and a is its arity.
  —**Memo table redundancy.** The memo table size for RCU is at most k times that of CU, which is finite. Hence only a finite number of extra rule reductions can be done.

(2) For both bind and memo table redundancy, the additional equations are always duplicates of existing equations or equations of some previous configuration. Therefore they add no additional information, and the Entailment property still holds.

This completes the proof. □

6.3 Safety

THEOREM (DU SAFETY). *If e is any execution of the DU algorithm, then $m(e)$ is an execution of the RCU algorithm, and the sequence of rules reduced in $m(e)$ can be constructed from e.*

PROOF. We will prove that Table III correctly gives the centralized rule of $m(e)$ corresponding to a distributed rule in e. A "SKIP" rule means that no rule is executed. The proof is by induction on the length of execution e. In the base case, the initial configuration c_1 of e has an empty store and memo table, and a set of equations placed on different sites. Therefore $m(c_1)$ is a valid initial configuration for the centralized algorithm.

In the induction case, we assume that the theorem holds for an execution e. We need to show that for each distributed rule applicable in the last configuration of e, that doing this rule maps to doing a corresponding centralized rule. We do a case analysis over the distributed rules. Section 6.3.1 covers the nonbind rules and Section 6.3.2 covers the bind rules.

6.3.1 *Nonbind Rules.*

Table III. Correspondence between Distributed and Centralized Rules

Distributed rule	Centralized rule
MEMO	R-MEMO
DEREFERENCE	R-DEREFERENCE
INTERCHANGE	INTERCHANGE
IDENTIFY	IDENTIFY
CONFLICT	CONFLICT
DECOMPOSE	DECOMPOSE
INITIATE	SKIP
WIN	R-BIND
LOSE	SKIP
ARRIVE	SKIP

6.3.1.1 *Decompose.* Assume that the distributed execution reduces a DECOMPOSE rule. Mapping the before and after configurations of the decomposition gives the following diagram:

$$\begin{array}{ccc} \dfrac{(t_1 = t_2)_s \wedge A}{\Sigma; M} & \xrightarrow{DEC} & \dfrac{\bigwedge_i (x_i = y_i)_s \wedge A}{\Sigma; M} \\ m \downarrow & & \downarrow m \\ \dfrac{t_1 = t_2 \wedge m_a(A)}{m_s(A,\Sigma); m_m(M)} & \xrightarrow{X} & \dfrac{\bigwedge_i x_i = y_i \wedge m_a(A)}{m_s(A,\Sigma); m_m(M)} \end{array}$$

It is clear from inspection that rule X is a centralized decomposition.

6.3.1.2 *Interchange, Identify, and Conflict.* These three rules are handled in the same way as the DECOMPOSE rule.

6.3.1.3 *Memo.* It is clear that the MEMO rule maps correctly to an R-MEMO rule, since from $M_s \subseteq \mu$ it follows that $MEM(x = u, M_s) \Rightarrow MEM(x = u, \mu)$. To be precise, the elements of μ have the form $(x = y, i)$, but the i can be safely ignored (see the two definitions of the MEM function).

6.3.1.4 *Dereference.* We now show that the DEREFERENCE rule maps to an R-DEREFERENCE rule. We have the following diagram (where $\Sigma_x = (x \leftarrow v)_s, \Sigma)$:

$$\begin{array}{ccc} \dfrac{(x = u)_s \wedge A}{\Sigma_x; M_s \cup M} & \xrightarrow{DRF} & \dfrac{(v = u)_s \wedge A}{\Sigma_x; ADD(x = u, M_s) \cup M} \\ m \downarrow & & \downarrow m \\ \dfrac{x = u \wedge m_a(A)}{m_s(A,\Sigma_x); m_m(M_s \cup M)} & \xrightarrow{X} & \dfrac{v = u \wedge m_a(A)}{m_s(A,\Sigma_x); m_m(ADD(x = u, M_s) \cup M)} \end{array}$$

We know that $\mathrm{less}(u, x) \wedge \neg MEM(x = u, M_s)$. Since each site has its own local memo table, there can be only one redundant equation per site. Therefore $\neg MEM(x = u, M_s)$ implies that $\neg MEM_k(x = u, \mu)$. That is, if at least one local memo table does not contain $x = u$, then the centralized memo table contains $x = u$ less than k times. It follows that rule X is an R-DEREFERENCE rule. This shows that relaxing the memoization condition to do only a check on the local part of the memo table is correct but may introduce one redundant equation per site.

6.3.2 *Bind Rules.*

6.3.2.1 *Initiate.*

$$\begin{array}{ccc} \dfrac{(x = u)_s \wedge A}{\Sigma; M} & \xrightarrow{\mathit{INI}} & \dfrac{x \sim u \wedge (x = u)_s \wedge A}{(x \leftarrow \bot)_s \Sigma; M} \\ m \downarrow & & \downarrow m \\ \dfrac{x = u \wedge m_a(A)}{m_s(A, \Sigma); m_m(M)} & \xrightarrow{X} & \dfrac{x = u \wedge m_a(A)}{m_s(A, \Sigma); m_m(M)} \end{array}$$

It is clear from inspection that X is a Skip rule. After the Skip, we know that less(u, x).

6.3.2.2 *Win.*

$$\begin{array}{ccc} \dfrac{x \sim u \wedge A}{\mathrm{unbound}(x), \Sigma; M} & \xrightarrow{\mathit{WIN}} & \dfrac{\bigwedge_{s \in S} (x \Leftarrow u)_s \wedge A}{\mathrm{bound}(x), \Sigma; M} \\ m \downarrow & & \downarrow m \\ \dfrac{m_a(A)}{m_s(A, \Sigma); m_m(M)} & \xrightarrow{X} & \dfrac{m_a(A)}{(x \leftarrow u), m_s(A, \Sigma); m_m(M)} \end{array}$$

It is not immediately clear that transition X maps to a rule. We will show that X maps to an R-Bind rule. First, we show that $x = u$ is in $m_a(A)$. From $x \sim u$ we know that an Initiate has been done, while unbound(x) means no Win has yet been done; together this means that A contains $(x = u)_s$, which maps to $x = u$. Second, we have less(u, x) because it is a condition of Initiate, and because its truth value never changes. Third, $x \notin \mathrm{lhs}(m_s(A, \Sigma))$, since no Win has been done, and the only way to get a centralized binding for x is through a Win. Taken together, these three statements imply that the centralized transition is an R-Bind reduction.

6.3.2.3 *Lose and Arrive.* These rules trivially map to a Skip rule.

This proves the theorem. □

6.4 Liveness

It remains to show that the DU algorithm always terminates in a configuration that maps to a terminal configuration of the RCU algorithm. The main step is to show that the DU algorithm always "makes progress," in the sense of this section. As a corollary, it follows that the DU algorithm is deadlock-free. We first prove a small lemma about the nonprogressing rules.

Lemma (Finiteness of Nonprogressing DU Execution). *Given any valid configuration d of the DU algorithm, then the number of consecutive nonprogressing rules that can be reduced starting from d is finite. All possible resulting configurations d' satisfy $m(d') = m(d)$.*

Proof. The proof is by induction on the length of the execution e of which d is the last configuration. We assume that the lemma holds for all configurations of e before d. We show that it holds for d by enumerating all possible executions of nonprogressing rules. Consider all rules that manipulate actions based on the same variable-term pair (say, x and u). Denote a configuration in which a nonprogressing rule is applicable by the name of that rule. Denote by X a configuration in which

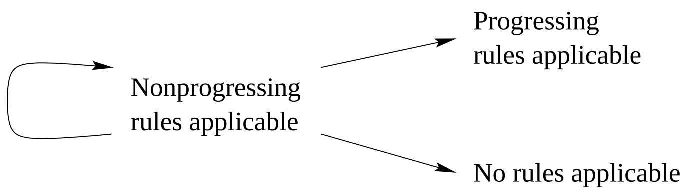

Fig. 17. Nonprogressing transitions in the DU algorithm.

no rules or only progressing rules are applicable. By inspecting the rules, we deduce the following graph of causal relationships:

$$\begin{tikzcd}[row sep=small] & & & \text{Lose} \arrow[r] & X \\ \text{Arrive} \arrow[r] & X & \text{Initiate} \arrow[ur] \arrow[dr] & & \\ & & & X & \end{tikzcd}$$

