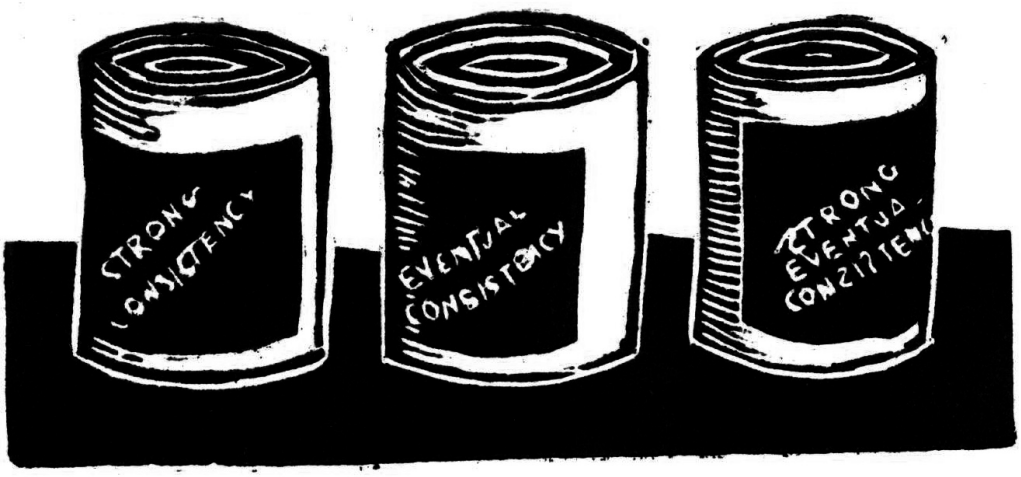
Conflict-free replicated data types
and
distributed concurrent constraint programming

student
Nicolas Brack

supervisor
Peter Van Roy



2012-2013
Université catholique de Louvain

Contents

Many thanks to my supervisor Peter Van Roy, who kept helping me with a smile even after weeks of doubtful progress in comprehension.

Thanks to Géry Debongnie who has co-supervised me half a year.

Thanks to my parents who have read my thesis and fix the most obvious mistakes of English.

Special thanks to my father for the cover image.

Thanks to my readers for taking time to read my thesis and especially to Nicholas Rutherford which helped me to fix a lot of mistakes.

And thanks to everybody which encouraged me and prevented me to give up when it was difficult !

Introduction

This thesis studies two concepts about distributed computing, conflict-replicated data types and concurrent constraint programming. It then tries to bring them together to obtain a powerful paradigm to program efficient distributed programs. The two concepts are linked to the concept of monotonicity. A monotonic component is a component that cannot go back to a state it was previously in. It is forced to progress or to stay idle.

The first concept is *conflict-free replicated data type* abbreviated CRDT. CRDTs themselves are monotonic, so a replica of an instance of a CRDT cannot go back to a former state. CRDTs also provide a way to combine information from different replicas together without merge conflicts. The combination can be done asynchronously, i.e. without requests-and-replies through a network, just notifications. CRDT are described in chapter 1.

In chapter 2, we provide a more formal way to define a CRDT using category theory. Category theory is a powerful tool with a full set of discovered theorems. Category theory augments CRDTs with the possibility to distinguish how the transition between states is done. Other facts from category theory may be applied. In this chapter we also overview a few ways to combine CRDTs together to obtain new CRDTs.

The second concept is *distributed concurrent constraint programming*. Constraint programming is done by discovering constraints on variable until we can find a way to satisfy a logic predicate. We cannot "forget" a constraint, so again the knowledge is monotonic throughout the execution steps. Concurrency means that solving the program may not be sequential. Distributed obviously means the program can be computed by several nodes, or *sites*, of a computer network. Concurrent constraint programming, abbreviated CCP, is the subject of chapter 3.

Another very important point of CCP is that it has the *confluence property*. This means that the language can apply concurrent rules of computation in any order, the result is the same. Confluence is very important to allow threads to be distributed over a network[1]. This confluence property makes the program text a CRDT. To add other CRDTs to this language, those CRDTs must be translated into logic formulation so that they can fit in the knowledge base (known as the *constraint store*) of a concurrently constraint programmed process. A few CRDTs are manually translated in chapter 3.

The last chapter, chapter 4, proposes a new programming language for distributed programming. This language is based on a concurrent constraint programming language and it adds support for CRDT. As such the language supports mutable structure while remaining monotonic. Again, monotonicity is important as it provides a way to asynchronously merge states of different

replicas. In this chapter we also outline some details of implementation, like the coherent denomination of variables over an asynchronous network.

Chapter 1

Conflict-free Replicated Data Types

In this chapter with introduce the first useful concept for distributed programming, conflict-free replicated data types. In the first section, we view three notions of consistency between different sites of a network. The traditional consistency, strong consistency has issues with availability so we advocate the use of a weaker form of consistency. The chosen notion of consistency, strong eventual consistency, is the core idea of the conflict-free replicated data types explained in the second section. Finally we roughly discuss of two ways of communications between distributed replicas of an instance of these data types. A more formal definition using category theory of conflict-free replicated data types is given in next chapter.

1.1 Eventual consistency

The purpose here is to provide a data structure distributed over a large network and manipulated by a large base of users around the world. We shall write a *"site"* in this thesis to speak about a node of this large network. A site is typically a single computer, but it may be a computer cluster with only one interface to other sites.

The provided data structure should follow the CAP properties : consistency, availability and partition-tolerance. Consistency, or *strong consistency* ensures that every site of the network has at any moment a (partial) replica of the same state of the data structure. Availability means that access to this data is always possible in a very short delay. Partition tolerance means that the data structure should be resilient to communication channel failures that leads to temporary partition between sites.

Unfortunately, the CAP theorem tells us that no system satisfying those three desirable properties together exists[2] : one has always to drop one of these properties. This has often been availability as this is sometimes the least critical issue. But in the in the case of interactive applications neglecting availability significantly harms efficiency of applications and thus the patience of human users.

To retain availability during network partition, we would like instead to drop strong consistency for a weaker form of consistency, *eventual consistency*[3]. A

shared data structure is eventually consistent if its replicas converge toward the same state. They converge in finite time, i.e. they should reach this common state when no users manually modify the replicas.

Eventual consistency does not sweep away synchronisation. Sites can still synchronise if they wish to. Strong consistency can even be viewed as a particular case of eventual consistency, since by keeping the same state at any moment we converge obviously to the same state.

Thus eventual consistency may also come at the cost of availability when synchronising. There is another form of consistency, *strong eventual consistency* that forbids synchronisation[4]. In this case, a replica of the shared data structure is coherent with other replicas that *have observed* the same operations. "Observe the same operations" means the data structure is in a state that can occur only after the operations have been applied. Thus we must be able to tell if two states of the data structure have observed the same operations. If it is not the case, we should have a way to merge the two states together so that the new state observes the operations of each. We must be careful not to apply twice an operation observed by both states.

Here is a short summary of the various forms of consistency.

- *Strong consistency* means that at any time, all replicas have the same state (neglecting the time needed to synchronise).
- *Eventual consistency* means that the replicas converge toward the same state if local updates of the replicas eventually stops.
- *Strong eventual consistency* means that the replicas converge toward the same state, like in eventual consistency and that at any time it possible to tell whether a replica dominates another.

In this thesis we advocate that the use of *Strong eventual consistency* is not a hindrance to system development.

1.2 Introduction to CRDTs

How to use strong eventual consistency in a program ? According to its definition, sites communicate one to each other asynchronously. A site receives new state information from one of its neighbour current state. The site can now *merge* this received state with its own state. It has now a state affected by the operations that affected the neighbourand the operations that affected its previous state[5]. We say that the resulting state *observes* the operation of the neighbour and that it *dominates* the current state of the neighbour. Of course, a state of some site always observes the operations committed at that site.

This may lead to inconsistencies if no one is aware which states are old and should be discarded and which states must be considered and merged with another replica. When given two states of the data structure, any computer must be able to determine alone whether they are concurrent or if one dominates the other without any synchronisation or communication with other computers. Some *order relation* on states must be given to determine this. If a state *dominates* another then it is greater in this order and if two states are *concurrent*—when no one dominates the other—there are not comparable in this

order. A transaction at a local replica must result in a state dominating the current local state.

The most common order is obtained from a vector clock. A vector clock is an array of virtual times of each replicas. The virtual time can be the number of operations applied, for example. This local order obtained from vector clocks is call the *causality*. However, any lattice is a valid set of states for a CRDT. The important point with causality is that when comparing two states, it tells us whether one happened before the other or if the two states are concurrent[6].

The sequence of states of a replica must also be *monotonic*[7]. This means a site cannot return to a former state, avoiding issues such as applying indefinitely a loop over successive dominant states for example.

(1.1) **Example.** Let us look up at a set data type[8]. As explained in figure 1.1, using a standard implementation of a set without synchronisation can only lead to errors. Sets are naturally ordered by set inclusions. Thus if we forbid users to remove elements from sets, every update adds an element and the set changes to a dominating set.

By using more sophisticated states we may remove elements from the set, keeping monotonicity of the states. The resultant set, is not monotonic, as it may return to a former state back. We say the add-remove set is a view of the more complex data structure. We will define more formally the notion of view in chapter 2. For example consider our states to be formed of two sets A and R. When a user requests to add an element to the viewed set, it is actually added to A. When another user requests to remove this element, it is actually added to R. Thus the viewed set is $A \setminus R$.

This construction is not perfect : when a user removes an element, it cannot be put back later.

1.3 Definition of CRDTs with semi-lattices

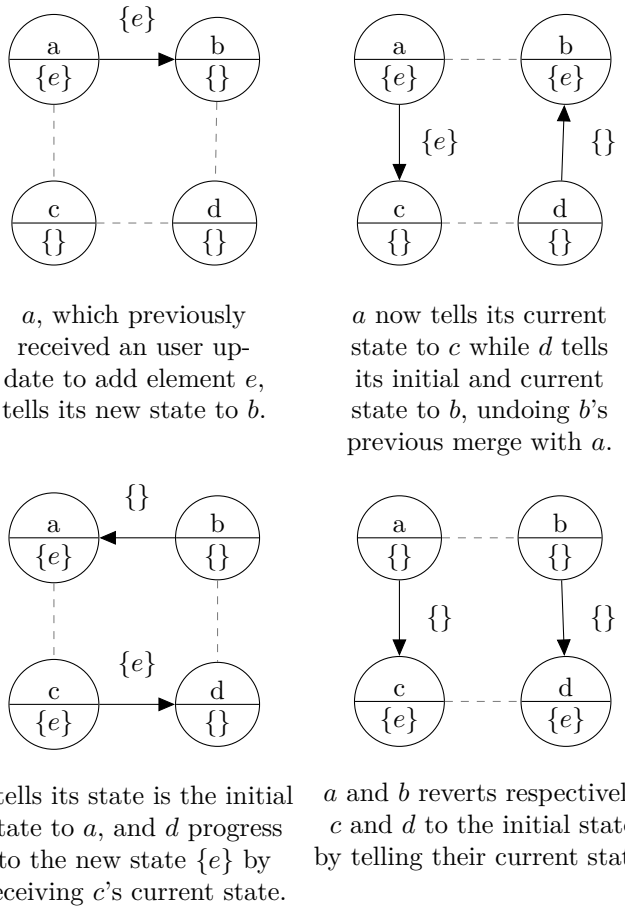
Eventual consistency is now a matter of ensuring that different orderings of the merged update will converge towards the same state. The easiest way is to ensure that applying two updates/merges in any order always leads to the same state. We say the two updates/merges *commutes*.

1.3.1 Semi-lattices

This commutativity is where the notion of *semi-lattice* comes in. A *semi-lattice* is a set together with an order that might be partial—some elements are not comparable—and a *joint* operation which for two states outputs the minimal state greater than each of the input state. The joint operation is usually written $\vee$ and sometimes called "supremum".

(1.2) **Definition.** A semi-lattice is a set L together with a binary operation $\vee$ called the joint such that for every x, y and z in L :

$$x \vee y = y \vee x \qquad x \vee x = x \qquad (x \vee y) \vee z = x \vee (y \vee z)$$



a, which previously received an user update to add element e, tells its new state to b.

a now tells its current state to c while d tells its initial and current state to b, undoing b's previous merge with a.

b tells its state is the initial state to a, and d progress to the new state $\{e\}$ by receiving c's current state.

a and b reverts respectively c and d to the initial state by telling their current state.

Figure 1.1: In this scenario, nodes a, b, c and d do not synchronise and do get get back to former states. You can observe that a user update at node a may eventually be lost, even though every node receives the update ! Arrows represents notifications of states which imply a state merge.

