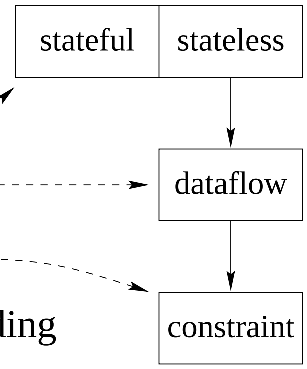
Efficient Logic Variables for Distributed Computing

March 29, 1999

Peter Van Roy

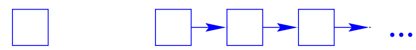
Dept. of Computing Science and Engineering
Université catholique de Louvain

Overview

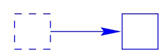


- A distributed state-aware store
- A distributed dataflow store
- A distributed constraint store
- Examples: client-server, variable binding
- Defining the algorithm
- Centralized rational tree unification
- Generalizing to a distributed setting
- Distributed rational tree unification
- Proof of total correctness
- What about efficiency?
- Conclusions and related work

A distributed state-aware store



- It is important to separate stateless and stateful data
  - We observe that they are fundamentally different
  - Reasoning about them is very different (monotonic vs. non-monot.)
  - Distributed implementation is very different
  - Programming with them is very different (values vs. addresses)
- Stateless data are preferable, but they are of limited use
  - More efficient, easier to reason about
  - But a value can't be changed!

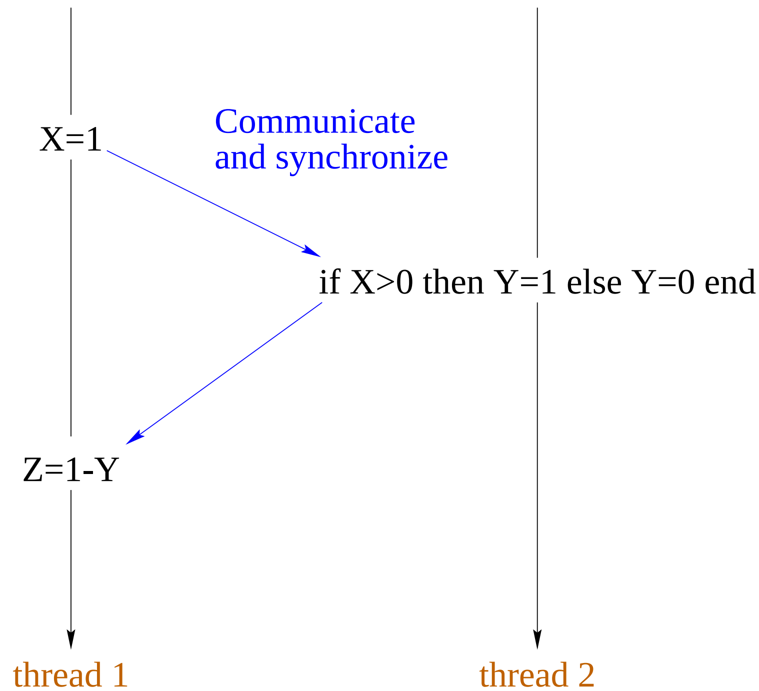


- Stateless data can be made more useful by making them dataflow
  - Separate declaring a variable from binding it. Variables can be assigned, but only to one value.
  - Useful programming primitive for concurrent programming
  - Improves latency tolerance in distributed programming
- Rest of talk
  - How to implement a distributed dataflow store
  - A practical distributed algorithm for rational tree unification

A distributed dataflow store

- Requirements:
  - Sites eventually see the same information and never conflict
  - New information can be added efficiently (‘binding’)
  - Separate variable declaration and binding (‘dataflow’)
- Generalizes a ‘value store’, i.e., the kind of store we all know and love (implemented in Java, C++, SmallTalk, Lisp, ...)
- Benefits:
  - Latency tolerance and third-party independence:
    - Variables can be sent to other sites before being bound
    - Binding does not depend on intermediate sites
  - Practical network transparency:
    - Common idioms are efficient independent of distribution
    - Variable bindings know where to go

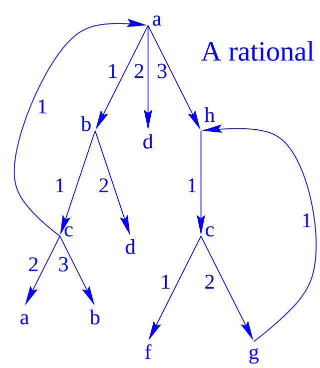
Dataflow behavior



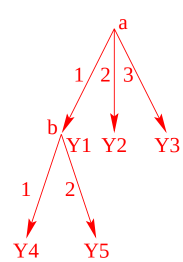
- A thread requiring a value will suspend until it knows the value
- A logic variable conceptually has a fixed value from the moment of its declaration. The value becomes known when the variable is bound.