That is, applying Initiate possibly leads to a configuration in which Lose is applicable, and so forth. A graph identical to this one exists for all variable-term pairs. In all resulting sequences there are no cycles. Therefore a configuration in which some nonprogressing rules are applicable will eventually lead to one in which no nonprogressing rules are applicable. □

Theorem (DU Liveness). *If e is any execution of the DU algorithm such that $m(e)$ is nonterminal in the RCU algorithm, then continuing e will always eventually reduce a progressing rule.*

Proof. Assume a distributed execution e with last configuration d_i such that $c = m(d_i)$ is nonterminal. We must show that $\exists j > i : d_i \rightarrow \cdots \rightarrow d_{j-1} \rightarrow d_j$ where $d_{j-1} \rightarrow d_j$ is an application of a progressing rule. Any execution starting from d_i and doing nonprogressing rules as long as possible must initially follow the state diagram of Figure 17. Applying the lemma, we can assume that no nonprogressing rules are applicable in d_{j-1}. It remains to show that a progressing rule is always applicable there. We do a case analysis over the RCU rules. Let the RCU configuration be c, so that $m(d_{j-1}) = c$. For each rule, we apply the inverse of mapping m, and we attempt to infer whether a progressing rule is applicable.

6.4.1 *Interchange.* Assume that the Interchange rule is applicable in c. Therefore less(u, x) holds, and c contains $u = x$. For some site s, d_{j-1} contains $(u = x)_s$. Therefore the Interchange rule is applicable in d_{j-1}.

6.4.2 *R-Memo and R-Dereference.* Except for the memo table, the conditions for these two rules are identical. For both R-Memo and R-Dereference, c contains $x = u$ and $x \leftarrow v$ and we know less(u, x), Therefore for some site s, d_{j-1} contains $(x = u)_s$. For this site, d_{j-1} contains one of $(x \leftarrow v)_s$ or $(x \Leftarrow v)_s$. The case $(x \Leftarrow v)_s$ is impossible by the lemma, since in that case one of Arrive or Initiate is applicable depending on whether or not $(x \leftarrow \perp)_s$ is in the store. If *MEM*$(x = u, M_s)$ then a Memo is applicable. Otherwise, a Dereference is applicable.

6.4.3 *R-Bind.* Both less(u, x) and $x \notin$ lhs(σ) hold. For some site s, d_{j-1} contains $(x = u)_s$. This site must also contain $(x \leftarrow \perp)_s$, since otherwise an Initiate is applicable. Since $x \notin \mathrm{lhs}(\sigma)$, we know unbound($x$), so the $x \sim u$ of the INITIATE still exists, and a WIN rule is applicable.

6.4.4 *Identify, Conflict, and Decompose.* These are straightforward.

This proves the theorem. □

6.5 Total Correctness

THEOREM (DU TOTAL CORRECTNESS). *Given any finite multiset of equations, then placing them on arbitrary sites and executing the DU algorithm will terminate and result in a configuration that maps to a configuration equivalent to that of a terminating CU execution.*

PROOF. From DU Safety, any results obtained are correct results for the RCU algorithm. From DU Liveness and the Finiteness Lemma, the DU algorithm will terminate and reach this correct result. From RCU Total Correctness, the result is equivalent to the result of a terminating CU execution. □

7. ON-LINE FINITE ENTAILMENT

In a real system, it is almost never the case that unification is initiated with a fixed set of equations and runs to termination without any interaction with the external environment. Rather, the algorithm will be running indefinitely, and from time to time an equation will be added to the current action. This is the *on-line* case. The interesting property is not termination, but whether the equation will be entailed by the store in a finite number of reductions (finite entailment). In this section, we extend the CU and DU algorithms to the on-line case, and we show that the extended algorithms satisfy the finite-entailment property. We use the standard weak fairness assumption that all rule instances applicable for an indefinite period of time will eventually be reduced. We show that this is not enough to guarantee that the equation will be entailed, but that we need an additional property, the finite-size property, to bound the amount of work needed to incorporate the equation in the store.

7.1 On-Line CU and DU Algorithms

We extend the CU algorithm (from now on called the *off-line* CU algorithm) with a new rule:

$$\text{INTRODUCE} \qquad \frac{\text{true}}{\sigma;\ \mu} \Bigg\| \frac{u = v}{\sigma;\ \mu}$$

This rule is always applicable, and it adds a new equation to the action when it reduces. The extended algorithm, also called the *on-line* CU algorithm, therefore does not terminate. We extend the DU algorithm in a similar way by an INTRODUCE rule that introduces $(u = v)_s$ for an arbitrary site s.

7.2 Finite-Size Property

Any store σ can be mapped to a graph with two kinds of nodes, variables and records. The graph is defined in a straightforward way from the store's bindings (see Figure 18):

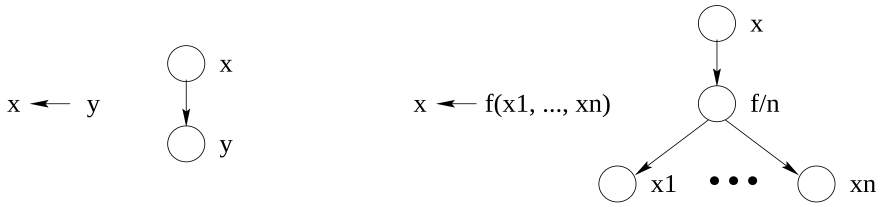

Fig. 18. Mapping the store to its graph.

—A binding $x \leftarrow y$ maps to variable nodes x and y, with a directed edge from x to y.

—A binding $x \leftarrow f(x_1, ..., x_n)$ maps to a set of variable nodes x, x_1, ..., x_n and a record node f/n, with directed edges from x to f/n and from f/n to every x_1, ..., x_n.

Given a variable node x, we define graph(x, σ) as the subgraph of σ's graph whose nodes and edges are reachable from x. We also define size(x, σ) as the number of edges in this subgraph. This quantifies the size of the data structure attached to x.

Given a valid configuration with store σ, it is clear that size(x, σ) is finite. However, the size may be unbounded when considering all configurations of a given execution. This leads us to define the following property. We say a variable x has the *finite-size* property for the execution e if

$$\exists n \geq 0 : \forall (\_; \sigma_k; \_) \in e : \text{size}(x, \sigma_k) \leq n.$$

That is, there is a finite upper bound on the size of x that holds for the whole execution. We say that an equation $u = v$ has the finite-size property if all its variables do. The finite-size property is used to avoid infinite executions caused by race conditions in two cases:

(1) **Dereference chains that increase in length indefinitely**: For example, consider the equation $x_0 = y$, which is accompanied by the infinite sequence of equations $x_i = x_{i+1}$, for i = 0, 1, 2, etc. These equations are added by the Introduce rule at the appropriate times. We assume that variable order is chosen so that lower-indexed variables are bound to higher-indexed variables, and that y is never bound. For each i, if $x_i = x_{i+1}$ is introduced and bound before $x_i = y$ is dereferenced, then the store will never entail $x_0 = y$.

(2) **Nested terms that increase in depth indefinitely**: For example, consider the equation $x_0 = y_0$, which is accompanied by the equations $x_i = f(x_{i+1})$ and $y_i = f(y_{i+1})$, for i = 0, 1, 2, etc. For each i starting with 0, if $x_i = f(x_{i+1})$ and $y_i = f(y_{i+1})$ are both introduced and bound before $x_i = y_i$ is decomposed, then the store will never entail $x_0 = y_0$.

It is remarkable that these two infinite executions are possible even with the weak fairness assumption. One way to avoid infinite executions would be to give the Introduce rule lower priority than the others, i.e., as long as another rule is applicable, do not reduce an Introduce rule. But this does not model the real world, in which equations can arrive at any time. The finite-size property does not have this deficiency. It does not restrict in any way *when* new equations are introduced. Rather, it forbids just those executions that would cause a problem.

The finite-size property can be enforced easily for dereference chains by requiring that all new variables have higher order than all existing variables. Then the total length of all dereference chains that need traversing is bounded by the number of variables in the system when the equation is introduced.