A semi-lattice induces a partial order relation $\leqslant$, where element x is smaller than element y if their joint is y itself. So every semi-lattice is a poset, a.k.a. a set with an order relation. The order is given by

$$x \leqslant y \text{ if and only if } x \vee y = y.$$

1.3.2 CRDTs

A *Conflict-free replicated data type*, also written CRDT, is the type of a data structure whose state space form a semi-lattice and whose actual sequence of states traversing the semi-lattice is monotonic. A CRDT is modified by two operations : a *merge* operation, which corresponds to the lattice's joint, and a *commit* operation, which is locally executed when a user of some site wants to

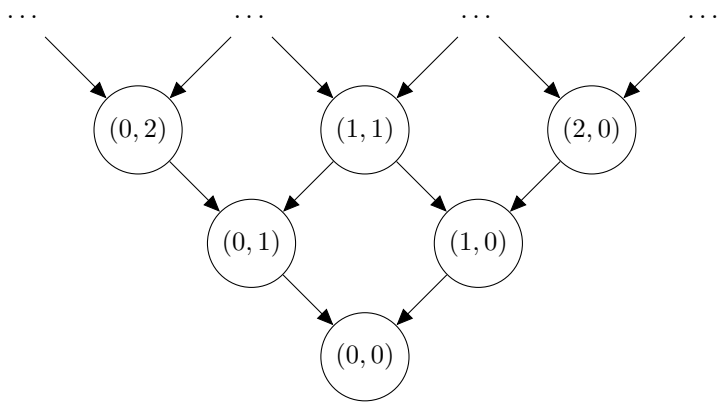


Figure 1.2: The graph representing the semi-lattice $\mathbb{N} \times \mathbb{N}$. A state dominates another if there is an oriented path of arrows from the former to the latter. The join of two nodes is the closest common node that dominates both states.

update the data structure. This commit must be *growing*, i.e. it must output a state greater than the input state. Every replica should also have a common "clean" starting state to avoid tainting existing replicas when creating a new "blank" replica, but this is not required for the CRDT to work.

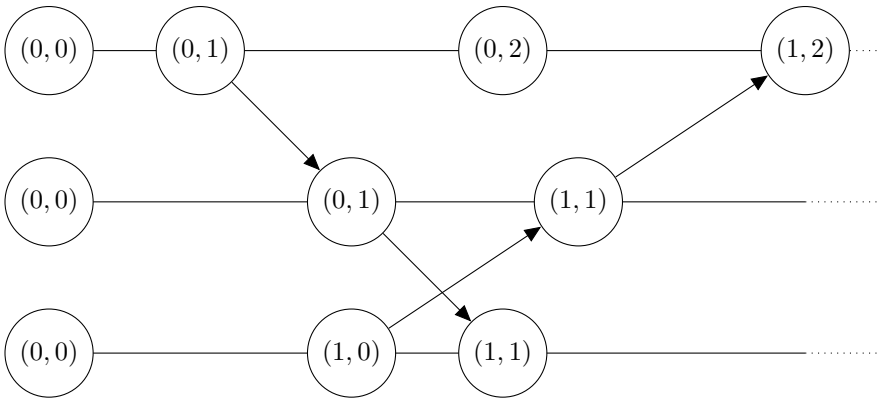


Figure 1.3: A CRDT with three replicas associated with the semi-lattice of figure 1.2. Each circle is a state value. Each replica starts with the state $(0, 0)$.

1.4 State-base merges and growth-based merges

So far we mostly looked at a local replica, but how do these replicas share their state one to each other in order to execute merges ? The obvious answer is to send the whole state from a replica to another. This is called *state-based merges*[9]. States-based merges are very effective for CRDT whose state consumes little memory.

On the other hand, state-based merges for CRDT with states that consume large memory would be highly ineffective, especially if updates are frequent and modify a tiny part of the state. A *growth-based* merge sends only the modified part of a state. Consider for now that *growth-based* merge are notified by sending the label of the function that was applied.

1.4.1 Growing operations and commutativity

We very briefly mentioned the notion of growth previously. A *growing* operation for a CRDT is a function over the states of the CRDT such that *it maps a state to states that dominate it*[10]. Instead of sending the whole state to another replica, a site may just tell what growing operation it applied and let its peers recompute the state.

Care must be taken, though. If using growth operations guarantees monotonicity of states at a local replica, it does not guarantees strong eventual consistency. The problem is that different sites can compute different states depending on which order the operations are applied. If this happens, the replicas will have a state that will eventually *diverge*.

Since it is the order of updates between different replica is uncertain, *commutativity* of the operators solve the problem. Two operations commute if applying the operation to a state in any order yields the same final state.

(1.3) **Example.** Consider the add-only set over natural numbers, $\mathcal{P}(\mathbb{N})$ and an element e. The operation $\texttt{add}_e$ defined by $\texttt{add}_e(S) = S \cup \{e\}$ is a growing operation. Indeed, clearly $S \subseteq S \cup \{e\}$ holds.

Furthermore, two such add operations commutes. Indeed,

$$\texttt{add}_e\big(\texttt{add}_{e'}(S)\big) = S \cup \{e\} \cup \{e'\} = \texttt{add}_{e'}\big(\texttt{add}_e(S)\big).$$

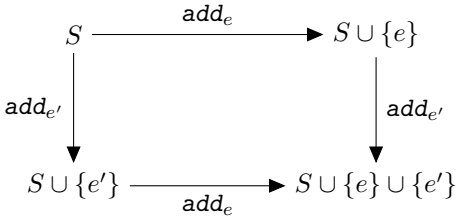


Figure 1.4: commutativity of operations $\texttt{add}_e$ and $\texttt{add}_{e'}$.

1.4.2 Growing operations and idempotence

Another issue is that sites may receive the same update twice. Thus updates should be idempotent, meaning that applying twice is equivalent to once. Rewriting of standard functions as idempotent ones can be done at runtime. So a commit operation is made of two part : a *conversion operation* that converts the operation in idempotent terms, and an *application operation* that applies and communicates those idempotent operations.

(1.4) **Example.** Let $\mathbb{N}$ be the add-only counter whose states are integers. Every site begins with its replica at the state '0'. Its interface provides only one operation to a user, `increments` that increment the counter from the value currently seen by the user.

Imagine the sites are arranged in a network similar to 1.1. If a increments its counter from 0 to 1 and tells its neighbour "to increment", then both these neighbours should increment their local replicas and tell the last site, d "to increment". Thus d will increment twice, which is not correct.

The solution is to convert the `increment` operation to a $\texttt{raiseTo}_1$ that raises the counter to 1 if it has a smaller value (which is equivalent to a state-based merge in this trivial example) and to send that operation to neighbours.

The conversion of operation might also have to solve issues with commutativity. In some complex cases it is probably hard to ensure commutativity this way. In those cases, an operation that is received and doesn't make sense, may be "*suspended*". When the local site observes some latter operation such that it would make sense to apply the former operation, then both are applied.

(1.5) **Example.** Consider adding and removing the element e from the $A \setminus R$ set of example 1.1. Let these operations be $\texttt{add}_e$ and $\texttt{del}_e$. We furthermore assume every replica is at the initial state $A = \{\}$, $R = \{\}$.

$\texttt{add}_e$ and $\texttt{del}_e$ obviously do not commute. If $\texttt{add}_e$ is applied first and $\texttt{del}_e$ secondly, then e is added and immediately removed from the viewed set. If $\texttt{del}_e$ is applied first, then nothing should be done as e is not in the viewed set. When $\texttt{add}_e$ is applied after, then e is now in the viewed set.

But if we convert $\texttt{del}_e$ to an operation $\texttt{del}'_e$ that suspends until the element e is in the set before removing it, then the operation commutes as both will output

$$A \setminus R = \{e\} \setminus \{e\} = \{\}.$$

There is no good choice between state-based merges and growth-based merges. The pros and cons of each depends on the used CRDT. But generally, the growth-based merged is better for CRDT whose states are large and whose updates only alter a tiny part of the information contained by the state.

Chapter 2

Categories

We shall motivate the use of categories now. Roughly speaking, a category is a general mathematical framework describing various objects by the possible mapping between those objects rather than by building them from more primitive structures and describing them from these building blocks.

A conflict-free replicated data type, as introduced in the previous chapter, may be easily described in such a framework. The considered objects are the possible states of a CRDT. The inner description of these states is not very relevant. What is more relevant is how to select a new state when committing an operation or merging two states.

Commits can be modelled with a possible mapping between two states. Merge can locally be described with similar mappings. Globally, though, they correspond exactly to a well-know categorical construction, the coproduct.

Category theory also offers an easy and yet powerful way to map categories between each other, as we shall see in the end of the chapter. This way the structure(s) that a CRDT intends to represent are simply obtained from these category-to-category mappings, after one properly defines the intended structure as a category.

2.1 Categories

Some theories in mathematics, or in computer science, share common aspects. Consider for example, the various notions of morphism (maps that preserves mathematical structures) or of operations. Category theory tries to capture these similarities in a common model[11]. Like set theory, category theory is concerned with collections. However unlike set theory, it has a particular interest in functions within a theory or framework and between theories.

A CRDT type can be viewed as a collection of the possible states of one of its instance. This is what we did when we defined a CRDT with a semi-lattice. But by using category theory we also take care of the transitions between those states.

(2.1) **Definition.** A *category* $\mathcal{C}$ is made of

- a collection of *objects* and a collection of *arrows* between two objects (The collection of objects shall indistinguishably be written $\mathcal{C}$ as well, but do

not confuse the category with its collection of objects which is only a part of it),

- a set of *arrows* from A to A' for objects A, A' in $\mathcal{C}$, written $\mathcal{C}(A, A')$ (A is called the *input* object and A' the *output* object),
- a *composition operator* $\circ$ that combines arrows from $\mathcal{C}(A, A')$ and $\mathcal{C}(A', A'')$ into arrows of $\mathcal{C}(A, A'')$ for any but not necessarily distinct objects A, A' and A'' in $\mathcal{C}$.

A category must respect the next axioms.

- The law $\circ$ is *associative* : for any arrows f, g and h that can be composed,

$$f \circ (g \circ h) = (f \circ g) \circ h.$$

- For every object $A \in \mathcal{C}$, there exists an *identity* arrow $\mathbf{1}_A \in \mathcal{C}(A, A)$, such that this arrow is neutral; that is, for any other object A' and every arrow $f \in \mathcal{C}(A, A')$ and $g \in \mathcal{C}(A', A)$, the following equalities holds :

$$\mathbf{1}_A \circ f = f, \qquad g \circ \mathbf{1}_A = g.$$

A category can embeds in another. This is the notion of *subcategory* obtained when the set of objects of one category is the subset of another and each set of arrow of the former category is a subset of the corresponding set of arrow of the latter category. This is useful to represent constraint on data types. As a primitive or combined categorical data type may have states that are forbidden by constraints. For example, the couple of a set and an integer that counts the number of elements in the set forbids the states where the integer is different from the size of the set. Thus we should take the sub-category of the category of a set and a integer.

(2.2) **Definition.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories. If the collection of objects of $\mathcal{D}$ is a sub-collection of $\mathcal{C}$ and if for every object $X, Y \in \mathcal{D}$, the set of arrow $\mathcal{D}(X, Y)$ is a subset of $\mathcal{C}(X, Y)$, then we say that $\mathcal{D}$ is a *subcategory* of $\mathcal{C}$ and we write $\mathcal{D} \subseteq \mathcal{C}$.

The notion of category is used for a wide range of domains. Below is a few examples. The two first following examples are not really related to the subject of this thesis, but there are easier to understand. There's quite a few examples, so they can be read later, when they will be needed.

Example will be used to convert a lattice-based CRDT to a category and example convert standard group data type (that is, data types whose values can be combined with a binary operator that admits inverse elements and neutrals) to a category as well. These two examples thus provides a way to convert standard description of data types using sets to categories.

(2.3) **Example.** The collection of sets forms the objects of the category Set. An arrow of $\mathsf{Set}(A, B)$ is a map, a function whose domain is total, from A to B. Maps can indeed be composed using the standard function composition and the standard function composition is clearly associative. The identity map $\mathsf{id}_A \colon A \to A \colon a \mapsto a$ is clearly the identity arrow $\mathbf{1}_A$.

(2.4) **Example.** The class of algebraic groups also forms a category, Grp. Its arrows are group morphisms, the maps that commute with the input and output group operations. Composition of two group morphisms and identity maps remains a group morphisms. Associativity and identity properties are deduced as in the case of Set.

It is worth noticing that Grp is not a subcategory of Set. Indeed, a set can have multiple group operations. For example, both XOR and XNOR are valid group operations over $\{\mathsf{true}, \mathsf{false}\}$.

(2.5) **Example.** Let $\mathcal{P}$ be a set together with an order relation $\leqslant$. They induce a category $\mathcal{P}$ whose objects are the elements of $\mathcal{P}$. For every object $x, y \in \mathcal{P}$, $\mathcal{P}(x, y)$ is made of one single arrow[12] if $x \leqslant y$ and $\mathcal{P}(x, y) = \{\}$ else. The composition rule is straightforward : to the only arrows of $\mathcal{P}(x, y)$ and $\mathcal{P}(y, z)$, is the only arrow of $\mathcal{P}(x, z)$ associated. This arrow must exist by transitivity of $\leqslant$. Such a composition is obviously associative. The identity arrow must exist by reflexivity of $\leqslant$.

Observe how an arrow goes from an element to another *larger* element.

(2.6) **Example.** Similarly, every group $(\mathcal{G}, \cdot)$ induces a category. Its objects are the elements of $\mathcal{G}$ and there is a single arrow between elements $g, h \in \mathcal{G}$, $\mathcal{G}(g, h) = \{g \cdot h^{-1}\}$. Composition is given by

$$(h \cdot g^{-1}) \circ (g \cdot f^{-1}) = h \cdot g^{-1} \cdot g \cdot f^{-1} = h \cdot f^{-1}.$$

Associativity follows, and the identity arrow is always the identity element.