A distributed constraint store

- Store is conjunction of constraints
  - Three primitive operations:
    Create variable, add constraint, wait for constraint to appear
- Practicality depends on existence of efficient implementation
  - Local operations have same efficiency as value binding
  - Remote operations have same message latency as explicit message passing
- An efficient algorithm exists:
  - Equality constraints over rational trees:
    - Rooted graph with labels
    - Represents any pointer-based data structure
    - Constraints X=Y and X=f(Y1, ..., Yn) give partial knowledge of the tree
  - Distributed rational tree unification
  - Almost as efficient as value binding (up to inverse Ackermann!)



A rational tree

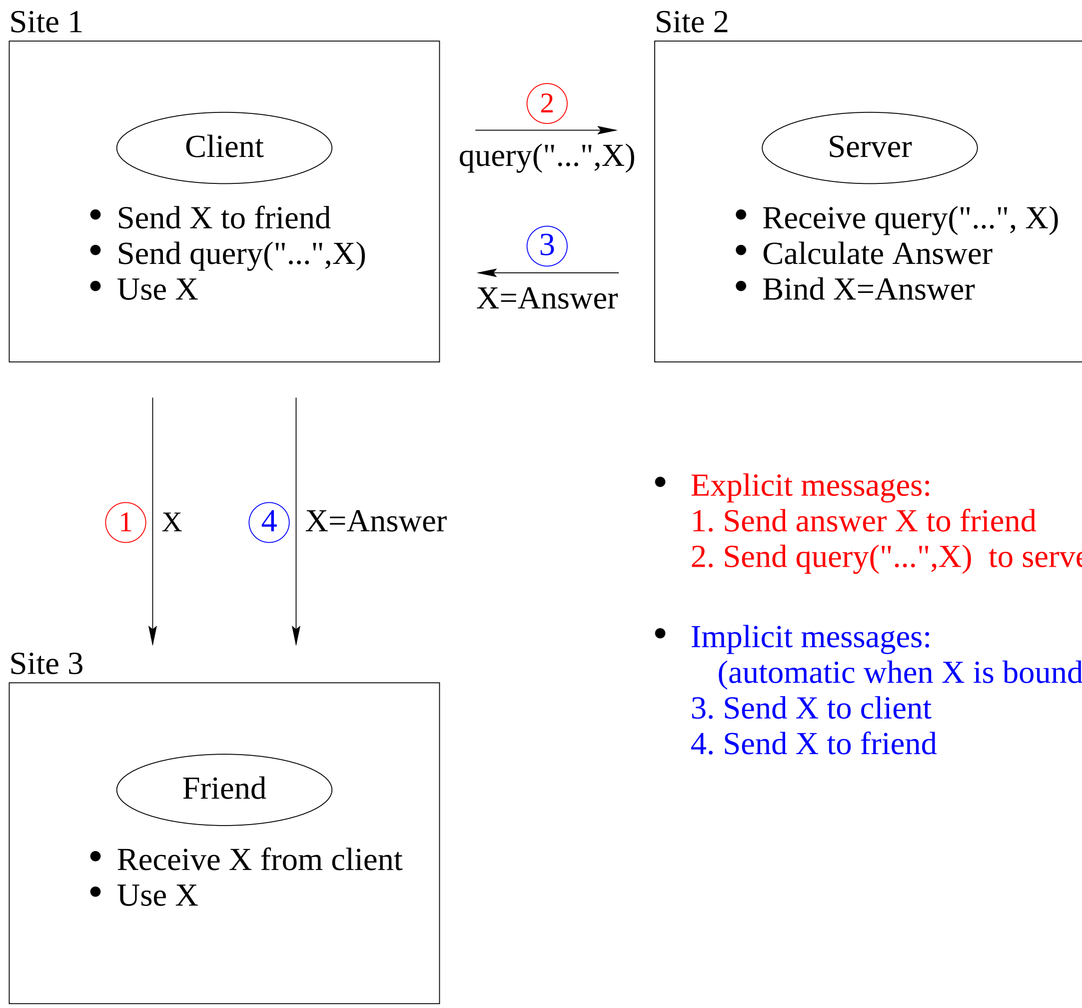


Partial knowledge of the above tree:

X=a(Y1,Y2,Y3)

/\ Y1=b(Y4,Y5)

Client-server-friend example

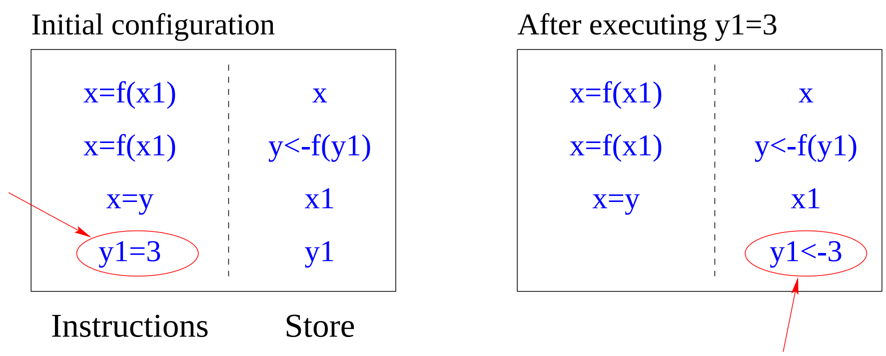


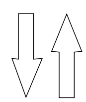
- Explicit messages:
  1. Send answer X to friend
  2. Send query("...",X) to server

- Implicit messages:
  (automatic when X is bound)
  3. Send X to client
  4. Send X to friend

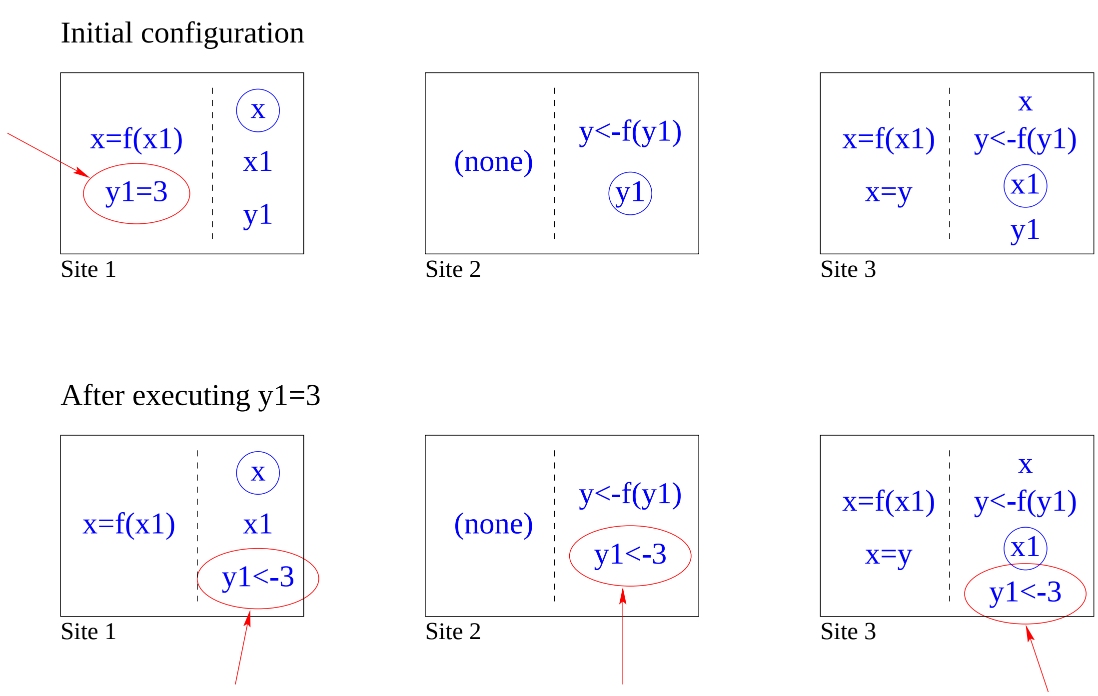
Example of distributed binding

Centralized setting (transparent point of view)





Distributed setting (actual execution)