In the case of nested structures, the finite-size property can be enforced by not unifying terms whose nesting depth is potentially unbounded. This seems to be a reasonable condition because when a potentially infinite unification is necessary in practice, then it is sufficient that it always makes progress, not that it completes (e.g., see the streams of Section 2.4.2). The weak fairness assumption is enough by itself to guarantee progress of infinite unifications and eventual termination of finite unifications. The finite-size property ensures that a unification that is intended by the programmer to be finite will actually be finite during the execution. These two conditions suffice for all practical programs we know of.

7.3 Finite Entailment

Under what conditions will the store entail a given equation after a finite number of reductions? In this section we show that two conditions are needed in addition to weak fairness. First, there must be no detected inconsistencies (false actions) within the context of the given memo table. Second, the amount of work needed to incorporate the equation into the store must be finite (finite-size property).

An inconsistency is detected at most once per memo table. This is true for both the centralized and distributed algorithms as well as the Mozart implementation. In the CU algorithm, there is only one memo table, so an inconsistency is detected at most once. In the DU algorithm, there is a memo table per site, so an inconsistency can be detected once per site.

THEOREM (FINITE ENTAILMENT OF ON-LINE CU). *Given (1) weak fairness, (2) any valid configuration c of the on-line CU algorithm that contains the equation $u = v$, and (3) any execution e that contains c and satisfies the finite-size property for $u = v$, then e will eventually contain either a false action or a store that entails $u = v$.*

PROOF. We outline the proof. We are given that size$(u = v, \sigma_k)$ has a finite upper bound in e. Therefore graph$(u = v, \sigma_k)$ has a finite limit graph. Let V denote the set of variables in this graph. Denote the store corresponding to the limit graph as σ_V. Since V has a finite limit, the set $\mu_V = \{x = y \in \mu_k | x, y \in V\}$, i.e., of equalities in μ_k whose variables are in V, also has a finite limit. When this limit is reached, then consider the equations α_V, part of α_k, whose variables are all in V. Consider an execution starting with $(\alpha_V;\mu_V;\sigma_V)$, without the INTRODUCE rule, and that reduces rules in the same order as e does. This is a continuation of an off-line CU execution. If no false actions occur, then the Entailment Property (see Section 4.2) implies that eventually we end up with a store that entails $u = v$. □

We now extend this result to the distributed case. First we extend the DU algorithm to an on-line DU algorithm by an INTRODUCE rule that introduces an equation on any site. It is easy to see that safety continues to hold. We now show liveness and finite entailment for the on-line DU algorithm.

THEOREM (LIVENESS OF ON-LINE DU). *Given (1) weak fairness and (2) any execution e of the on-line DU algorithm such that $m(e)$ is nonterminal in the RCU algorithm, then continuing e will always eventually reduce a progressing rule.*

PROOF. The proof follows by minor modification of the proof of DU Liveness, using weak fairness to compensate for the INTRODUCE rule. □

THEOREM (FINITE ENTAILMENT OF ON-LINE DU). *Given (1) weak fairness, (2) any valid configuration d of the on-line DU algorithm that contains the equation $(u = v)_s$, and (3) any execution e that contains d and such that $m(e)$ satisfies the finite-size property for $u = v$, then e will eventually contain either a $false_s$ action or a store on site s that entails $u = v$.*

PROOF. We outline the proof. The execution on site s has a local memo table M_s for site s. We consider this execution to be a centralized execution with memo table $\mu = M_s$. By the previous theorem, the result holds for the centralized execution. Therefore the result holds also for the distributed execution on site s. □

8. THE MOZART IMPLEMENTATION

The Mozart system contains a refined version of the on-line DU algorithm, called "Mozart algorithm" in what follows. Section 8.1 summarizes how the implementation differs with respect to the on-line DU algorithm. Section 8.2 introduces the distribution graph, which is the framework in which the Mozart algorithm is defined. Then Section 8.3 defines the properties of the network and the notation used to define the distributed algorithm. After these preliminaries, the algorithm itself is defined. Section 8.4 defines the local algorithm, and Section 8.5 defines the distributed algorithm.

8.1 Differences with On-Line DU

The Mozart algorithm refines the on-line DU algorithm by making concrete decisions regarding several aspects that were left open. Furthermore, the Mozart algorithm does several optimizations to improve performance and has several extensions including a model for failure detection and handling. This section summarizes these refinements, optimizations, and extensions.

8.1.1 *Refinements.*

8.1.1.1 *Separation into Local and Distributed Algorithms.* The Mozart algorithm consists of two parts: a purely local algorithm (corresponding to the DU nonbind rules; see Section 8.4) and a distributed algorithm (corresponding to the DU bind rules; see Section 8.5). A thread wishing to tell an equation invokes the local algorithm. To bind a distributed variable to another one or to a record, the local algorithm invokes the distributed algorithm. The thread blocks, waiting for a reply. When the variable binding is known locally, then the thread continues.

8.1.1.2 *The Owner Site.* Each distributed variable is managed from a special site, the owner site, which is where the variable was originally created. This site contains the variable's unbound/bound flag and other information, e.g., the register list (see below).

8.1.1.3 *Variable Ordering.* The Mozart algorithm implements the order relation less(u, v) as follows. Records are less than distributed variables, which are less than local variables. Distributed variables are totally ordered; local variables are totally ordered per site; and records are unordered. Local variables are ordered according to a per-site index i that is incremented for each new variable.[5] Distributed variables are ordered according to a pair (s, i) where s is the site number on which the variable was initially created and i is the index of the variable on that site. From this ordering relation it follows that if the number of sites is finite and data structures of unbounded depth are not created, then the Mozart algorithm satisfies the finite-size property (see Section 7.2).

8.1.2 *Optimizations.*

8.1.2.1 *Globalization.* The Mozart algorithm distinguishes between local and distributed variables (see Section 8.5.1).

8.1.2.2 *Variable Registration.* A variable binding is not sent to all sites, but only to registered sites (see Section 8.5.2).

8.1.2.3 *Grouping Nested Data Structures.* Binding a nested data structure to a distributed variable is done by the Mozart algorithm as a single operation (see Section 8.5.3).

8.1.2.4 *Winner Optimization.* When a variable is bound to a term, then the term does not have to be sent back to the site that initiated the binding (see Section 8.5.4).

8.1.2.5 *Asynchronous Streams.* To allow streams to be created asynchronously, variables are given a set of registered sites as soon as they are globalized (see Section 8.5.5).

8.1.3 *Extensions.*

8.1.3.1 *Lazy and Eager Variables.* The laziness property affects the moment when the variable is registered. Eager proxies are registered immediately. Lazy proxies delay registration until a binding attempt is made (see Section 8.5.6).

8.1.3.2 *Read-Only Logic Variables.* Standard logic variables have two operations, reading the value and binding. For various reasons, it is often useful to prohibit binding, for example, when building abstractions or when passing the variable to a less-trusted site [Mehl et al. 1998; Mehl 1999].[6]

8.1.3.3 *Garbage Collection.* Distributed garbage collection is based on a credit mechanism that collects all garbage except cross-site cycles between stateful entities (see Section 8.2.1).

8.1.3.4 *The Failure Model.* The Mozart algorithm is conservatively extended with a model for failure detection and handling that reflects network and site failures to the language level (see Section 8.2.2).

---

[5]For local variables, the index is simply the variable's address.

[6]Read-only logic variables are confusingly called "futures" in these two references.

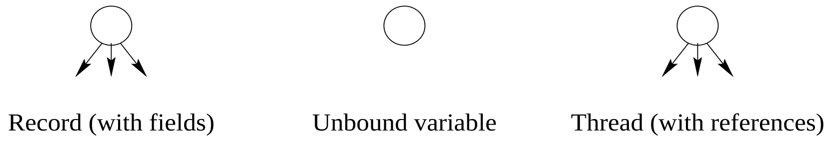

Fig. 19. The three node types of the language graph.

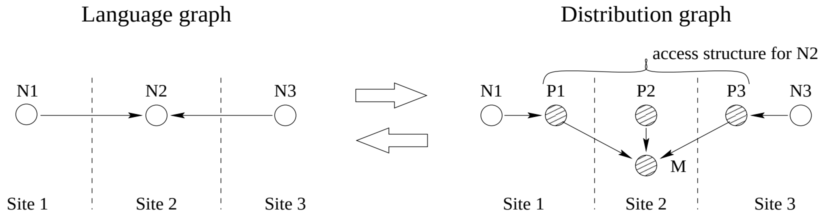

Fig. 20. An access structure in the distribution graph.