2.2 Functors

Since we shall represent each CRDT with a category, we also need a way to represent functions that take a CRDT in argument and output another data types. This is more generally achieved by *functors*, a sort of "function between categories". Functors maps the objects to objects, but also arrows to arrows.

Thus we keeps some information about the possibles transitions in a CRDT (that are represented as arrow). Imagine there is a non-return point state N in the output category. Then we can identify every arrow whose of the CRDT category whose output object is send to N by the functor. These arrow correspond to arrow that reach the non-return point at the CRDT level.

(2.7) **Definition.** A *functor* $\mathcal{F} \colon \mathcal{C} \to \mathcal{D}$ maps an object of $X \in \mathcal{C}$ to an object of $\mathcal{F}X \in \mathcal{D}$ and an arrow $a \in \mathcal{C}(X, Y)$ to an arrow of $\mathcal{F}a \in \mathcal{D}(\mathcal{F}X, \mathcal{F}Y)$ under the following conditions :

- The image of an identity arrow is an identity arrow : $\mathcal{F}\mathbf{1}_X = \mathbf{1}_{\mathcal{F}X}$.
- The image of a composition of arrows is the composition of the images of the arrows : $\mathcal{F}(f \circ g) = (\mathcal{F}f) \circ (\mathcal{F}g)$.

Here is again a easy example followed by an interesting one. The second example explains how to get the data type a CRDT intends to represent from that CRDT. In other word, it is an example of a view of a simple CRDT.

(2.8) **Example.** Let $\mathcal{U}\colon \mathsf{Grp} \to \mathsf{Set}$ be the category that maps a group to its supporting set and that maps a group homomorphism to itself viewed as a standard set map. Such a functor is named a *forgetful* functor, as it just forgets some of the structure of the objects or arrows of the input category.

(2.9) **Example.** Let N^2 be the category made over the set $\mathbb{N}\times\mathbb{N}$ and the order relation $\leqslant$ given by $(m_1, n_1) \leqslant (m_2, n_2)$ if and only if $m_1 \leq m_2$ and $n_1 \leq n_2$, as done in (2.5). Let also $\mathbb{N}$ be the category from group $(\mathbb{N}, +)$ as defined in (2.6).

The functor $\mathcal{F}$ that maps (n, m) to $n - m$ is well-defined. The sole possible arrow between (m_1, n_1) and (m_2, n_2) is sent to the only arrow between $m_1 - n_1$ and $m_2 - n_2$. The identity arrow of (n, m) is then sent on the identity arrow, the unique arrow between $n-m$ and itself. Composition is obviously preserved.

This last example is of particular interest. Indeed assume the objects of N^2 and $\mathbb{N}$ represent the possible state of some computer data structure. Also assume that arrows represent the possible state transitions. Then a functor is an *view* of a data structure into another, in this case a couple of integers is interpreted as a single integer.

Thus to represent an integer, we may choose to use a specific data structure with distinguished properties. For example, notice that N^2 has no non-trivial loop, i.e. once a data structure leaves some state (n, m), it cannot return to it later. As such, a couple of integer with the restricted transitions is a *monotonic* data structure.

Finally note that nothing forces to have at most one transition arrow between two states, when viewed as a category. As such multiple transitions between two states can be used to distinguish different side effects.[13]

2.3 Products and coproducts

Consider a CRDT category obtained from a CRDT semi-lattice, defined in (1.2). As shown after that definition, a lattice induces a poset which by example (2.5) induces a category. To find how to represent the join operator $\vee$ of this semi-lattice in this category, we need to explore a bit more of category theory.

We introduce the notion of *product* and *coproduct*. The latter correspond to the joint operator. The former is its "dual" notion and find the largest state *dominated* by the current states. Knowing the largest states dominated by the current states could be useful for garbage collection, for example.

Using coproducts highlight a nice property, the *universal property* that ensures that a merged state has transition to every states that dominates the two states it merged.

(2.10) **Definition.** Let $\mathcal{C}$ be a category and $\mathcal{X}$ a collection of objects of $\mathcal{C}$. The *product* of $\mathcal{X}$ is a object P together with projection arrows $p_X\colon P \to X$ for every X in $\mathcal{X}$.

It must furthermore obey the *universal property* which tells that P and the p_X's are the finest choice that can be made. It states that if there is another object P' with projections arrows p'_X for $X \in \mathcal{X}$, then this projection factorizes through P. In more precise terms, there must exist a unique $q\colon P' \to P$ such that for every $X \in \mathcal{X}$ we have $p'_X = p_X \circ q$.

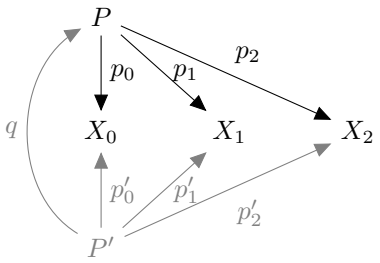


Figure 2.1: P is the product for X_0, X_1 and X_2

Figure 2.1 shows graphically what it means. The product of three objects X_0, X_1 and X_2 is simply an object P with an arrows p_i from P to each X_i. There are several possible P, so we choose the minimal in the following sense. If we take any other object P' that has also an arrow p'_i to each X_i then there must be an arrow q from P' to P, *common* to p'_0, p'_1 and p'_2, such that p'_i is the composition of p_i and q.

Let us show the concept on a easy example, the category Set.

(2.11) **Example.** Consider the category Set, and two sets A and B. Their categorical product is their Cartesian product $A \times B$ together with the respective projections

$$p_A \colon A \times B \to A \colon (a, b) \mapsto a,$$
$$p_B \colon A \times B \to B \colon (a, b) \mapsto b.$$

If we have a set S and two function $f \colon S \to A$ and $g \colon S \to B$, let $q \colon S \to A \times B$ be the function defined by $q(s) = \big(f(s), g(s)\big)$

There is a dual version of this notion, called the *coproduct*. These notions are linked as follows : if one inverses the orientation of every arrow of a category $\mathcal{C}$, then its products become coproducts and vice-versa.

(2.12) **Definition.** Let $\mathcal{C}$ be a category and $\mathcal{X}$ a collection of objects of $\mathcal{C}$. The *coproduct* of $\mathcal{X}$ is an object C together with arrows $c_X \colon X \to C$ for every X in $\mathcal{X}$.

It also has to obey a *universal property*. In this case, it states that if there is another object C' together with arrows $c'_X \colon X \to C'$ for $X \in \mathcal{X}$, there exists a unique $q \colon C \to C'$ such that for every $X \in \mathcal{X}$ we have $c'_X = q \circ c_X$.

For two objects A and B, the product and the coproduct are respectively written $A \times B$ and $A \sqcup B$. Figure 2.2 illustrates the definition in the case of the coproduct of three objects. The coproduct of zero objects is called the *initial object*. It has arrows to any other object of the category by the universal property. We usually ignore the coproduct of zero objects when speaking of coproducts.

(2.13) **Example.** Consider again the category Set, and two sets A and B. Their coproduct is their disjoint union $A \sqcup B$. $A \sqcup B$ is defined by labelling

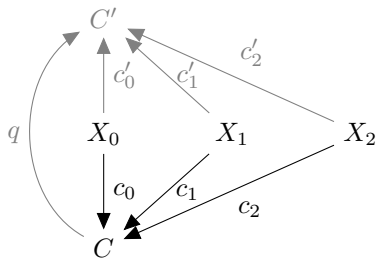


Figure 2.2: C is the coproduct for X_0, X_1 and X_2

elements of A as "*left*" and elements of B as "*right*" and to take the union of labelled "*left*" elements from A and labelled "*right*" elements from B. Formally,

$$A \sqcup B = \big\{(\mathit{left}, a)\big|a \in A\big\} \cup \big\{(\mathit{right}, b)\big|b \in B\big\}.$$

The coprojections are the standard inclusions :

$$c_A \colon A \to A \sqcup B \colon a \mapsto (\mathit{left}, a),$$
$$c_B \colon B \to A \sqcup B \colon b \mapsto (\mathit{right}, b).$$

If we have a set S and two function $f \colon A \to S$ and $g \colon B \to S$, let $q \colon A \sqcup B \to S$ be the function defined by

$$q(x) = \begin{cases} f(a) & \text{if } x = (\mathit{left}, a) \text{ where } a \in A, \\ g(b) & \text{if } x = (\mathit{right}, b) \text{ where } b \in B. \end{cases}$$

Let us now make the link with the semi-lattice. There is an interesting property to observe.

(2.14) **Proposition.** Let L be a semi-lattice with joint $\vee$ and let x, y and z be elements of L. If $x \leqslant z$ and $y \leqslant z$ for the induced partial order relation $\leqslant$, then $x \vee y \leqslant z$.

The proof is not very hard. If $x \leqslant z$, then, by definition of the induced partial order relation, $x \vee z = z$ and similarly for y. Thus we can conpute :

$$(x \vee y) \vee z = x \vee (y \vee z) = x \vee z = z,$$

so $(x \vee y) \vee z = z$ holds and thus $x \vee y \leqslant z$.

(2.15) **Example.** Let L be a semi-lattice with joint $\vee$. L induces a partial order relation on itself. Thus L is a poset. Let $\mathcal{L}$ be the category induced from this poset as in example (2.5). Then the coproduct of two elements is their joint $x \vee y$.

Indeed we obviously have $x \leqslant x \vee y$ and $y \leqslant x \vee y$ thus there is an arrow from x to $x \vee y$ and an arrow from y to $x \vee y$.

If z is also such that $x, y \leqslant z$, then by (2.14), $x \vee y \leqslant z$, so the unique arrow between z and x (or y) is factorised by the unique arrow between z and $x \vee y$ and the arrow between $x \vee y$ and x (respectively y).

2.4 Conflict-free replicated data type as a category.

2.4.1 CRDT states

As exemplified in (2.6) a data type can be represented as a category whose objects are the possible states of an instance of the data type and whose arrows are the allowed transitions between states. That category represents a *data type*, not an instance of that type. An instance of that data type is simply a container of one of the objects from the category.

When speaking of CRDT, we sometimes speak of an *instance* of a conflict-free replicated data *type*. Using CRDT as an abridged form of instance of CRDT may confuse the inattentive reader. The above definition is vulgarly "the type of a CRDT instance". A stated four paragraphs earlier, an instance of a CRDT is a container of one object from the CRDT category. So be aware there are two levels in a CRDT : the type level, the category of possible states, and the instance level, a mutable container to a particular state from that category.

Unlike the semi-lattice formalism, nothing prevents having more than one arrow between two states. For example we can use an arrow per possible combination of side effects—like message printed on the screen—in every transition between two states.

The formalism of semi-lattices embeds in one of the Categories, as it is shown in (2.15). The joint operator is simply taken as the coproduct of the two objects. The important point concerning CRDTs, is to ensure *monotonicity*. This means there cannot be non-trivial "cycle" of arrows in the state space. This is actually the direct translation of the antisymmetry property of an order relation.

(2.16) **Definition.** A category $\mathcal{C}$ is a *CRDT category*, or simply *CRDT*, if for any composable arrow f, g of the category such that $f \circ g$ is an arrow in $\mathcal{C}(X, X)$, f and g must also be arrows in $\mathcal{C}(X, X)$. Moreover the category must admit finite non-trivial coproducts.

2.4.2 CRDT views

A view is a functor that maps a CRDT type to another type without any more constraints than those of a functor. The functor in example (2.9) is an example of a view. A view from a CRDT type to another CRDT type is called a conflict-free replicated functor, or CRF.

2.4.3 CRDT merges

Strong eventual consistency of a CRDT is possible because a CRDT Category admits coproducts. Indeed : assume we are given n instances of the CRDT over n sites. The states contained by each instance are $x_1, \dots x_n$. Those states $x_1, \dots x_n$ correspond to objects of the CRDT category.

If no operation is applied, each instance should merge two by two. But taking recursively coproducts two by two in any order or taking the coproduct of $x_1, \dots, x_n$ at the same time leads to the same state[14]. This property correspond to the associativity of the joint operator $\vee$. Thus at any time, the

state towards which $x_1, \ldots, x_n$ will converge is their coproduct, the minimal state that dominates each x_i by virtue of the *universal property*.

Here is two simple example of a CRDT.

(2.17) **Example.** Assume we have k sites each with one replica of a counter CRDT. Each replica has a state of the CRDT category $\mathcal{Z}_k$ whose set of possible states is $(\mathbb{N}^2)^k = \mathbb{N}^{2k}$. We shall write a state by $(p_0, n_0, \ldots, p_{k-1}, n_{k-1})$ where the p_i and n_i are non-negative integers.

The initial state is obviously $(0, 0, \ldots, 0, 0)$. The coproduct is given by :

$$(p_0, n_0, \ldots, p_{k-1}, n_{k-1}) \sqcup (p'_0, n'_0, \ldots, p'_{k-1}, n'_{k-1}) = \\ (\max(p_0, p'_0), \max(n_0, n'_0), \ldots, \max(p_{k-1}, p'_{k-1}), \max(n_{k-1}, n'_{k-1})).$$

The ith site is allowed to increment either p_i and n_i only.