Defining the algorithm

- Set of atomic reduction rules with interleaving semantics

- Configuration: $\dfrac{\alpha}{\sigma \; ; \; \mu}$
  - α ← Instructions (e.g., set of constraints)
  - μ ← Memo table (to handle cycles)
  - σ ← Store (set of bindings)

- Reduction rule: $\dfrac{\alpha}{\sigma \; ; \; \mu} \Big\| \dfrac{\alpha'}{\sigma' \; ; \; \mu'}$ $\qquad$ $\dfrac{x=3}{\sigma \; ; \; \mu} \Big\| \dfrac{\text{true}}{x{<}{-}3, \sigma \; ; \; \mu}$

  Before → After $\qquad$ Example

  - Atomic reduction of single instruction to new, simpler instruction

- Context:
  - Structural rule and congruence (not shown here)
  - To allow reducing any single instruction in a set of instructions

Extending rules to a distributed setting

- Annotate instructions and bindings with their site:

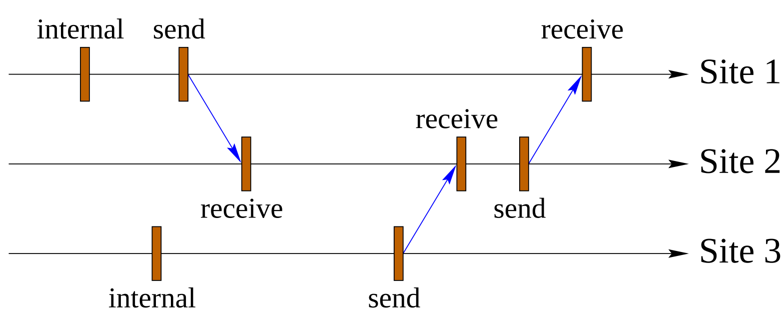
  $(x=3)_s$ $\quad\quad$ $(y<-4)_s$ $\quad\quad$ (s identifies the site)

- To be efficiently implementable, a rule must be a local rule, i.e.:

$(x=3)_s$	true
$\sigma \; ; \; \mu$	$(x<-3)_s \, , \; \sigma \; ; \; \mu$

  - Input and output on same site
  - Except that output instructions can be on any site. They correspond to messages in an asynchronous network.

- A set of local rules defines a transition system [Tel 94]:



Generalizing the centralized algorithm to a distributed setting

- Annotate all instructions and bindings in the centralized unification algorithm by their sites
- In the resulting distributed algorithm, all rules are local except for three:

  - DEREFERENCE and MEMO — Make them local by giving each site its own memo table. This affects efficiency, not correctness.
  - BIND — Replace by four local rules that do coherent distributed variable binding

- The surprising result is that this gives a practical distributed algorithm

- Proof of total correctness is straightforward since the algorithm is so closely related to a known centralized one

- Notation: x is a variable; t is a nonvariable term; u is either
- Variables can be bound to variables
- Memo table stores pairs (x,y) to avoid redoing work already done
- Variables are ordered to avoid binding cycles

What about efficiency?

- Centralized unification can be implemented very efficiently
  - As efficient as value binding (proof: Aquarius Prolog)
- Distributed unification is in two parts: local algorithm and distributed binding algorithm. The former is as efficient as centralized unification; the latter has same message complexity as explicit message sending.
- Mozart system implements this algorithm augmented with:
  - Refinements (ex.: execute WIN rule on owner site)
  - Extensions (ex.: failure detection and handling)
  - Optimizations (ex.: variable registration, asynchronous streams)

Conclusions and related work

- Distributed dataflow store efficiently implemented in Mozart system
- Improves behavior at system level (latency tolerance) and language level (practical network transparency)
- First time that data availability (‘dataflow’) shown to be useful in a distributed setting
  - Previous work used it in a parallel setting to decouple independent computations (I-structures [Arvind et al 80], futures [Halstead 85])
- First complete definition and proof of distributed unification algorithm
  - Previous work in distributed implementation of concurrent logic languages (Multi-PSI [Ichiyoshi et al 87], Parlog [Foster 88], D/C-Parlog [Leung 93], KLIC [Rokusawa et al 96])
  - Usefulness of dataflow store to distributed computing not recognized (e.g., no asynchronous streams); done just to implement language semantics
  - Algorithms defined informally and incorrectly
- This talk only scratches the surface; for more see article (TOPLAS 99)