8.2 The Distribution Graph

We model distributed executions in a simple but precise manner using the concept of a *distribution graph*. We obtain the distribution graph in two steps from an arbitrary execution state of the system. The first step is independent of distribution. We model the execution state by a directed graph, called *language graph*, in which a record, an unbound variable, and a thread each correspond to one node (see Figure 19). A node can be considered as an active entity with internal state that can asynchronously send messages to other nodes and that can synchronously receive messages from other nodes. The edges in the language graph denote the node's references: a record and a thread refer to other nodes; an unbound variable has no references.

In the second step, we distribute the execution over a set of sites. Assume a finite set of sites, and annotate each node by its site (see Figure 20). If a variable node, e.g., `N2`, is referenced by at least one node on another site, then map it to a *set* of nodes, e.g., {`P1`,`P2`,`P3`,`M`}. This set is called the *access structure* of the original node. An access structure consists of one *proxy node* `Pi` for each site that referenced the original node and one *owner node* `M` for the whole structure. The resulting graph, containing both local nodes and access structures where necessary, is called the *distribution graph*. The execution of the distributed algorithm is defined in terms of this graph. Execution consists of atomic graph transformations that are initiated by the nodes. A logic variable implemented as an access structure is called a *distributed* variable (as opposed to a *local* variable). A variable referenced on more than one site is certain to be represented by an access structure.

Each access structure is given a global name n that is unique systemwide. In Distributed Oz, n is the pair (s, i) that is also used to order the distributed variables. The global name n encodes (among other things) the owner site s. Furthermore, a proxy node is uniquely identified by the pair (n, s'), which contains the proxy site s'. On each site, n indexes into a table that refers to the proxy. This allows to enforce the invariant that each site has at most one proxy.

Messages are sent between nodes in access structures. In terms of sites, a message is sent from the source node's site to the destination node's site. The message may contain a subgraph of the distribution graph. Just before the message leaves the source site, a new access structure is created for each local variable in the subgraph. When in transit, the message refers only to proxy nodes, not to local variables. When the message arrives, the subgraph becomes known at the destination site. Each proxy node is looked up in the site table. If it is not present, then a new proxy node is created and entered in the table. This extends an existing access structure with one new proxy. The process of creating an access structure is called *globalization* (see Section 8.5.1).

The behavior of a distributed variable is defined as a protocol between the nodes of its access structure. In general, nodes other than variable nodes can also have access structures, and therefore obey a protocol. The Distributed Oz implementation uses four nontrivial protocols. Three are designed for specific language entities, namely variables, object records, and state pointers. Variables use a *variable binding* protocol, which is part of the distributed unification algorithm and is presented in this article. Object records use a *lazy replication* protocol. State pointers use a *mobile state* protocol. See Alouini and Van Roy [1999], Van Roy et al. [1997; 1999], and Haridi et al. [1998] for more information on these protocols. The fourth protocol is a distributed garbage collection algorithm using a credit mechanism. Garbage collection is part of the management of access structures, and it therefore underlies the other three protocols.

8.2.1 *Garbage Collection.* Distributed garbage collection is based on a credit mechanism, which is a variant of weighted reference counting [Plainfossé and Shapiro 1995]. The credit mechanism interfaces with the local garbage collector of each site. All distributed garbage is removed except for cross-site cycles between stateful entities on different owner sites. The mechanism has four useful properties. First, creating a new proxy requires essentially zero network messages in addition to the messages sent by the application. Second, each proxy site does not need to know any other site except the owner site. Third, the owner site does not need to know any proxy site. Fourth, sites that no longer locally reference a proxy will, after a local garbage collection, no longer affect the access structure in any way.

The global name of an access structure is associated with a pool of *credits*. The owner site lends credits to sites and messages that know the global name. All proxy sites and messages must hold at least one credit. The owner site has an integer corresponding to the total number of credits lent. The proxy site keeps a count of how many credits it has borrowed from the owner site. If local garbage collection removes the proxy node, then all its credits are returned to the owner site. Both the global name and the owner node can be reclaimed after all credits have returned to the owner site. When that happens, the language entity becomes local again (see Section 8.5.1).

8.2.2 *The Failure Model.* The failure model is designed according to the principle that an Oz program should be able to make all decisions regarding failure behavior [Van Roy et al. 1999; Haridi et al. 1998; Brand et al. 1999; Van Roy 1999]. That is, the implementation does not make any irrevocable decisions by default. Full treatment of the model is beyond the scope of this article. We briefly summarize the main ideas. The failure model considers failures at the level of individual

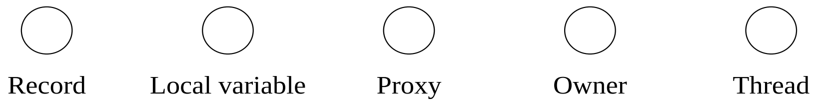

Fig. 21. The five node types of the distribution graph.

language entities, e.g., logic variables. The model covers site failures (fail-silent with failure detection) and network failures (temporary or permanent; characterized by lack of any definite information). The model has two effects at the language level:

—It extends each operation on an entity to have three possible results. If there is no failure then the operation succeeds. If there is a failure, then the operation either waits indefinitely until the problem goes away or aborts and is replaced by a user-defined procedure call. The call can retry the operation. There are no default time outs; it is up to the program to decide whether to continue to wait or not. For example, an entity can be configured so that all operations wait indefinitely on network failures (in the hope that they are temporary) and raise an exception on a permanent site failure.

—It allows to eagerly detect a problem with an entity, i.e., without having to do an operation on the entity. When a user-specified failure condition is detected then a user-defined procedure is called in its own thread. The failure condition does not necessarily keep the entity from working, i.e., it can just give information. For example, a remote proxy site failure will often have no effect at all on the binding of a logic variable, but it may nonetheless be important to be notified of this failure.

This failure model is fully implemented as part of the Mozart system. We are currently developing more powerful abstractions in Oz on top of this model.

8.3 Basic Concepts and Notation

8.3.1 *Network.* Consider a single owner node M, a set of p proxy nodes P_i with $1 \leq i \leq p$, and a set of m thread nodes T_i with $1 \leq i \leq m$. Let these nodes be linked together by a network N that is a multiset containing messages of the form $d : m$ where d identifies a destination (thread, proxy, or owner node) and where m is a message. Then the distributed part of the Mozart algorithm is defined using reduction rules of the form

$$\frac{\text{Condition}}{\text{Action}}.$$

Each rule is defined in the context of a single node. Execution follows an interleaving model. At each reduction step, a rule with valid condition is selected. Its associated actions are reduced atomically. A rule condition consists of boolean conditions on the node state and one optional receive condition **Receive**(d,m). The condition **Receive**(d,m) means that $d : m$ has arrived at d. Executing a rule with a receive condition removes $d : m$ from the network and performs the action part of the rule. A rule action consists of a sequence of operations on the node state with optional sends. The action **Send**(d,m) asynchronously sends message m to node d, i.e., it adds the message $d : m$ to the network.

Table IV. Node State

Node	Attribute	Type
Any node	id	NodeId
	type	{RECORD,LOCVAR,PROXY,MANAGER,THREAD}
Record	label	Atom
(N.type=RECORD)	arity	Integer
	args	**array**[1..arity] **of** Node
Local variable	state	{UNBOUND, BOUND(Node)}
(N.type=LOCVAR)	eager	{FALSE, TRUE}
Proxy	state	{UNBOUND, INITIATED, BOUND(Node)}
(N.type=PROXY)	eager	{FALSE, TRUE}
	reg	{FALSE, TRUE}
	owner	NodeId
Owner	state	{UNBOUND, BOUND(Node)}
(N.type=MANAGER)	reglist	**set of** NodeId
Thread		
(N.type=THREAD)		

We assume that the network and the nodes are *fair* in the following sense. The network is asynchronous, and messages to a given node take arbitrary finite time and may arrive in arbitrary order. All rule instances that remain applicable for an indefinite period of time will eventually be reduced.

8.3.2 *Node State.* There are five node types in the distribution graph (see Figure 21). Table IV defines the state of each node type by listing the attributes of each node. All nodes have attributes "id" and "type," which have constant values. The types NodeId, Atom, and Integer are atomic types implemented in a straightforward way. In the real system, threads, proxies, and owners have more attributes, e.g., threads have an execution stack, while proxies and owners maintain information for distributed garbage collection.