We want to retrieve a single integer modifiable by every users. Thus this counter should remember every increment/decrement requested. The outputted integer lies in the category $\mathbb{Z}$ obtained from the additive group $\mathbb{Z}$ as done in (2.6). The view from $\mathcal{Z}_k$ to $\mathbb{Z}$ is given by the functor $\mathcal{V}$ that maps an object $(p_0, n_0, \ldots, p_{k-1}, n_{k-1})$ to :

$$\mathcal{V}(p_0, n_0, \ldots, p_{k-1}, n_{k-1}) = p_0 + \cdots + p_{k-1} - n_0 - \cdots - n_{k-1}.$$

(2.18) **Example.** Let A be a finitely generated abelian group[15] and $\mathcal{A}$ be its associated category as done in (2.6). Let $a_1, \ldots, a_k$ be a basis for this group, meaning that every element of A is written $a_1^{n_1} \cdot \ldots \cdot a_k^{n_k}$ for $n_1, \ldots, n_k \in \mathbb{N}$. A usual CRDT $\mathcal{A}^\uparrow$ is the category generated from the semi-lattice $\mathbb{N}^k$ whose joint is done by taking the maximum per component and where a element is smaller than another when every of its component are smaller.

The view functor $\mathcal{M}$ from $\mathcal{A}^\uparrow$ to $\mathcal{A}$ maps $(n_1, \ldots, n_k)$ to $a_1^{n_1} \cdot \ldots \cdot a_k^{n_k}$. This can easily be checked to be a functor.

If we want no concurrent user commits to be merged together, i.e. apply it twice whenever it has been requested to be applied twice, we may instead use the category $(\mathcal{A}^\uparrow)^k$ generated from the semi-lattice $\mathbb{N}^{k \cdot s}$ where s is the number of sites. The user of some site may only modify the copy of $\mathcal{A}^\uparrow$ associated to his or her site. The view functor now maps a state $(n_1^1, \ldots, n_k^1, \ldots, n_1^s, \ldots, n_k^s)$ to $a_1^{n_1^1 \cdot \ldots \cdot n_1^s} \cdot \ldots \cdot a_k^{n_k^1 \cdot \ldots \cdot n_k^s}$.

2.5 Combining CRDT categories

When given one or several CRDTs, with possibly a few other elements, it is possible to combine them to give new CRDTs. These are very useful constructs, as it allows a person to easily build a CRDT from a small set of basic CRDTs.

2.5.1 Cartesian product

The Cartesian product is the easiest combination. Basically it combines two CRDTs into one couple as if we were using them in parallel. Thus we merge the couple by merging each component and we progress in the CRDT couple state if we progress in any of its two component.

Using the construct recursively, we may obtain triples or tuples of any length. The cartesian product is the way to create tuples on the data type level.

(2.19) **Definition.** If $\mathcal{C}$ and $\mathcal{D}$ are two categories, their Cartesian product is the category $\mathcal{C} \times \mathcal{D}$ whose objects are couples (C, D) from the usual Cartesian product of the set of object of each category and whose arrows are also given by the usual Cartesian product :

$$\mathcal{C} \times \mathcal{D}((C, D), (C', D')) = \mathcal{C}(C, C') \times \mathcal{D}(D, D')$$

The composition is defined component-wise :

$$(f_{\mathcal{C}}, f_{\mathcal{D}}) \circ (g_{\mathcal{C}}, g_{\mathcal{D}}) = (f_{\mathcal{C}} \circ g_{\mathcal{C}}, f_{\mathcal{D}} \circ g_{\mathcal{D}})$$

If $\mathcal{C}$ and $\mathcal{D}$ are CRDT categories it can be easily checked that $\mathcal{C} \times \mathcal{D}$ is also a CRDT category.

(2.20) **Example.** Let $\mathcal{C} = \mathcal{Z}_1$ from example 2.17 and $\mathcal{D} = \mathcal{Z}_1$ also. Let $s_{\mathcal{C}}$ and $t_{\mathcal{C}}$ be couple states of $\mathcal{C}$ and let $s_{\mathcal{D}}$ and $t_{\mathcal{D}}$ be states of $\mathcal{D}$ such that there is a single arrow in $\mathcal{C}(s_{\mathcal{C}}, t_{\mathcal{C}})$ and none in $\mathcal{D}(s_{\mathcal{D}}, t_{\mathcal{D}})$. Is there an arrow in $\mathcal{C}\big((s_{\mathcal{C}}, s_{\mathcal{D}}), (t_{\mathcal{C}}, t_{\mathcal{D}})\big)$? No because the standard Cartesian product of a set and an empty set is an empty set.

We can actually see the Cartesian product of a category induced by a poset is the category induced by the Cartesian product of the poset. Thus

$$(s_{\mathcal{C}}, s_{\mathcal{D}}) \leq (t_{\mathcal{C}}, t_{\mathcal{D}}) \quad \text{if and only if } s_{\mathcal{C}} \leq t_{\mathcal{C}} \text{ and } s_{\mathcal{D}} \leq t_{\mathcal{D}}$$

Therefore we showed $\mathcal{Z}_2 = \mathcal{Z}_1 \times \mathcal{Z}_1$. Actually $\mathcal{Z}_k = \mathcal{Z}_1 \times \mathcal{Z}_{k-1}$.

The Cartesian product of two posets always produces a non-total order relation on the product set, even if the two posets have a total order relation.

2.5.2 Lexicographic product

There is another way to define the order on the Cartesian product of two such posets that keeps a total order relation in the end. This order is called the *lexicographic order*, and we call the Cartesian product with such an order the lexicographic product.

The lexical product may be better viewed by using figure 2.3. The lexical product may be though as $\mathcal{C}$ with each object holding its own copy of $\mathcal{D}$. One can move over $\mathcal{D}$ and keep the same object $c \in \mathcal{C}$, or can change of object $\mathcal{C}$ and start over a new usage of $\mathcal{D}$.

(2.21) **Definition.** If $\mathcal{C}$ and $\mathcal{D}$ are two categories, their lexicographic product is the category $\mathcal{C} \ltimes \mathcal{D}$ whose objects are couple (C, D) from the usual Cartesian product of the set of object of each category and whose arrows

$$\mathcal{C} \ltimes \mathcal{D}((C, D), (C', D')) = \begin{cases} \mathcal{C}(C, C') & \text{if } C \neq C', \\ \mathcal{C}(C, C') \times \mathcal{D}(D, D') & \text{if } C = C', \end{cases}$$

The composition is done on the $\mathcal{D}$ component when combining arrows into the same object, else the combination is done on the $\mathcal{C}$ level. Again $f_{\mathcal{C}}$ and $g_{\mathcal{C}}$

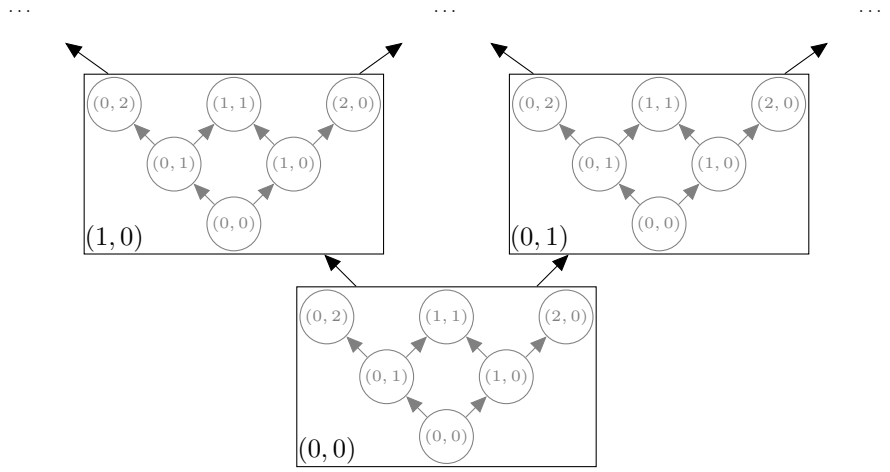


Figure 2.3: The lexicographic product $(\mathbb{N} \times \mathbb{N}) \ltimes (\mathbb{N} \times \mathbb{N})$ of two instances of the category induced from the poset $\mathbb{N} \times \mathbb{N}$. The "embedded" $\mathbb{N} \times \mathbb{N}$'s are using circular state shapes in grey while the value of the "containing" $\mathbb{N} \times \mathbb{N}$ uses black rectangular state shape with the value of the state written at the bottom left of the rectangle.

denote arrows in $\mathcal{C}$ and $f_{\mathcal{D}}$ and $g_{\mathcal{D}}$ denote arrows in $\mathcal{D}$.

$$\begin{aligned} f_{\mathcal{C}} \circ g_{\mathcal{C}} &= f_{\mathcal{C}} \circ g_{\mathcal{C}} \\ f_{\mathcal{C}} \circ (g_{\mathcal{C}}, g_{\mathcal{D}}) &= f_{\mathcal{C}} \circ g_{\mathcal{C}} \\ (f_{\mathcal{C}}, f_{\mathcal{D}}) \circ g_{\mathcal{C}} &= f_{\mathcal{C}} \circ g_{\mathcal{C}} \\ (f_{\mathcal{C}}, f_{\mathcal{D}}) \circ (f_{\mathcal{C}}, g_{\mathcal{D}}) &= (f_{\mathcal{C}}, f_{\mathcal{D}} \circ g_{\mathcal{D}}) \end{aligned}$$

If $\mathcal{C}$ and $\mathcal{D}$ are CRDT categories it can be easily checked that $\mathcal{C} \ltimes \mathcal{D}$ is also a CRDT category.

2.5.3 Lexicographic disjoint union

Another efficient way to combine data types is *disjoint union*. An instance of a disjoint union data type has a state that is either from the first used data type or the second one. The first data type is known as the *left* type and the second one as the *right* type. The lexical product may be better viewed with figure 2.4. The box on the left contains "left" states while the box on the right contains "right" states.

With CRDT, we must be careful to keep an CRDT catogory structure. That is why we can move the state from the *left* type to the *right* type.

(2.22) **Definition.** If $\mathcal{C}$ and $\mathcal{D}$ are two categories, their lexicographic disjoint union is the category $\mathcal{C} \sqcup \mathcal{D}$. Its objects are from the usual disjoint union $\mathcal{C} \sqcup \mathcal{D}$, see example 2.11 for the definition of the disjoint union. Its arrow are given by

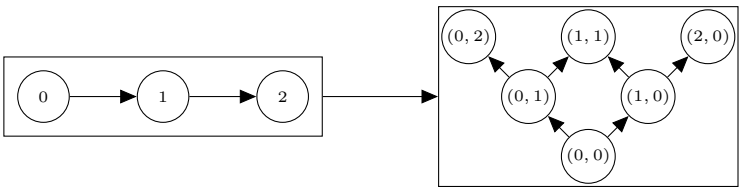


Figure 2.4: The lexicographic disjoint union $\mathbb{N} \sqcup (\mathbb{N} \times \mathbb{N})$ of two instances of the category induced from the poset $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$. The left states are the disks within the left rectangle while the right states are the disks within the right rectangle.

the arrow of $\mathcal{C}$ of $\mathcal{D}$ when using relevant objects or a singleton $\{m_{C,D}\}$ else[16]. Symbol $m_{C,D}$ stands for a "move right" arrow.

$$\mathcal{C} \sqcup_A \mathcal{D}\big((\mathit{left}, C), (\mathit{left}, C')\big) = \mathcal{C}(C, C')$$
$$\mathcal{C} \sqcup_A \mathcal{D}\big((\mathit{right}, D), (\mathit{right}, D')\big) = \mathcal{C}(D, D')$$
$$\mathcal{C} \sqcup_A \mathcal{D}\big((\mathit{left}, C), (\mathit{right}, D)\big) = \{m_{C,D}\}$$

The composition rules are quite straightforward from the above definition. When using only left arrows or only right arrows we use respectively the composition rules of the original left CRDT or the right CRDT. Interesting cases involve arrows of type $m_{C,D}$. Basically we always output the arrow of type $m_{C,D}$ corresponding to the final input and output object of the composition. Thinner behaviour requires the use of more than one arrows between left and right states.

$$f_{D,D'} \circ m_{C,D} = m_{C,D'} m_{C',D} \circ f_{C,C'} = m_{C,D}$$

If $\mathcal{C}$ and $\mathcal{D}$ are CRDT categories it can be easily checked that $\mathcal{C} \sqcup_M \mathcal{D}$ is also a CRDT category. If P and Q are posets, and $\mathcal{P}$ and $\mathcal{Q}$ are the induced CRDT categories from these posets, then $\mathcal{P} \sqcup \mathcal{Q}$ is the induced CRDT category of $P \sqcup Q$, the poset over the disjoint union of the poset whose order is given by

$$(\mathit{left}, p) \leq (\mathit{left}, p') \text{ if } p, p' \in P,$$
$$(\mathit{left}, p) \leq (\mathit{right}, q) \text{ if } p \in P, q \in Q,$$
$$(\mathit{right}, q) \leq (\mathit{right}, q') \text{ if } q, q' \in Q.$$

2.5.4 Other combinations

There are other ways to combine CRDTs that we shall not describe in this thesis because either they are not very useful or it would introduce even further more mathematical notions to describe with category theory. A very interesting case we did not cover is done with the category of functors. If $\mathcal{A}$ and $\mathcal{B}$ are two regular categories, there is a way to produce a category $\mathsf{Fun}(\mathcal{A}, \mathcal{B})$ whose objects are all the possible functors from $\mathcal{A}$ to $\mathcal{B}$. Its arrows are *natural transformations*, entities we did not define. A natural transformation can be

imagined as "nice" mappings between functors with the same input and output category. The category of functors is interesting because if $\mathcal{B}$ is a CRDT category so is $\mathsf{Fun}(\mathcal{A}, \mathcal{B})$. See Adámek et al. [1990] for more details on the category $\mathsf{Fun}(\mathcal{A}, \mathcal{B})$ or any book introducing category theory.