8.3.3 *Utility Operations.* Most of the utility operations are defined in Figure 22. In addition, there is also a boolean function less(N_1,N_2) that defines the variable ordering. Finally, there is a memo table data type with three operations: a procedure clearmemo(M), a procedure add(N_1,N_2,M), and a boolean function mem(N_1,N_2,M). The latter two are exactly the *ADD* and *MEM* operations defined in Section 3.2.

8.4 The Local Algorithm

Figure 23 defines the local algorithm, which executes in each thread that does a unification. The definition follows closely the nonbind rules of Section 5.4.1, where a rule corresponds to a guard and its body. The two main differences are that the local algorithm maintains a memo table that is shared among the rules and that a sequential order is imposed on rule reductions. The **if** is a guarded command that suspends until at least one of its guards is true.

In the implementation, executions of the local and distributed algorithms are not interleaved. Rather, the local algorithm is executed atomically until it exits (either normally or through an exception) or until it blocks after sending a binding request.

The execution of unify completes in two ways: when it terminates normally or when an inconsistency is detected, in which case a failure exception is raised.

$$
\begin{array}{lcl}
\mathrm{eq(N_1,N_2)} & = & \mathrm{N_1.id=N_2.id} \\
\mathrm{locvar(N)} & = & \mathrm{N.type=LOCVAR} \\
\mathrm{disvar(N)} & = & \mathrm{N.type=PROXY} \\
\mathrm{nonvar(N)} & = & \mathrm{N.type=RECORD} \\
\mathrm{var(N)} & = & \mathrm{locvar(N)} \vee \mathrm{disvar(N)} \\
\mathrm{bound(N)} & = & \textbf{if}\ \mathrm{var(N)} \rightarrow \mathrm{N.state=BOUND(\_)}\ \textbf{fi} \\
\mathrm{initiated(N)} & = & \textbf{if}\ \mathrm{disvar(N)} \rightarrow \mathrm{N.state=INITIATED}\ \textbf{fi} \\
\mathrm{deref1(N)} & = & \textbf{if}\ \mathrm{var(N)} \wedge \mathrm{bound(N)} \rightarrow \\
 & & \quad \mathrm{N_1}\ \textbf{where}\ \mathrm{N.state=BOUND(N_1)}\ \textbf{fi} \\
\mathrm{compatible(N_1,N_2)} & = & \textbf{if}\ \mathrm{nonvar(N_1)} \wedge \mathrm{nonvar(N_2)} \rightarrow \\
 & & \quad \mathrm{N_1.label=N_2.label} \wedge \mathrm{N_1.arity=N_2.arity}\ \textbf{fi} \\
\mathrm{proxyids(N)} & = & \textbf{if}\ \mathrm{locvar(N)} \rightarrow \{\} \\
 & & [\,]\ \mathrm{disvar(N)} \rightarrow \\
 & & \quad \textbf{if}\ \mathrm{bound(N)} \rightarrow \mathrm{proxyids(deref1(N))} \\
 & & \quad [\,]\ \textbf{not}\ \mathrm{bound(N)} \rightarrow \{\mathrm{N.id}\}\ \textbf{fi} \\
 & & [\,]\ \mathrm{nonvar(N)} \rightarrow \bigcup_{1 \le i \le \mathrm{N.arity}} \mathrm{proxyids(N.args}[i])\ \textbf{fi}
\end{array}
$$

Fig. 22. Utility operations.

During the execution, the thread will block when waiting for a binding request to complete. This works as follows. The first thread that tries to bind a variable will send a binding request (INITIATE rule). Other threads that try to bind the same variable on the same site will not send any message. All threads then block at the **if**: no guard is true because the node satisfies disvar(N_1) $\wedge$ **not** bound(N_1) $\wedge$ initiated(N_1). As soon as the binding arrives, bound(N_1) is true, and the **if** becomes reducible. All threads can then continue.

The local algorithm is optimized to bind local variables locally. With a suitable data representation, the algorithm is implementable very efficiently [Gudeman 1993; Van Roy 1994]. The implementation does binding in place and dereferencing inline. Common cases of unification such as parameter passing and local variable initialization do not need binding nor dereferencing, but reduce to single register moves or stores.

The local memo table is implemented by means of *forwarding pointers* between variables [Haridi and Sahlin 1984]. That is, when the equation $x = y$ is encountered for the first time, a forwarding pointer is installed from x to y. This allows a very fast check of memo table membership. Namely, if $x = y$ or $y = x$ is encountered later on, then dereferencing will reduce the equation to $y = y$, which does no further work. The forwarding pointers are installed in the context of a single atomic unification operation. They are removed when the local algorithm exits or blocks.

Other operations can be performed on a site while a unification is blocked. For correctness, the forwarding pointers must be removed whenever execution leaves the local algorithm. This is modeled in Figure 23 by creating a new memo table when unify is called and by clearing the memo table after an INITIATE rule. This means that the memo table starts from empty at each atomic execution of the local algorithm. The Mozart algorithm therefore potentially does more redundant work than the on-line DU algorithm, because the DU algorithm never clears the local memo tables.

```
procedure unify(N1,N2)
  define memotable M
  define procedure inner_unify(N1,N2)
    if /**** INTERCHANGE ****/
      var(N2), less(N1,N2) → inner_unify(N2,N1)
    [] /**** IDENTIFY ****/
      var(N1), eq(N1,N2) → skip
    [] /**** MEMO ****/
      var(N1), less(N2,N1), bound(N1), mem(N1,N2,M) → skip
    [] /**** DEREFERENCE ****/
      var(N1), less(N2,N1), bound(N1), not mem(N1,N2,M) →
      add(N1,N2,M)
      inner_unify(deref1(N1),N2)
    [] /**** BIND ****/
      locvar(N1), less(N2,N1), not bound(N1) →
      N1.state ← BOUND(N2)
    [] /**** INITIATE ****/
      disvar(N1), less(N2,N1), not bound(N1), not initiated(N1) →
      N1.state ← INITIATED
      Send(N1.owner,binding_request(N2))
      clearmemo(M)
      inner_unify(N1,N2)
    [] /**** DECOMPOSE ****/
      nonvar(N1), nonvar(N2), compatible(N1,N2) →
      for i:=1 to N1.arity do inner_unify(N1.args[i],N2.args[i])
    [] /**** CONFLICT ****/
      nonvar(N1), nonvar(N2), not compatible(N1,N2) →
      raise failure_exception
    fi
  end
in
  clearmemo(M)
  inner_unify(N1,N2)
end
```

Fig. 23. Distributed unification part 1: Local algorithm.

8.5 The Distributed Algorithm

Figure 24 defines the distributed algorithm, which extends the DU bind rules of Section 5.4.2 with variable registration. The algorithm also does four important optimizations, namely globalization, grouping nested data structures, the winner optimization, and asynchronous streams. We do not define all these optimizations formally, but explain briefly how to extend the algorithm to include them. We also explain how to extend the algorithm for lazy and eager variables.

8.5.1 *Globalization.* Newly created variables are always local. When a message is sent that contains a local variable, then a new distributed variable is created, and the local variable is bound to it. This is called *globalizing* the local variable. An access structure is created when a local variable is globalized. When the message arrives then a new proxy will be created for the distributed variable if none exists on the arrival site. Therefore globalization is part of the **Send** and **Receive** operations [Alouini and Van Roy 1999]. The inverse operation, *localization*, consists of

/**** WIN ****/

$$\frac{\textbf{Receive}(\text{M.id,binding\_request(N)}) \ \wedge \ \text{M.state=UNBOUND}}{\begin{array}{l}\forall i \in \text{M.reglist: } \textbf{Send}(i, \text{ binding\_in\_transit(N)}) \\ \text{M.state} \leftarrow \text{BOUND(N)}\end{array}}$$

/**** LOSE ****/

$$\frac{\textbf{Receive}(\text{M.id,binding\_request(\_)}) \ \wedge \ \text{M.state=BOUND(\_)}}{\textbf{skip}}$$

/**** ARRIVE ****/

$$\frac{\textbf{Receive}(\text{P.id,binding\_in\_transit(N)}) \ \wedge \ (\text{P.state=UNBOUND} \vee \text{P.state=INITIATED})}{\begin{array}{l}\forall i \in \text{proxyids(N): } \textbf{Send}(i, \text{ reg}) \\ \text{P.state} \leftarrow \text{BOUND(N)}\end{array}}$$

/**** Variable registration ****/

$$\frac{\textbf{Receive}(\text{P.id,reg}) \ \wedge \ \text{P.reg=FALSE}}{\begin{array}{l}\text{P.reg} \leftarrow \text{TRUE} \\ \textbf{Send}(\text{P.owner,register(P.id)})\end{array}}$$

$$\frac{\textbf{Receive}(\text{P.id,reg}) \ \wedge \ \text{P.reg=TRUE}}{\textbf{skip}}$$

$$\frac{\textbf{Receive}(\text{M.id,register(PId)}) \ \wedge \ \text{M.state=UNBOUND}}{\text{M.reglist} \leftarrow \text{M.reglist} \cup \{\text{PId}\}}$$

$$\frac{\textbf{Receive}(\text{M.id,register(PId)}) \ \wedge \ \text{M.state=BOUND(N)}}{\textbf{Send}(\text{PId,binding\_in\_transit(N)})}$$

Fig. 24. Distributed unification part 2: Distributed algorithm.

removing the access structure when the variable is only referenced on one site (see Section 8.2.1). The distributed variable becomes a local variable again.

All local variables and proxies have a boolean attribute "eager" that determines whether the node is eager or lazy. The attribute affects only the network operations of the distributed algorithm. Assume we have a local variable L with L.state=UNBOUND and L.eager=b. After globalizing, the original site contains three nodes, L, P, and M, with the following states:

L.state=BOUND(P)
P.state=UNBOUND, P.eager=b, P.reg=b, P.owner=M.id
M.state=UNBOUND, M.reglist=**if** b **then** {P.id} **else** {} **fi**

8.5.2 *Variable Registration.* In the DU algorithm, a binding arrives on a site if the variable exists in the site's store. In the Mozart algorithm, the variable's owner keeps track of the sites that reference the variable. A site that receives a distributed variable for the first time (i.e., when a term containing the variable first arrives on the site) has to register with the owner in order to receive the variable's binding. When a variable is bound to N, then a binding_in_transit(N) message is sent to all registered sites. When this message reaches a site then the ARRIVE rule reduces (see Figure 24). This sends reg messages to all proxies in the binding. The reg message causes all unregistered proxies to register with their owner. If the variable is already bound when the register message arrives then the binding is sent back immediately.

8.5.3 *Grouping Nested Data Structures.* The DU algorithm binds only a single record at a time, namely the top level of the tree. The Mozart algorithm binds a complete tree in a single operation. In this way, it avoids the creation of distributed variables for the intermediate nodes. For example, the unification $x_1 = f(g(a))$ is represented in the DU algorithm as three actions $x_1 = f(x_2) \wedge x_2 = g(x_3) \wedge x_3 = a$. In the DU algorithm, the arrival of $x_1 \Leftarrow f(x_2)$ enables the arrival of $x_2 \Leftarrow g(x_3)$, and similarly for x_3. In the Mozart algorithm, the binding $x_1 \Leftarrow f(g(a))$ arrives in one step, so the variables x_2 and x_3 are never created.

8.5.4 *Winner Optimization.* The winner is the proxy that sent a successful binding_request(N). This proxy does not need to get back N, since N already exists on the proxy's site. The proxy can be sent a simple acknowledgment that its binding request was successful. This avoids the redundant work done by the R-Bind rule (see Section 6.2).

The winner optimization requires the following protocol extensions: an extended proxy state INITIATED(N) where N is the binding, an extended message binding_request(N,PId) where PId identifies the winning proxy, and a new message binding_ack from the owner to the winning proxy. When the proxy receives binding_ack, then it retrieves N from the INITIATED(N) state.

8.5.5 *Asynchronous Streams.* A variable that is exported from its owner site can be *preregistered*. That is, the destination site is added to the owner's reglist without waiting for a registration message. This is correct if there is a FIFO connection to the destination site. Preregistering variables allows elements to be added to streams asynchronously. The example of Section 2.4.2 relies on this behavior.

Let us look closely to see what happens. Assume variable `X0` exists on sites 1 and 2. Binding `X0=m1|X1` on site 1 causes `m1|X1` to be sent to site 2. `X1` will be preregistered, i.e., M_{X1}.reglist $\leftarrow$ M_{X1}.reglist $\cup$ M_{X0}.reglist when the binding leaves site 1. If `X1` is later bound on site 1, then its binding will be sent immediately to site 2 without waiting for a registration request from site 2.

If preregistration is not done, then adding elements to a stream requires a round-trip message delay for each element. This is because remote proxies have to be registered before they can receive a binding. In our example, binding `X0=m1|X1` on site 1 causes `m1|X1` to be sent to site 2. When it arrives, an `X1` proxy is created on site 2 which promptly registers with site 1. Binding `X1=m2|X2` on site 1 will not send the binding to site 2 until the registration arrives on site 1. Therefore, each new element appears on site 2 only after a round trip.

8.5.6 *Lazy and Eager Variables.* Lazy and eager logic variables are defined informally in Section 2.2.2. In terms of the on-line DU algorithm, they differ only in the scheduling of the Arrive rule. To be precise, laziness is a property of a variable proxy, not of a variable. A proxy is *lazy* if the reduction of Arrive is delayed until after Initiate reduces on that site. If no such delay is enforced then the proxy is *eager*.

In terms of the Mozart algorithm, this is implemented by registering lazy and eager proxies at different times. Eager proxies are registered as soon as they appear on a site (see Arrive rule in Figure 24). Lazy proxies are only registered after the Initiate rule is reduced (see Figure 23), i.e., when a binding request is made.

When two proxies are bound together, the result must be eager if at least one of the two was eager. When a local variable is bound to a proxy, the proxy must become eager if the local variable was eager. Implementing this requires replacing the reg message by three messages: (1) in the ARRIVE rule, reg becomes reg_if_eager, (2) in the INITIATE rule, a new message reg_always is sent, and (3) in the BIND rule, a new message reg_and_make_eager is sent if the local variable is eager.

9. RELATED WORK

There are two main streams of related work. Some distributed implementations of concurrent logic languages do distributed unification (see Sections 9.1 and 9.3). Some imperative or dataflow language implementations have a kind of synchronizing variable (see Section 9.2). To our knowledge, the present article gives the first formal definition and correctness proof of a practical algorithm for distributed rational tree unification. The present article also clearly explains for the first time the advantages of using logic variables in a distributed system.

9.1 Concurrent Logic Languages

Many concurrent logic languages have been implemented in distributed settings. These systems do not use logic variables primarily to improve latency tolerance and network transparency. Rather, logic variables are integral parts of their execution models, and the distributed extensions must therefore implement them. We summarize the distributed unification algorithms used in Flat GHC, Parlog, Pandora, DRL, and KLIC.

9.1.1 *Flat GHC, Parlog, and D/C-Parlog.* Among early implementations doing some form of distributed unification are a Flat GHC (Guarded Horn Clauses) implementation on the Multi-PSI [Ichiyoshi et al. 1987], a Parlog implementation on a network of workstations [Foster 1988], and designs for distributed implementations of Parlog, Pandora, and D/C-Parlog [Leung 1993; Leung and Clark 1996]. Pandora extends Parlog with determinacy-driven execution (the Andorra model). D/C-Parlog extends Parlog with linear real-number constraints, namely equations, inequalities, and disequalities. All the above distributed unification algorithms are defined informally by explaining what happens with arguments of different types. No formal definitions nor correctness arguments are given.

The Parlog implementation contains an algorithm due to Foster [1988]. Variables exist on one site and have remote references, which is similar to the owner/proxy model of the Mozart algorithm. Variable-variable unification avoids binding cycles by ordering the variables, as is done in the DU algorithm. All remote references to variables are lazy, and dereference chains may cross sites. Preregistering is not done, so asynchronous streams are not possible.

Like early Prolog systems, Foster's algorithm does neither an occur-check nor memoization. When unifying two cyclic structures it may go into an infinite loop. The algorithm has proxy registration (called "ns_read") similar to the Mozart algorithm and a novel form of registration (called "read") that sends the binding only when the variable is bound to a nonvariable term. This is used to get the value for operations that need a nonvariable.