Conclusion about Categories

Categories provides a way to capture data types. The possible state of an instance of the data type are the objects of the categories, and the transition function between these states are the arrows of the category. It is possible to describe conflict-free replicated data types only with property of categories such as the coproduct.

Categories provides a definition of a view of a CRDT with a functor. A functor keeps the information of what transition is possible from what state.

A final very important point is that it is possible to combine categories together to form new categories. This mean create new (CRDT) data types from former (CRDT) data types. The most obvious combination is the *Cartesian* product, but other are possible like the disjoint union, the lexicographic product or the category of functors.

Chapter 3

Concurrent Constraint Programming

Concurrent constraint programming should actually be named *concurrent constraint logic programming*[17]. With this latter denomination, it becomes clear that concurrent constraint programming, or CCP for brevity, mixes three computer paradigms.

Logic programming. Logic programming is a paradigm where a program is a logical predicate. The execution of this program consists to deduce whether there exists an *interpretation* to the logical predicate or not.

A interpretation, or a *model*, is made of a *valuation* and the output *logic structure* of that valuation. By a logic structure, we mean a set together with a fixed set of functions over this set and a fixed set of relations. In logic programming, we do not bother with relations, and use equality as the sole valid relation. By a valuation, we mean a mapping from free variables of the logical predicates to the logic structure and function symbols to the logic structure functions. The mapping should of course recursively evaluate the functions in the output formula. If $\mathcal{M}$ is a model of predicate φ, we write $\mathcal{M} \models \varphi$. If φ has a model, it is said to be *satisfiable*. See the book of Rautenberg [2006] for more details.

This notion brings up another important logic notion, *logical entailment*. A predicate φ entails a predicate ψ if for every model $\mathcal{M}$ such that $\mathcal{M} \models \varphi$ we also have $\mathcal{M} \models \psi$. In other words the valid models of φ is a subset of ψ. Finding if the program predicate is satisfiable is done by inferring new logical rules from former rules. Inference is done with entailment rules.

Logic program obeys the *confluence property* that states that any order of reductions in the program leads to the same result. In logic programming terms this means, that rules can be inferred in any order, the result will always be the same.

Constraint programming. Constraint logic programming adds to logic programming the possibility to tell constraints on variables. This simply means we may add relation symbols to predicates and that a valuation has to map this relation symbol to a logic structure relation. A logic structure relation evaluates to "true" if the relation holds, to "false" otherwise.

Those constraints enable to expression of a wider range of knowledge. Satisfiable constraints are stored in a *constraint store*.

A very nice property of the constraint store is that it is monotonic. Deducing new rules does not affect older ones in any way. This enable to share constraints without synchronization.

Concurrency. This is the nowadays a well-known property that the execution of a program needs not to be sequential or linear. Several threads of execution can virtually work at the same time to advance in the execution progress.

Concurrency raises many issues, including race condition and non-determinism, but logic programming provides a nice framework to use concurrency. As the main structure of a logic program process is its constraint store, and as that store is monotonic, it provides an easy way to share knowledge without the overhead of synchronisation. Thus concurrent logic programming is can possibly used to do *strong eventual consistency*.

3.1 Constraint Logic Programming

In constraint logic programming, computation is done by adding constraints on possible values of variables.

Unlike imperative programming, variables in logic programming are not "read" nor "written". The value they represent is free, but constrained by some rules. Those rules are stored in a set, called the *constraint store*[18]. In logic programming, those rules are described with a *predicate*.

To find a solution, we must find some assignment of free variables to values. If such an assignment is such that every rule of a set is true, we call it a *model* of the set[19]. In other words, a model is a satisfiable assignement of symbols and variables. Some authors use the word "model" to mean assignement but we don't. A correct solution is thus a model of the constraint store. A store without a model is called inconsistent.

When any model of a set S_1 is also a model of set S_2, we say that S_1 entails S_2. The entailment relation is usually written with the symbol $\models$. Since the constraint store is a finite set, entailing each of its predicates is equivalent to entailing the conjunction of those predicates. An inconsistent store is equivalently a store that entails any predicate or yet equivalently a set that entails the predicate "false" (written $\bot$).

In logic programming we thus allow to ask ourselves whether a predicate is entailed by the constraint store. This introduces a new primitive, $ask(c)$, that succeeds when the constraint c is entailed by the constraint store, that fails when its negation $\neg c$ is entailed—because then the store will never entails c in a consistent way—and that suspends its execution until more constraints are added to the store when neither c nor $\neg c$ can be deduced.

The store can grow by adding more constraints. This reduces the number of valid models, refining the set of a valid solution. Let us be careful though : adding a constraint may lead to an *inconsistent* constraint store, which has no models that satisfy it. Thus the new primitive $tell(c)$ adds the constraint c to the store, but only if it does not lead to an inconsistent constraint store. Else its executions fails.

There are several ways to handle the failure. The brutal way is to stop the execution of the whole program. A more common and softer way is to ignore the operation (i.e. leave the store unaffected) and to report the failure. Reporting the failure is commonly done by raising an exception.

Note you can only add a constraint to the store. In other words, the store is *monotonic*. The set of models, satisfiable assignment of symbols and free variables, also decreases. A ruled-out assignement can never be reused later. Nevertheless, if non-determinism is added to the model—by making a random choice when several possibilities are offered—the process could backtrack from when it made such a choice, for example when the process reaches a failure.

In concurrent constraint logic programming, several threads of execution may operate on some of the constraints of the store concurrently. We call a single thread of execution an *agent*. Agents communicate together through the store. A sending agent binds a variable to a message and a receiving agent blocks on that variable because it needs its value but doesn't know it.

Do not confuse agents with sites. The formers are virtual concurrent thread of execution and their execution may be distributed among the latter, physical entities that execute the program.

3.2 The syntax of the CCP' language

We define a theoretical computer language that allows concurrent constraint programming. It behaves similarly to cc, the original concurrent constraint programming language defined in Saraswa and Martin [1990]. We redefine a new language because it will serve as a strong basis for the language of next chapter[20]. A CCP' program is described by CCP' agent declarations. The agents progress by executing *reductions*[21]. Reductions transform the current instructions and the constraint store accordingly to some *reduction rules*. When the instructions are reduced to a logical predicate (the return value), the program is finished.

If several reductions are possible, one is chosen randomly. CCP' obeys the *confluence property*, so reductions may be applied in any order. This property is essential as it allows different sites to execute concurrent threads without bothering to synchronise with other sites to ensure the results are consistent. Since CCP' is low level, at least compared to the language of next chapter, it is possible to write codes that depend on the reduction order : it is up to the programmer to ensure the confluence property is kept. Conversely if no reduction is possible and the agent is not finished, then the agent blocks until some reduction becomes possible. Reductions can be applied concurrently as the name "concurrent constraint programming" suggests.

We provide in this section only the syntax construct of the language, see next sections for semantics.

CCP' may be made of block instructions. Block instructions always ends with a mandatory **end**. Thus a block scope is always clear.

Special block of instructions follows. We described their meaning in next sections. these are

declaration of variable block :	**let**...**st**...**end**,
defintition of function block :	**def**...**by**...**end**,
usage of concurrent threads :	**concurrently do**...**do**... ...**do**...**end**,
blocking conditional block :	**react on**...**do**... ...**on**...**do**...**end**,
non-blocking conditional block :	**react on**...**do**... ...**on**...**do**...**otherwise do**...**end**.

react is a broader equivalent to **ask**. Another valuable instruction is the previously explained **tell**.

A comment begins with # and ends with a newline. Numbers literals are defined as in every programming languages and string literals are enclosed in double quotes.

CCP' provides also a few common unary and binary operators, whose semantics will not be explicated as it is obvious from their name. These operators are : +, −, ∗, /, ^, **mod**, **not**, **and**, **or**, **xor**.

We provide two semantics of this syntax : a centralised one and a distributed one. The distributed semantics maps to the centralised one. Those two semantics are the subject of the remaining sections of the chapter. They are partly inspired from Smolka [1994].

3.3 A centralised semantics of CCP'

In the centralised version we assume a global store is shared between every agent. We do not discuss implementation details of such a shared store and we assume reductions are atomic, as our point of interest dwells in the distributed version of constraint logic programming and not in the centralised one.

In centralised CCP' the store acts as a mean of communication with logical variables as messages between concurrent agents. Agents can communicate by binding a variable to the message value, and agents can wait to receive a message by **ask**ing possible values of such a variable. Agents must coordinate which variable to use : it must be the same one. Any constraint type can be used as a mean of communication, by **tell**ing it and **ask**ing on possible constraint-messages values.

Figure 3.1 lists the possible reductions of the centralised language. In this figure, program texts are often denoted by $\mathcal{T}$ or $\mathcal{S}$, constraint by c, expressions by p and the constraint store is usually σ. An expression is a predicate or a term expression (anything that can be equalised to a variable). A state is a pair of a program text to reduce a constraint store, written with angle brackets[22]. A reduction is a rule that maps a type of pair to another type of pair.

Initially, the text is the full program text and the store is empty.

The two first rules handle instructions composition. If $\mathcal{S}$ and $\mathcal{T}$ are two program texts, then writing sequentially $\mathcal{S}$ and $\mathcal{T}$ means we must first reduce $\mathcal{S}$, then $\mathcal{T}$ once the $\mathcal{S}$ is done (it is reduced to a single expression p).

The next line is a variation of the **ask** condition discussed above. The c_i are constraints, and the $\mathcal{T}_i$ are program codes to perform if one of these constraints is true. Rather than returning *true* or *false* for a condition, **react** branches immediately on an agent associated with a condition that is verified by the store, and blocks otherwise. The next two lines describe a non-blocking version of **react**. The non-blocking version behaviour is highly dependent on the store state. This is not a problem in the case of a sequential program, as the store state at one point in time can be easily deduced, but it may cause

$$
\begin{array}{lll}
\langle\, \mathcal{S}\mathcal{T}\,,\,\sigma\,\rangle & \to \langle\, \mathcal{S}'\mathcal{T}\,,\,\sigma'\,\rangle & \text{if } \langle\, \mathcal{S}\,,\,\sigma\,\rangle \to \langle\, \mathcal{S}'\,,\,\sigma'\,\rangle, \\
\langle\, p\mathcal{T}\,,\,\sigma\,\rangle & \to \langle\, \mathcal{T}\,,\,\sigma\,\rangle, & \text{if } p \text{ is a logical predicate}, \\
\langle\, \textsf{react}\ldots\textsf{on } c_i \textsf{ do } \mathcal{T}_i\ldots\textsf{end}\,,\,\sigma\,\rangle & \to \langle\, \mathcal{T}_i\,,\,\sigma\,\rangle & \text{if } \sigma \models c_i, \\
\langle\, \textsf{react}\ldots\textsf{on } c_i \textsf{ do } \mathcal{T}_i\ldots\textsf{otherwise do } \mathcal{S}\textsf{ end}\,,\,\sigma\,\rangle & \to \langle\, \mathcal{T}_i\,,\,\sigma\,\rangle & \text{if } \sigma \models c_i, \\
\langle\, \textsf{react}\ldots\textsf{on } c_i \textsf{ do } \mathcal{T}_i\ldots\textsf{otherwise do } \mathcal{S}\textsf{ end}\,,\,\sigma\,\rangle & \to \langle\, \mathcal{S}\,,\,\sigma\,\rangle & \text{if } \sigma \not\models c_i \text{ for every } c_i, \\
\langle\, \textsf{tell}\,(c)\,,\,\sigma\,\rangle & \to \langle\, ,\,\sigma \cup \{c\}\,\rangle & \text{if } \sigma \cup \{c\} \not\models \bot, \\
\langle\, \textsf{tell}\,(c)\,,\,\sigma\,\rangle & \to \mathit{failure} & \text{if } \sigma \cup \{c\} \models \bot, \\
\langle\, \textsf{let } x \textsf{ st } \mathcal{T}\textsf{ end}\,,\,\sigma\,\rangle & \to \langle\, \mathcal{T}_x^\xi\,,\,\sigma\,\rangle &
\end{array}
$$

where ξ is a new unique variable name and where $\mathcal{T}_x^\xi$ is $\mathcal{T}$ with each occurrence of x substituted by ξ,

$$
\begin{array}{lll}
\langle\, \textsf{def } p(s) \textsf{ by } \mathcal{T}\textsf{ end}\,,\,\sigma\,\rangle & \to \langle\, ,\,\sigma \cup \{p(s) = \mathcal{T}\}\,\rangle, & \\
\langle\, p(t)\,,\,\sigma \cup \{p(s) = \mathcal{T}\}\,\rangle & \to \langle\, \mathcal{T}_s^t\,,\,\sigma \cup \{p(s) = \mathcal{T}\}\,\rangle &
\end{array}
$$

where $\mathcal{T}_s^t$ is $\mathcal{T}$ with every occurrence of s replaced by t,

$$
\begin{array}{l}
\langle\, \textsf{concurrently do } \mathcal{T}_1\ldots\textsf{do } \mathcal{T}_i\ldots\textsf{do } \mathcal{T}_n\textsf{ end}\,,\,\sigma\,\rangle \\
\qquad \to \langle\, \textsf{concurrently do } \mathcal{T}_1\ldots\textsf{do } \mathcal{T}_i'\ldots\textsf{do } \mathcal{T}_n\textsf{ end}\,,\,\sigma'\,\rangle \quad \text{if } \langle\, \mathcal{T}_i\,,\,\sigma\,\rangle \to \langle\, \mathcal{T}_i'\,,\,\sigma'\,\rangle, \\
\langle\, \textsf{concurrently do } \mathcal{T}_1\ldots\textsf{do } \mathcal{T}_i\ldots\textsf{do } \mathcal{T}_n\textsf{ end}\,,\,\sigma\,\rangle \\
\qquad \to \langle\, \textsf{concurrently do } \mathcal{T}_1\ldots\textsf{do } \mathcal{T}_{i-1}\textsf{ do } \mathcal{T}_{i+1}\ldots\textsf{do } \mathcal{T}_n\textsf{ end}\,,\,\sigma'\,\rangle \quad \text{if } \mathcal{T}_i \text{ is a logical predicate.}
\end{array}
$$

Figure 3.1: The reductions of the cc language

trouble when concurrency is introduced. It is up to the programmer to ensure the program behaves non-hazardously when the program uses this structure.

The two following lines describe **tell** which adds the constraint c to the store σ if this does not lead to an inconsistent store. We do not explain how to handle the failure in the table. As stated in a former section a common and simple and not too destructive way is to keep the store unchanged and to raise an exception.

We did not explain how a constraint is known to be true in the case of **react** or it is added to the store in the case of **tell**. This is because it depends on the inner structure of the constraint store. Keeping the structure of the constraint store unconstrained enables custom and latter optimizations. However, this is a centralised semantics of the language. There is a unique site that has full view of the constraint store. So adding a constraint to it causes no problem.

The next three reductions describe functional abstraction declaration, call, and local variable declaration. The code of a functional abstraction is simply saved into the store. Calling this code with some argument simply replaces the call instruction with the code, with free occurrences of the parameters identifiers in the text replaced by the given arguments.

The two last reductions are the reductions for parallel execution. Reduction of the concurrent agents is equivalent to reducing any of the agents.

Some syntactic sugar can surely be added for readability. For example, the statement **tell** $(x{=}p)$ may be simply written $x{:=}p$. Below is a few other helpful constructs.

ask We can easily implement the ask as described in the beginning of the chapter. When **ask**ing a condition c, we are simply **react**ing on it to return true or on its negation $\neg c$ to return false.

```
def ask(c) by
  react
    on c do true
```

```
    on not(c) do false
  end
end
```

decide We can also implement a decide function that returns true when the store entails or disentails a condition and that returns false otherwise.

```
def decide(c) by
  react
    on c do true
    on not(c) do true
    otherwise do false
  end
end
```

choose Picking a random element from a set is also an easy construct. Remember that if several reductions are possible, we pick up one randomly. Thus if several condition of a **react** block are entailed, by figure 3.1, several agent reduction are possible and pick up one randomly. But the always-true predicate $\top$ is always entailed. Thus if we set every condition to true, they are all entailed, and we must pick randomly a corresponding instruction block to reduce. If each corresponding block simply sends back an element to be chosen, we achieve the desired behaviour. Below is the conversion for a three element choice.

```
def choose(a,b,c) by
  react
    on true do a
    on true do b
    on true do c
  end
end
```

Since we may call choose with a variable number of arguments, choose is actually syntactic sugar generally non-definable in CCP'.

if-then-else This could also be defined as a function.

```
def if(condition, this, that) by
  react
    on condition do this
    on not(condition) do that
  end
end
```

But the function notation is not very readable in this case, so instead we would better use some custom notation.

```
if(condition)
  then this
  else that
end
```

The converse translation can be done by nesting **if**s and using **hesitate** to avoid asking a **if** if it is blocking.

Finally, a set of free variables may be defined using the same variable name and an index over a non-negative integer or some other literal. The index is written after the name within square brackets. Imagine that for every human, we use free variable that records the year of birth of that human. For a particular human, you may call the variable like this : *yearOfBirth*[*someHumanVariable*].

This last construction can have stronger semantics meaning than syntactic sugar, if one allows to use other variables as indexes. This allows to make statement over a bunch of variables.

Similar indexing can be used for truth values associated with the indexed variable. This can effectively be viewed as a relation symbol.

example of centralised CCP' programs

The immutable set

The immutable set is determined by stating what elements belong to it and what elements do not belong to it. A predicate telling whether an element is in the set or not is all that is needed. Its usage is very simple. At the beginning, every elements are in a unknown state as no predicate tells their status. A new relation predicate symbol is added, *element*. The predicate parameters are put between square brackets.

```
def add(e, S) by tell( element[S,e] ) end
def del(e, S) by tell( not(element[S,e]) ) end
def in(e, S) by ask( element[S,e] ) end
```

The immutable list

The immutable list is as straightforward as the immutable set. It is also described little by little. One function predicate symbol, *value* maps a list symbol and a index to its value.

```
def set_value'(L, i, e) by tell( value[L,i]=e ) end
def has_value(L, i, e) by ask( value[L,i]=e ) end
```

Note that due to its structure in the constraint store, a list has an infinite length. A new function predicate symbol *length* can also be used to fix it to a finite length to the list.

```
def new(L, n) by tell( length[L]=n ) end
def set_value(L, i, e) by
  if( i < length[L] )
    then tell( value[L,i] = e )
    else fail
  end
end
```

3.4 A distributed semantics of CCP'

In the distributed version of the CCP' language, we assume each agent has only a partial view over the knowledge of the constraint store. The global constraint store can be defined as the union all partial stores.

3.4.1 Semantics notation

The semantics of the language must be augmented to reflect which agent knows which constraints. We now use a set of states $\{\langle\, \mathcal{T}_s\,,\, \sigma_s\,\rangle\}_{s\in\mathbb{S}}$, where $\mathbb{S}$ is the set of sites. The symbol $\mathcal{T}_s$is a symbol for a (partially reduced) program text. Different sites may have different program texts as they may have used different reductions steps on the text. Each σ_s is the local partial knowledge of the constraint store of each site.

However, if a reduction affects a single site t, we may drop the indexed set notation and write

$$\langle\, \mathcal{T}\,,\, \sigma\,\rangle \;\rightarrow\; \langle\, \mathcal{T}'\,,\, \sigma'\,\rangle \qquad \text{or} \qquad \langle\, \mathcal{T}_t\,,\, \sigma_t\,\rangle \;\rightarrow\; \langle\, \mathcal{T}'_t\,,\, \sigma'_t\,\rangle.$$

This is equivalent to

$$\{\langle\, \mathcal{T}_s\,,\, \sigma_s\,\rangle\}_{s\in\mathbb{S}\setminus\{t\}} \cup \{\langle\, \mathcal{T}_t\,,\, \sigma_t\,\rangle\} \;\rightarrow\; \{\langle\, \mathcal{T}_s\,,\, \sigma_s\,\rangle\}_{s\in\mathbb{S}\setminus\{t\}} \cup \{\langle\, \mathcal{T'}_t\,,\, \sigma'_t\,\rangle\}.$$

With this short notation in mind, one may read again all the reductions of 3.1 and transpose them to the distributed case.

But more reductions are needed. The constraints known by some agents should eventually propagate to other agents. How this propagation occurs is an implementation detail transparent to the programmer. The centralised version and the distributed version of CCP' appears essentially the same to the programmer. One difference, though, is that in distributed CCP', one cannot tell if the store doesn't entails a condition or its negation, as an agent has only a partial view of the store. Rather, the programmer can tell if this partial view entails that condition or its negation. This is a fundamental difference, as the global store can indeed already entail a condition whereas a local view might not be able to entail it. If one is careless, this may lead to an inconsistent global store.

As the constraint store is monotonic, once constraints are told to other agents they cannot be backtracked. Indeed, a backtracking would require *all* agents to backtrack to the state before they received the discarded constraints, leading to heavy synchronisation or logging support that will anyhow significantly hinder performance. Thus the inconsistency of the store of one agent is synonym to the inconsistency of the store of all agents.

If the agent has not told its peers the constraints it has added then if the agent fails it can backtrack to a previous choice and make another one. This can occur because the agent has detected an inconsistency before adding a constraint. This is an example why ignoring the *tell* of a constraint that lead to an inconsistent store and raising an exception is a better solution than storing the constraint first then abruptly stop the execution of all agents.

But this still do not completely avoid inconsistency of the store. Since different sites have different partial views of the store, merging two partial views may lead to inconsistencies. There are two ways to avoid an inconsistency. The

first is to delay transmission until we are perfectly sure they are valid. Unfortunately "perfectly sure" often means synchronization. The second method is to use more complex constraints so that no inconsistency can arise without synchronisation. In other words, build a CRDT with logic predicates.

3.4.2 Reductions

Most reductions of distributed CCP' are translated from centralised CCP' by applying it locally to a site. But we need to specify that sites can tell their peers the constraint they know. The rule is

$$\left\{\left\langle\, \mathcal{T}_s\,,\, \sigma_s \,\right\rangle\right\}_{s\in\mathbb{S}} \;\rightarrow\; \left\{\left\langle\, \mathcal{T}_s\,,\, \sigma_s \,\right\rangle\right\}_{s\in\mathbb{S}\setminus\{t\}} \cup \left\{\left\langle\, \mathcal{T}_t\,,\, \sigma_t \cup \{c_s\} \,\right\rangle \omega_t\right\}$$

if c_s is a constraint in local store σ_s for some $s \in \mathbb{S} \setminus \{t\}$.

Remember that this rule does not prevent an inconsistent store from occurring.

The next chapter propose to add CRDT to the language primitives. In the next subsection there are a few examples of such CRDTs manually translated to logical predicates.

A final remark before these examples are given, though. The distributed CCP' reductions can be mapped to the centralised CCP' reductions, only losing the distributed meaning of the semantics. This enables the programmer to reason in terms of the easier-to-think-about centralised CCP language, and to adapt it to the distributed CCP' with minor and possibly automated changes.

If reductions can be mapped, this means states of execution can also be mapped. The output of the mapping of states is actually the result in the global store, a.k.a. the union of the knowledge of each agent.

To the set of local states $\left\{\left\langle\, \mathcal{T}_s\,,\, \sigma_s \,\right\rangle\right\}_{s\in\mathbb{S}}$ is mapped to

$$\left\langle\, \mathcal{T}_{\mathbb{S}}\,,\, \bigcup_{s\in\mathbb{S}} \sigma_s \,\right\rangle,$$

where $\mathcal{T}_{\mathbb{S}}$ is the partially reduced program text with "all applied reduction to some agent applied to this program text". We mean by that that $\mathcal{T}_{\mathbb{S}}$ is a partially reduced program text that is a reduced form of $\mathcal{T}_s$ for every $s \in \mathbb{S}$. Furthermore it is and is the least-reduced partially reduced program verifying this property. In the next chapter we shall see the making of this partially reduced program text of a CRDT.

3.4.3 Version number

As conflicts may arise we should carefully design our types to avoid them. By resolving conflicts we also get for free the ability for some user to correct a wrong manipulation. In these examples, we shall use version numbers of the element of two containers. We should have a way to deduce the next version number of an element from the store.

Assume version of element e is recorded as an indexed variable $\mathit{version}[e]$. When we progress in the modifications of the variable, we know that the final version of the variable will be greater than the current stored version, say n. Thus the store entails $\mathit{version}[e] >= n$. When we reach an integer $n1$ such that $\mathit{version}[e] >= n1$ is not entailed, $n1$ is the next version number to be used.

```
def nextVersionNumber(e) by
  nextVersionNumber'(e,0)
end

def nextVersionNumber'(e,n) by
  react
    #if version n is known to be reached, recurse
    on version[e] >= n do nextVersionNumber'(e,n+1)
    otherwise do
      react
        #we a have a maximal value for version number
        #that prevent to have a next version number.
        on version[e] == n do fail
        #we cannot tell wheter version[e] is greater,
        #equal or less than n.
        otherwise do n
      end
  end
end

def currentVersionNumber(e) by
  nextVersionNumber(e) - 1
end
```

3.4.4 Examples of distributed CCP' programs

We provide now a few examples of CRDT written in CCP'. Some of these CRDTs will be used as primitive in the language of next chapter.

Modification-wins versioned set

The modification-wins versioned set is an editable set. Imagine we have a set that keeps track of the current citizens of a country. Then imagine Lucy, a citizen of that country, lost her nationality. She has to be removed from the set. Thus we add a contradiction by stating that Lucy is not in the set, where the constraint store contains that Lucy is in the set.

Of course there are way to circumvent that. A quick workaround is to give a version $version[(S,e)]$ to each element e of the set S. Thus when we say Lucy is in the set, it has version 1. When she lost her nationality and leaves the set, this information is version 2. If she ever manages to get her nationality back and reintegrate the set, the predicate is version 3.

This is enough for centralised CCP', but this isn't enough for distributed CCP'. Now imagine Lucy can apply to regain her former nationality, but she has to do it within the ten years that follows her lost of nationality. Let us say Lucy apply the very last day of these 10 years and her application form is accepted in the ten last minutes before the civic office clauses that day, so she is added to the set. This add a information with the next version number that Lucy is part of the set. Meanwhile, another civic employee notices it is the last day for Lucy to apply and he has not seen Lucy's form. He genuinely believes it is highly unlikely a form will be received in the next ten minutes and to leave

work ealier, register Lucy as definitely not in the set. This add a information with the same next version number that Lucy is not part of the set. We thus have two concurrent conflicting information bearing the same version number. We do still have to fix the store !