9.1.2 *DRL.* DRL [Diaz et al. 1997] (Distributed Real-time Logic language) is a concurrent logic language extended with features for distribution and soft real-time control. Distribution is introduced by allowing computations on different sites to communicate through shared logic variables. In DRL, the representative of a logic variable on a site is called a *logic channel*. A logic channel is always statically marked with a direction, which is either output or input. For a given logic variable, only one channel is marked output. Binding the output channel to a term causes the term to appear at all corresponding input channels. The binding blocks until the term contains only ground subterms and logic channels. It follows that variables can be transferred between sites only if they are statically declared as logic channels.

Logic channels can be connected together. This operation is called "unification" in DRL, but the shared logic variables are not actually unified together. To be precise, no variable elimination is done, but communication links are set up between variables. Connecting two output channels causes a future binding of one of them to be sent also to the other. Connecting an input channel to another channel suspends until the input channel receives a value. It follows that dependencies on intermediate sites are not removed.

9.1.3 *KLIC.* KLIC [Fujise et al. 1994] is an efficient portable implementation of the concurrent logic language KL1 (Kernel Language 1) for distributed and shared-memory machines. KLIC achieves these goals by compiling into C. On one processor running a series of representative benchmarks, the performance of KLIC approaches that of C and C++ implementations. The distributed implementation of KLIC does distributed unification [Rokusawa et al. 1996], including binding variables to variables. However, the algorithm has several curious properties: binding cycles can be created when binding variables to variables; inconsistencies are ignored; and a variable may be bound to different values on different sites. Apparently, the algorithm is only intended to be used in settings where there is no possibility of inconsistency.

9.2 Languages Not Based on Logic

We first compare logic variables with futures and I-structures (see Section 9.2.1), which have been used to improve expressiveness of parallel languages and performance of parallel systems. Then we briefly discuss traditional distributed architectures and how they could be extended to incorporate logic variables (see Section 9.2.2).

9.2.1 *Futures and I-Structures.* The purpose of futures and I-structures is to increase the potential parallelism of a program by removing inessential dependencies between calculations. They allow concurrency between a computation that calculates a value and one that uses the value. This concurrency can be exploited on a parallel machine. To our knowledge, they have not been used in distributed programming. We compare futures and I-structures with logic variables (see also Section 2.4.4).

The call `(future E)` (in Lisp syntax) does two things: it immediately returns a placeholder for the result of E, and it initiates a concurrent evaluation of E [Halstead 1985]. When the value of E is needed, the computation blocks until the value is available. We model this as follows in Oz (where E is a zero-argument function):

```
fun {Future E}
   thread {E} end
end
```

An important difference with a logic variable is that a future can only be bound by the concurrent computation that is created along with it. Therefore the above definition is not quite right; to precisely model futures a read-only logic variable should be used (see Section 8.1.3).

An I-structure (for *incomplete structure*) is a single-assignment array whose elements can be accessed before all the elements are computed [Arvind and Thomas 1980; Veen 1986; Iannucci 1990]. It permits concurrency between a computation that calculates the array elements and a computation that uses their values. When the value of an element is needed, then the computation blocks until it is available. An I-structure differs from an array of logic variables in that its elements can only be bound by the computation that calculates them.

9.2.2 *Two-Level Addressing.* Systems with support for distributed computing commonly provide two-level addressing. This provides the ability to use local and remote references interchangeably. References that arrive on a site are automatically converted to the local form if they refer to local entities. Typical examples include Java RMI [Sun Microsystems 1997], CORBA [Otte et al. 1996], and the Ericsson OTP (Open Telecom Platform) [Armstrong et al. 1996; Wikström 1994].

Two-level addressing can be extended to provide weak logic variables (see also Section 2.1.4). It suffices to add an "unknown" state to variables: (1) threads block when the variable is unknown, (2) when the value is available, all remote references to the variable leave the unknown state, and (3) no forwarding chains are created if a reference travels among many sites. There should be no overhead if the variable is on one site only. To provide full logic variables this is further extended with variable-variable unification. As the CC-Java implementation illustrates, dynamic typing is not necessary (see Section 2.5.1).

9.3 Sending a Bound Term

A basic operation in distributed unification is sending a bound term across the network. Lamma et al. [1997] investigate the costs of this operation and sketch an algorithm to send only that part of a term required by a consumer. Sending too little increases the message latency, since multiple requests will be done. Sending too much increases network load and memory consumption at the consumer. The proposed algorithm sends exactly that part of a term required by a consumer. For example, a list-appending procedure requires only the spine of the list, and not the terms in the list. The algorithm uses "consumption specifications," simple tree grammars extended with an additional terminal, `Remote`. These specifications can be given by static analysis or by programmer annotation.

10. CONCLUSIONS

This article has examined the use of logic variables in distributed computing. We have shown that if the logic variables are well implemented, then common distributed programming idioms can be written in a network-transparent manner, and they behave efficiently when distributed. We have defined the CU algorithm, a centralized algorithm for rational tree unification, and the DU algorithm, its conservative extension to a distributed setting. The DU algorithm has just two changes with respect to the CU algorithm. First, CU's single BIND rule is replaced by four rules that do coherent variable elimination. Second, CU's unique memo table is replaced by a local memo table per site.

We show that the DU algorithm has good network behavior for common distributed programming idioms. We prove that the DU algorithm is a correct implementation of unification, and we bound the amount of extra work it can do compared to the CU algorithm. We show that both lazy and eager logic variables are implemented by the DU algorithm. They differ only in the scheduling of a single reduction rule.

We extend both the CU and DU algorithms to the on-line case, in which new equations can be introduced indefinitely during execution. We show that if a weak fairness condition holds and if all variables in the equation satisfy the *finite-size* property, then any introduced equation will eventually be entailed by the store.

The Mozart system implements the Distributed Oz language and was publicly released in January 1999 [Mozart Consortium 1999]. Mozart contains an optimized version of the on-line DU algorithm. Distributed Oz, also known as Oz 3, conservatively extends Oz 2 to allow an efficient distributed network-transparent implementation [Haridi et al. 1998; Van Roy et al. 1997; Haridi et al. 1997]. Oz 2 has a robust centralized implementation that was officially released in February 1998 [DFKI Oz 1998]. Oz 3 keeps the same language semantics as Oz 2 and extends it with support for mobile computations, open distribution, component-based programming, and orthogonal failure detection and handling within the language. Oz 2 programs are portable to Oz 3 almost immediately.

ACKNOWLEDGEMENTS

Michael Mehl implemented the distributed unification algorithm in the Mozart system. Per Brand and Erik Klintskog have extended it to incorporate orthogonal failure detection and handling. Iliès Alouini and Mustapha Hadim gave many valuable comments on this article. We thank Andreas Podelski and all our colleagues in the Mozart Consortium (DFKI, SICS, UCL, and UdS). Finally, we thank the referees for comments that let us vastly improve the presentation.

REFERENCES

ALFORD, M. W., LAMPORT, L., AND MULLERY, G. P. 1985. Lecture Notes in Computer Science, vol. 190. Springer Verlag, Chapter 2. Basic Concepts, in Distributed Systems–Methods and Tools for Specification, An Advanced Course.

ALOUINI, I. AND VAN ROY, P. 1999. Le protocole réparti de Distributed Oz (in French). In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 99)*. 283–298.

ARMSTRONG, J., WILLIAMS, M., WIKSTRÖM, C., AND VIRDING, R. 1996. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, N.J.

ARVIND AND THOMAS, R. E. 1980. I-Structures: An efficient data type for functional languages. Tech. Rep. 210, MIT, Laboratory for Computer Science.

BAL, H. E., STEINER, J. G., AND TANENBAUM, A. S. 1989. Programming languages for distributed computing systems. *ACM Comput. Surv. 21,* 3 (Sept.), 261–322.

BRAND, P., VAN ROY, P., COLLET, R., AND KLINTSKOG, E. 1999. Formal definition and correctness of a reliable mobile-state protocol for constructing fault-tolerant applications. In preparation.

CARDELLI, L. 1995. A language with distributed scope. *ACM Trans. Comput. Syst. 8,* 1 (Jan.), 27–59.