It is important to not that Lucy has only *two* possible viewed states : "in the set" or "out of the set". When adding or removing element e modifies the viewed state of e, $version[(S,e)]$ is incremented and the predicate of the state of element at the new version is set. The version number is not modified if the operation does not change the viewed state of e. If two users modify an element, either they both modify it the same way (both remove or both add the element) in which case no conflict arises, either only one has requested change that affects the viewed state of the element, in which case the modification that also modifies the viewed state will propagate and take precedence.

```
#test wheter e is in set S
def set_contains(S, e) by
  react
    on set_element[ S, e, currentVersionNumber((S,e)) ]
       do true
    otherwise do false
  end
end

#adds e in set S
def set_add(S, e) by
  if( not set_contains(S, e) )
  then
    tell(    version[(S,e)] >= nextVersionNumber((S,e))
       and
        set_element[ S, e, nextVersionNumber((S,e)) ]
           )
  end
end

#remove e from set S
def set_delete(S, e) by
  if( set_contains(S, e)
  then
    tell(    version[(S,e)] >= nextVersionNumber((S,e))
       and
        not set_element[ S, e, nextVersionNumber((S,e))
           ]    )
  end
end
```

Observe this "logic-programming" set is a CRDT as described in chapter 1. The above operations are the few commit operations available to the CRDT set. The merge of the CRDT is given for free, as it corresponds to the merge of two constraint stores. We proved at the beginning of this example that the merge of two stores with different constraints for the modification-win versioned set gives a consistent store, the CRDT merge is well-defined.

The views simply test for the latest known version of an element whether is belongs to the set or not. We formally have a view per possible element given as parameter. Note as a side-effect of our design, versions of element *e* are alternatively positive and negative. Predicate *element*[*S*, *e*, $2n$] is followed by **not** *element*[*S*, *e*, $2n+1$].

Modification-wins versioned graph

The notion of the set can be generalised to produce a graph. A graph is made of a set of vertices and for each couple of vertices it is also made of a set of edges. The set of vertices can be modelled by the versioned set described in the previous section. Each edge can be represented by a boolean (a truth value) indexed by the vertices and the version of the vertices it links to.

But care must be taken. If we remove a vertex, all the still connected edges attached to it should be removed. This can be easily done by using a set not between two vertices but between two *versions* of vertices. Thus when a vertex is removed, it accesses to new edge sets, which are empty by default. On the other hand, if we add an already present vertex, we should not remove the edges attached to it. Since we do not augment the version number in such a case, this is automatically verified. Thus the graph has also a "modification wins" policy.

```
#direct mapping to standard set functions
def vertex_contains(G, v) by
    set_contains(G[vertexes],v) end
def vertex_add(S, e) by
    set_add(G[vertexes],v) end
def vertex_delete(S, e) by
    set_delete(G[vertexes],v) end

# helper function to use _versioned_ vertices
def vertex_current(v) by
    (v, currentVersionNumber(v)) end

# test wheter a oriented edge between v1 and v2 exists
def edge_contains(G, (v1,v2)) by
  let edge_set_identifier st
    tell(  edge_set_identifier=
        G[edges][vertex_current(v1)][vertex_current(v2)]
         )
    react
      on graph_edge[edge_set_identifier,
          currentVersionNumber(edge_set_identifier)]   )
          do true
      otherwise do false
    end
  end
end

# add an edge
def edge_add(G, (v1,v2)) by
```

```
  # if the vertice do not already exists,
  # add them
  vertex_add(G, v1)
  vertex_add(G, v2)
  if( not edge_contains(S, (v1,v2)) )
  then
    let edge_set_identifier st
      tell(  edge_set_identifier =
          G[edges][vertex_current(v1)][vertex_current(v2)]
           )
      tell(  version[edge_set_identifier] >=
          nextVersionNumber(edge_set_identifier)
           and graph_edge[edge_set_identifier,
              nextVersionNumber(edge_set_identifier)]  )
    end
  end
end

# remove an edge
def edge_delete(G, (v1,v2)) by
  if( edge_contains(S, (v1,v2)) )
  then
    let edge_set_identifier st
      tell(  edge_set_identifier=
          G[edges][vertex_current(v1)][vertex_current(v2)]
           )
      tell(  not graph_edge[edge_set_identifier,
          nextVersionNumber(edge_set_identifier)]  )
    end
  end
end
```

Versioned cell

The versioned cell is a mutable container for any content. It could be viewed as a mutable pointer to a logical term.

The easiest way to convert it to a CRDT, written in the distributed CCP' language is to record every of its assigned value for each site and each version, and to have rule to retreive the "good" assignement when requesting the value of the cell. This rule must be deterministic : it always outputs the same result on same input. Thus we use one variable $cell[c,s,n]$ per site s and per version number n.

We consider we may get the local site identifier through the procedure $siteid()$ and know sites with $site()$.

```
#change the value of the cell for this site s
def cell_set(cell,val) by
  let (s, n) st
    tell( s = siteid() )
    tell( n = nextVersionNumber(cellVersion[cell,s]) )
```

```
    tell( cellVersion[cell,s] >= n )
    tell( cell[cell,s,n] = val )
  end
end


# get the value of the cell by taking the maximum value
def cell_get(cell,val) by
  # * map is the standard functional function that apply
  #   a function to every element of a list and returns
  #   a list of the results.
  # * maximum return the maximum element of the
  #   collection given as first argument for the order
  #   given as second argument.
  maximum( map(sites(),cell_get'), fst_lt )
end

#compare the first component of two couples
def fst_lt(x,y) by
  let x_fst x_snd y_fst y_snd in
    tell( (x_fst, x_snd) = x )
    tell( (y_fst, y_snd) = x )
    x_fst <= y_fst
  end
end

# get the version and value of the cell for site 'site'
def cell_get'(cell,val,site) by
  let ncur value st
    tell( ncur =
       currentVersionNumber(cellVersion(cell,site)) )
    tell( value = cell[cell,site,ncur] #create an alias
       name )
    (ncur, value)
  end
end
```

Chapter 4

Combining CRDT and CCP

We will now explain a way to combine *conflict-free replicated data types* of chapters 1 and 2 with *concurrent constraint programming* of chapter 3. The aim to do so is to be able to code a strongly eventual consistent software in a network-transparent way—which is an easier way to reason with. The result would be a program that can be distributed over a large network with minimal cost in terms of slowness and unreliability, as we explained in the first section.

In chapter 1, we briefly explained what a CRDT is and why it brings *strong eventual consistency* through *monotonicity*. Remember monotonicity means an instance cannot go back to a former state. This property of strong eventual consistency was critical to build asynchronous programs while retaining availability and partition tolerance. We shall see at the beginning of the third section of this chapter that the components of a process shared among many sites can be monotonic, and thus can be CRDTs.

Using asynchronous data sharing is not without its own problems, though. In the second section, we discussed a particular problem, *variable declaration*, and how to handle the concurrent declarations of the same variable. This will add a new component to the tuple of shared components, the *symbol table*.

Finally, in the third section, after briefly explaining why the shared components of a process are CRDT, we define a new language based on CCP', *Roze* and provide a few reductions to proceed the computation as we did for CCP' in chapter 3.

4.1 Behaviour of a generated CRDT

Network communication is transparent to the programmer. Thanks to CRDTs we know synchronization is not needed to create a distributed program. Thus we can parallelise computations without having to wait slow and expensive network operations to complete.

Actually if all sites try to reduce the same initial program text we do not have to rely on network operations at all. Every replica can execute on each site without a single message from other sites. But without any communication, the whole computation has to be repeated on each replica. The sole gain to do that is to recover from an unexpected failure of a site.

Since we try to create a network-transparent language to the programmer, this latter person will not be able to specify different behaviour on different

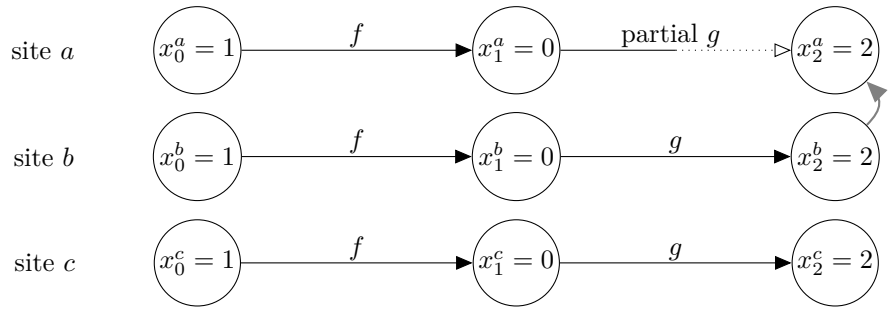


Figure 4.1: The progress of computation

sites. In other words from now on we shall assume every site begins with the same initial program text. They can of course reduce it differently.

Optimization can be made. We would like to be able to send messages between the result of computations to avoid redundant execution of the same expensive execution steps. Firstly, if we add the possibility to share asynchronously the store's constraints, then each site can take advantage of the knowledge learnt from other sites. Thus the programmer should assume the store contains information it is unaware of, and he or she should take advantage of that when writing the program text. Nevertheless, we must not rely on the network operations as they are slow and unreliable, so these messages must be unsynchronised and unexpected. Taking advantage of these properties, the messages could dynamically be chosen to be sent, depending on the difficulty of the computation to obtain a result, and the network overload and the assumed speed benefit from the operation. When to send a message is a complicated issue we do not discuss here. We shall simply assume any site may send a message to one of its peer at any time.

But sharing only constraints is not always so much beneficial. Sometimes it is the program text itself that is difficult to reduce, i.e. advance its conversion of instructions. In such a case, another optimization could be done by sharing the applied reductions of the program text. Figure 4.1 represent three sites computing three value, x_0, x_1 and x_c by applying a function, f or g, on the previous result. All three sites can do the computation independently, as c does, but a site, like b, can send a message to another site, like a, with the result of a computation if it believes that receiving the message is faster than doing the actual computations. Thus a does not need to reduce g furthermore.

The language we define here, based on CCP', is network-transparent to the programmer. Those network communications are done asynchronously. Thus the state of the processes of each site should evolve asynchronously. Remember that the network is not needed and each site may do the whole computation alone. We thus remove features than enable the programmer to behave unexpectedly and asynchronously or to rely on the network—such as mutable containers—and replace most of them with monotonic structures defined over constraint in the store, CRDTs.

4.2 Distributed procedures and identifiers

Entities used in Roze are immutable or monotonically mutable. It is the constraints about them that evolves and also the constraint store. This means functions have no special behaviour to handle about the data type

But there is another issue with procedures. To each variable name must correspond a *logical variable*, used in the constraint store. We shall now use *variable* for logical variable and *identifier* for variable name. Procedures must be able to correctly assign a variable to each identifier it meets.

The same identifier used in different scopes may refer to different variables. In a centralised language, the current variable referred by a name can effectively be remembered by a symbol table hierarchy. When a new variable is declared with **let**...**st**...**end**, we should use a *fresh variable*[23]. A fresh variable is a variable unused in the constraint store.

4.2.1 Two issues with distributed identifiers

Unfortunately the centralised symbol table is not possible in a distributed model, because a site has only a partial knowledge on what variable has been declared. By the non-synchronization principle, we allow sites to declare variables concurrently. There are two issues to tackle.

The first is to avoid *variable collisions*. It means we have to avoid that two unrelated identifiers on concurrent sites refer to the same variable where they should refer to two different ones. In other words we must ensure that the representation of each variable is *globally unique* over the network.

The second, the converse of the first, is to avoid *declaration partition*, where a variable declaration intended to be the same on two sites use a different logical variable on each site. This second problem is an issue raised by the fact that we use *strong eventual consistency* and not *eventual consistency*. This form of consistency disallows any synchronisation to occur. Therefore when a variable is declared by some site, it cannot be sure other sites are aware of the declaration before they redeclare the variable themselves. Thus they may use another variable representation for the same variable. We must be able to identify those two apparently different variable, i.e. we must do *variable identification*.

A very easy way to avoid the former issue is to use one distinct variable set per site. A site is allowed to only declare a variable in its own variable set. To tackle the latter issue, we can add constraints that identifies distinct variables from different sites intended to be the same. A set of variables from different sites that are together identified for such reasons is called an *equivalence class*[24]. But how to compute the complete and correct equivalence class of each variable ?

4.2.2 Distributed symbol table

The question of how to compute an equivalence class is subtly tied to the question of knowing what variable an identifier refers to. Indeed if we look at an identifier in a global way (not localised to a site) and at a variable in a distributed way (localised to a site), then to a global identifier corresponds the set of the corresponding localised variable from each site. In other words, the

variable pointed by a fixed identifier on any site is valid, thus all such variable for this fixed identifier must be part of the same equivalence class.

The common structure to select the variable from identifier is called a *symbol table*. It is a map from identifiers to their corresponding equivalence class. The map is actually a hierarchy of maps, meaning that the scope of an identifier may be limited to a small portion of a program. Furthermore an identifier in some scope may very well override another identifier from one of its ancestor's scope. We want to build the global map from identifiers to variables. At least we want a symbol table large enough to be able to recognise the identifier associated with a variable from other site and used in the constraint store. The symbol table thus make sense to be a CRDT.