CHOW, R. AND JOHNSON, T. 1997. *Distributed Operating Systems and Algorithms.* Addison-Wesley, San Francisco, Calif.

COLMERAUER, A. 1982. *Prolog and Infinite Trees.* Academic Press. In *Logic Programming*, Keith L. Clark and Sten-Åke Tarnlund, eds.

COURCELLE, B. 1983. Fundamental properties of infinite trees. *Theoretical Computer Science 25,* 95–169.

DFKI Oz 1998. DFKI Oz version 2.0. Available at `http://www.ps.uni-sb.de`.

DIAZ, M., RUBIO, B., AND TROYA, J. M. 1997. DRL: A distributed real-time logic language. *Comput. Lang. 23,* 2–4, 87–120.

DUCHIER, D., KORNSTAEDT, L., SCHULTE, C., AND SMOLKA, G. 1998. A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling. Tech. rep., Programming Systems Lab, DFKI and Universität des Saarlandes. DRAFT.

FOSTER, I. 1988. Parallel implementation of Parlog. In *International Conference on Parallel Processing.* IEEE Computer Society, 9–16.

FUJISE, T., CHIKAYAMA, T., ROKUSAWA, K., AND NAKASE, A. 1994. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94).* 66–79.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification.* Addison-Wesley. Available at `http://www.javasoft.com`.

GUDEMAN, D. 1993. Representing type information in dynamically typed languages. Tech. Rep. TR93-27, University of Arizona, Department of Computer Science. Sept.

HALSTEAD, R. H. 1985. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4 (Oct.), 501–538.

HARIDI, S. 1981. Logic programming based on a natural deduction system. Ph.D. thesis, Royal Institute of Technology, Stockholm.

HARIDI, S. AND FRANZÉN, N. 1999. Tutorial of Oz. Tech. rep. In Mozart documentation, available at `http://www.mozart-oz.org`.

HARIDI, S. AND SAHLIN, D. 1984. *Efficient implementation of unification of cyclic structures.* Ellis Horwood Limited. In *Implementations of Prolog*, J. A. Campbell, ed.

HARIDI, S., VAN ROY, P., BRAND, P., AND SCHULTE, C. 1998. Programming languages for distributed applications. *New Generation Computing 16,* 3 (May), 223–261.

HARIDI, S., VAN ROY, P., AND SMOLKA, G. 1997. An overview of the design of Distributed Oz. In the *2nd International Symposium on Parallel Symbolic Computation (PASCO 97).* ACM.

HENZ, M. 1997a. *Objects for Concurrent Constraint Programming.* The Kluwer International Series in Engineering and Computer Science, vol. 426. Kluwer Academic Publishers, Boston.

HENZ, M. 1997b. Objects in Oz. Ph.D. thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

IANNUCCI, R. A. 1990. *Parallel Machines: Parallel Machine Languages. The Emergence of Hybrid Dataflow Computer Architectures.* Kluwer, Dordrecht, the Netherlands.

ICHIYOSHI, N., MIYAZAKI, T., AND TAKI, K. 1987. A distributed implementation of Flat GHC on the Multi-PSI. In *4th International Conference on Logic Programming.* MIT Press, 257–275.

JAFFAR, J. AND MAHER, M. 1994. Constraint logic programming: A survey. *J. Log. Prog. 19/20,* 503–581.

LAMMA, E., MELLO, P., STEFANELLI, C., AND VAN HENTENRYCK, P. 1997. Improving distributed unification through type analysis. In *Euro-Par '97 Parallel Processing.* Lecture Notes in Computer Science, vol. 1300. Springer-Verlag, 1181–1190.

LEA, D. 1997. *Concurrent Programming in Java.* Addison-Wesley.

LEUNG, H.-F. 1993. *Distributed Constraint Logic Programming.* Series in Computer Science, vol. 41. World Scientific, Singapore.

LEUNG, H.-F. AND CLARK, K. L. 1996. Constraint satisfaction in distributed concurrent logic programming. *J. Symbolic Computation 21,* 699–714.

LLOYD, J. 1987. *Foundations of Logic Programming, Second Edition.* Springer-Verlag.

MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4,* 2 (Apr.), 258–282.

MEHL, M. 1999. The Oz virtual machine - records, transients, and deep guards. Ph.D. thesis, Technische Fakultät der Universität des Saarlandes.

MEHL, M., SCHULTE, C., AND SMOLKA, G. 1998. Futures and by-need synchronization for Oz.

MEHLHORN, K. AND TSAKALIDIS, A. 1990. Data structures. In *Handbook of Theoretical Computer Science – Volume A: Algorithms and Complexity*, J. van Leeuwen, Ed. Elsevier MIT Press, 301–341.

MOZART CONSORTIUM. 1999. The Mozart programming system (Oz 3). Available at `http://www.mozart-oz.org`.

OTTE, R., PATRICK, P., AND ROY, M. 1996. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice-Hall PTR, Upper Saddle River, N.J.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1995. A survey of distributed garbage collection techniques. In the *International Workshop on Memory Management.* Lecture Notes in Computer Science, vol. 986. Springer-Verlag, Berlin, 211–249.

PODELSKI, A. AND SMOLKA, G. 1997. Situated simplification. *Theoretical Computer Science 173*, 209–233.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 23–41.

ROKUSAWA, K., NAKASE, A., AND CHIKAYAMA, T. 1996. Distributed memory implementation of KLIC. *New Generation Computing 14,* 3, 261–280.

SARASWAT, V. A. 1993. *Concurrent Constraint Programming.* MIT Press.

SCHULTE, C. 1997. Programming constraint inference engines. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1330. Springer-Verlag, Schloß Hagenberg, Austria, 519–533.

SHAPIRO, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3 (Sept.), 413–510.

SMOLKA, G. 1995. The Oz programming model. In *Computer Science Today.* Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, 324–343.

SMOLKA, G. 1996. Problem solving with constraints and programming. *ACM Computing Surveys 28,* 4es (Dec.). Electronic Section.

SMOLKA, G. 1998. Concurrent constraint programming based on functional programming. In *Programming Languages and Systems*, C. Hankin, Ed. Lecture Notes in Computer Science, vol. 1381. Springer-Verlag, Lisbon, Portugal, 1–11.

SMOLKA, G., SCHULTE, C., AND VAN ROY, P. 1995. PERDIO—Persistent and distributed programming in Oz. BMBF project proposal. Available at `http://www.ps.uni-sb.de`.

STROUSTRUP, B. 1997. *The C++ Programming Language, Third Edition.* Addison-Wesley.

SUN MICROSYSTEMS. 1997. *The Remote Method Invocation Specification.* Available at `http://www.javasoft.com`.

SUNDSTRÖM, A. 1998. Comparative study between Oz 3 and Java. Tech. rep., Uppsala University and Swedish Institute of Computer Science. July.

TAYLOR, A. 1991. High-performance Prolog implementation. Ph.D. thesis, Basser Department of Computer Science, University of Sydney.

TEL, G. 1994. *An Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, United Kingdom.

VAN ROY, P. 1994. 1983–1993: The wonder years of sequential Prolog implementation. *J. Log. Prog. 19/20*, 385–441.

VAN ROY, P. 1999. On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99).* Tohoku University, Sendai, Japan.

VAN ROY, P., BRAND, P., HARIDI, S., AND COLLET, R. 1999. A lightweight reliable object migration protocol. Lecture Notes in Computer Science, vol. 1686. Springer Verlag.

Van Roy, P., Haridi, S., and Brand, P. 1999. Distributed programming in Mozart – A tutorial introduction. Tech. rep. In Mozart documentation, available at `http://www.mozart-oz.org`.

Van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M., and Scheidhauer, R. 1997. Mobile objects in Distributed Oz. *ACM Trans. Program. Lang. Syst. 19,* 5 (Sept.), 804–851.

Veen, A. H. 1986. Dataflow machine architecture. *ACM Comput. Surv. 18,* 4 (Dec.), 365–396.

Warren, D. H. D. 1977. Applied logic–its use and implementation as a programming tool. Ph.D. thesis, University of Edinburgh. Reprinted as Technical Note 290, SRI International.

Wikström, C. 1994. Distributed programming in Erlang. In the *1st International Symposium on Parallel Symbolic Computation* (*PASCO 94*). World Scientific, Singapore, 412–421.

Received February 1998; revised September 1998; accepted December 1998