The symbol table is not a static component. This is important because we can declare a variable in a function and call this function several times. At different calls, these same identifiers in the same call hierarchy, must be bound to different variables.

This produces yet another issue. If we have to merge to a symbol table, how to ensure that the same functional scopes correspond to the same function calls ? The obvious solution is to differentiate the calls. If functions are *pure*, i.e. the function always gives the same output on the same input, then we may declare a function scope per choice of possible input variables. Closure variables are induced by the parent's scope. However, if we call twice the same function with the same arguments but from different scopes two new scopes have to be created. The hierarchy is a *tree*

Scopes are associated with a procedure, a **let**...**st**...**end** structure or the global scope. Procedures introduce argument identifiers and **let** scopes introduce new unbound identifiers. Such a program text is bound to a *variable* when defined, so each scope is annotated with this variable. Those variables bound to a program are also site-specific, and the variables from different sites have to be identified. This causes no problem for named procedures, as the identifier of the procedure will link to the equivalence class. But anonymous procedures, and scope declarations using **let** have no given identifier.

The quick workaround is to generate an identifier for anonymous procedures in advance—for example during compilation time. Generating identifiers at compile-time is acceptable, but selecting what variable is bound to what identifier at compile time is not. Variable cannot be generated at compile-time because the same portion of code may be executed with different variables. But identifiers of the portion of code remains the same in each execution of it, so generating a identifier for this portion of code is acceptable. What variable is assigned to each such portion of code, is done at run-time, though.

We can also factor scopes together, for example, when a **let** scope is the sole element of a procedure. The scope of the procedure and the **let** structure can be merge in a single map.

If the function is not pure then a least a variable in its closure or in its input is mutable. Hopefully we use a monotonic language, which means that the incriminated variables, to be shared with other sites, must be CRDTs. Thus we can take a *snapshot* of the state of the variable. If we consider different snapshots of the same variable are different inputs, then the function is now pure.

Combining the scope is done identifier-wise. In a scope map, if an identifier is unique to a site, it is added in the merged symbol table. Else, the new set of

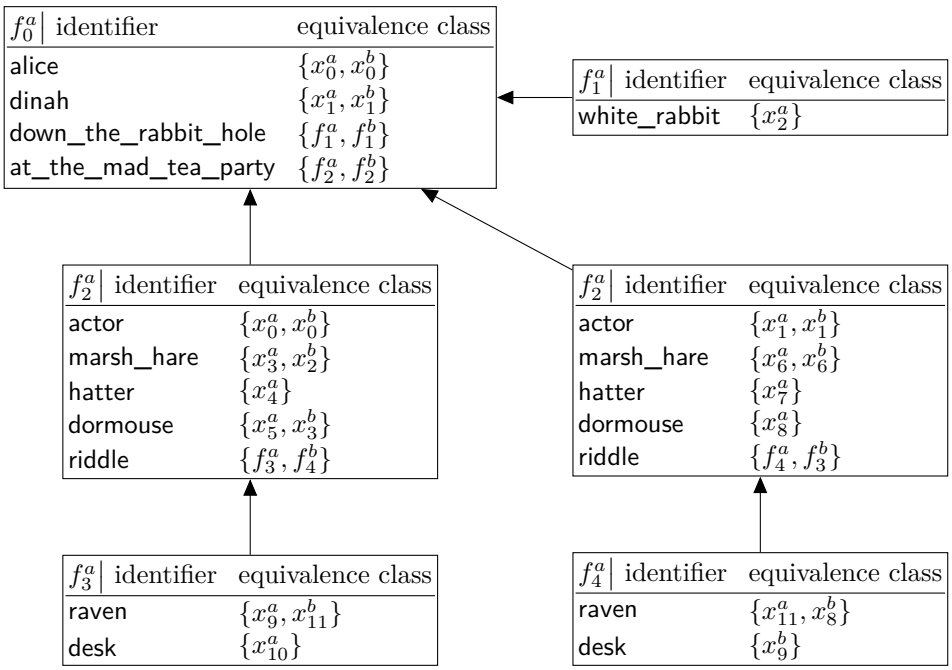


Figure 4.2: An example of a CRDT symbol table.

variables is the union of each set of variables from each original entry. If two variables bound to a function, a procedure, or **let** structures are identified in the process, we recur to merge the scope maps of each of these variables.

Let us sum up how the symbol table is made. The symbol table is a *rooted tree*. A node correspond to the scope of one or several variables in the program text. The node is made of a map from symbols to the set of variable, one per site, that the symbol is referring to. This set is called an equivalence class. The parent of the node is the smallest closure scope embedding the scope of the node. Other ancestors of the node are other closure scopes. Merging maps is done entry-wise, by taking the union of the equivalence classes. If two variables bound to procedures are merged, their scope map is recursively merged.

An example of such a symbol table over two sites, a and b, is given in figure 4.2. Each variable is linked with an equivalence class. In the example, there are three functions. down_the_rabbit_hole is called once, at_the_mad_tea_party is called twice which in turns call once `riddle`. We also combine procedure scopes with the **let** structure that composed their entire body.

The local site must of course keep track of the correct scope being used. It may just use a stack frame with each frame pointing to the correct scope.

4.3 Roze, a monotonic and distributed language

Figure 4.3 shows the kernel instructions of Oz. Oz is another programming language built over the notion of confluence of execution. Thus the same Oz program can use threads and be executed on two different machines to give the same output (if one do not use non-determinism like a cell's state). Oz

```
S ::= S S
    | X= f(l1:Y1, ... ,ln:Yn)
    | X= <number> | X = <atom> | {Newname X}
    | local X1 ... Xn in S end | X= Y
    | proc { X Y1 ... Yn } in S end | {X Y1 ... Yn}
    | {NewCell X Y} | {Exchange X Y Z} | {Access X Y}
    | if X then S else S end
    | thread S end | {GetThreadId X}
    | try S catch X then S end | raise X end
```

Figure 4.3: Kernel features of the Oz language from Haridi et al. [1999].

uses mainly *logical variable* as computer variables[25] but it allows the use of non-monotonic mutable data types, cells. The state of a cell is highly non-deterministic as any thread of execution can modify it at any moment. This enforces some form of synchronisation upon cells, to ensure different sites use the correct state of the cells. Therefore we cannot support cells in their Oz behaviour with a strongly eventual consistent program.

Inspired by CCP' and Oz, we present the language Roze (we couldn't find an elegant name involving "CCP"), which is complete and monotonic. This paper argues that such synchronisation is unnecessary. We firstly show how some Oz primitive are easy to distribute over an asynchronously network and can be reused in Roze. Then we show the syntax of Roze and finally we present its semantics, by the possible reductions it can do.

4.3.1 Program CRDTs

At this network-aware layer, a process on a site is made of a few components, three of which are shared CRDTs, the others are site-specific, such as the stack frame. There is a *constraint store* that records the assignment or other constraints about logical variables. There is a *hierarchical symbol table* that binds identifiers to a set of variables they denote. Finally there is the *"reduced" program text* that corresponds to a partially reduced state of the original program text. All of these components are CRDTs according to the definition of chapter 1 and 2. We shall briefly remind why.

Constraint store CRDT

The constraint store is a conjunction of immutable constraints. It can be imagined as the *set* of those constraints. As we can *only add* constraints, this set is growing. Thus it is a CRDT by taking the union of sets for the merge. Commits add new constraints to the set and the merge of two sets is their mathematical union.

Hierachical symbol table CRDT

The hierarchical symbol table is a tree of maps from names to a set of variables, the equivalence class. Each of these map is a CRDT, by mapping a symbol θ to the union $s \cup s'$, where s and s' are the variable sets associated with θ in each

of the initial map—whether it is the set of variables or the set of tuples. If two variables bound to functions are identified, then the scope of those functions must recursively be merged. So the whole tree is also a CRDT.

Partially reduced program text CRDT

The reduced program text may only differ in what reductions have been applied. We search the joint of two partially-reduced programs by reducing both texts until they are equals, neglecting the side-effect on the constraint store. The common text obtained by a minimal number of reductions is the joint of the two reduced program texts. The fact that reduced program texts form a CRDT is exactly the confluence property. Reducing the program gives the same result, independently from how the order of possible reduction are chosen. As we outlined in the introduction, this is a very nice property that ensures we can use threads in process without having to synchronise different replicas.

Merging the partially reduced program texts might not be very efficient, so we may just instead broadcast to peer sites what reductions have been done. Anyway, sites should be able to advance in the computation alone, so they can ignore reductions performed messages if they believe it is easier to handle and only let the store and the symbol table to be merged.

4.3.2 Primitive values

The syntax of Roze is mainly inspired from the syntax of CCP'. We pick up from Oz the compiler-enforced convention that variable identifiers begin with a capital letter and atom identifiers begin with a small letter.

In figure 4.3, we see there are five basic types assignable to a logic variable : *numbers*, *atoms*, *records* (a named tuple, very similar to a C structure), *names* and *cells*. Cells are discarded in CCP' and Roze. We shall briefly discuss how other primitive types are handled in a asynchronously distributed system.

We already spoke of primitive values. Here is how they are syntactically defined

Numbers Numbers are immutable constants, either integers or floating-point values. As they do not vary, no special representation is needed to represent them in the store. $x_3^a = 1$ is a valid constraint.

If the same variable must be equal to two distinct numbers, there is no way to resolve the conflict, and the store is inconsistent. This should not happen.

Numbers in Roze share the same syntax as numbers in Oz. Example of valid number are `1` for 1, `~2` for -2 and `1.5` for $\frac{3}{2}$.

Standard binary operators for numbers are provided : the sum `+`, the difference $-$, the product $*$, the division `/`, the modulo **`mod`** and the exponentiation $\widehat{}$.

Booleans Booleans are not present in the kernel Oz language, but exist in the full version. *`true`* and *`false`* are the two boolean values, they can be viewed as atoms. A predicate is a not fully reduced expression that once reduced produces a Boolean.

More interesting about them is that they can be combined with built-in binary and unary operators **and**, **or**, **not** and **xor**.

Atoms *Atoms* are identifier constants converted to integers at compile-time. Thus their final representation should be handled like numbers.

The identifier of an atom begins with an uncapitalised letter. Examples of atoms are *ok* and *north_east*. An example of variable identifier is *Area*.

Tuples *Tuples* are borrowed from CCP'. A tuple is a fixed-width list of variables. An example of tuple is (1,0).

When writing X=(1,0), we use three variables. Firstly, there is the variable x referred by identifier X. But x is bound to a tuple of two variables s and t. s is bound to 1 and t to 0.

Records *Records* are also borrowed from Oz. They correspond to labelled tuples, with possibly a labelled index to each components. They are somewhat an equivalent to C "structs".

Their syntax is fairly straightforward : use a prefix atom followed by a tuple. Entries in tuples may moreover be prefixed by a number or another atom together with a colon. An example of record is *character*(*sex*:*female*, *name*:"*Alice*␣*Liddell*", *age*:*Y*).

Fields can be accessed with the dot notation. For example, if *Alice* is the identifier of a variable bound to the previous record, then *Alice*.*name* is the identifier of a variable bound to "*Alice*␣*Liddell*".

As for previous types, merging two records is only possible if they have the same labels, the same indexes (in same quantity and order) and the two variables of each index can also be merged together. Consider for example, we have two sites a and b. We want to merge records *circle*(*x*:0, *y*:y^a, *radius*:r^a) and *circle*(*x*:x^b, *y*:y^b, *radius*:1), where y^a, r^a, x^b and y^b denote unconstrained assignments. The merged record is therefore *circle*(*x*:0, *y*:y^a, *radius*:1) with the constraint $y^a = y^b$.

Names Oz also provides a feature called *name*. A name is a unique identifier generated at run-time. Names are introduced mainly to provide some form of synchronization. Therefore, we drop names out of Roze, as synchronization is exactly what we wanted to avoid.

However it is possible to create asynchronously distributed names. Adjective "unique" means that names are unique in the layer that do not see the network operations, not unique at the network layer. Thus the same problem arises as with identifiers given in a program. The issue is discussed in a previous section. Briefly summarised, each site generates its own unique name values. Then values from different sites that should have the same name are identified. The identification is known thanks to an *equivalence class*, a set of identified values. Be aware that does not mean that telling any two variables are equal leads to equality of the names they may be bound to. Solely when the variable are identified because they represent the same variable.

How to find equivalence classes is a domain-specific issue. In the case of identifiers, it was done through a *symbol table*. We do not provide a

solution of how to compute the equivalence class of a name because we do not seek one, since we drop names from the language.

4.3.3 Reductions

The semantic of Roze is specified by reductions. Those reductions take into account the multiplicity of sites and stores on the site, and the sharing of the symbol table. By virtue of the convergence property, these reductions can be applied in any order to give the same result. In the description of the reductions, $\mathcal{S}$, $\mathcal{T}$ and $\mathcal{U}$ denote program texts, e denotes an expression, typically a primitive value or a predicate or a variable (do not confuse with statements of the program text we reduce below), c denotes a predicate also known as a constraint, x or ξ denote a variable, σ denotes the constraint store, ω denotes the symbol table and $\mathbb{S}$ is the set of sites.

On each site, the computation state is now made of triples $(\mathcal{T}, \sigma, \omega)$, where $\mathcal{T}$ is the partially reduced program text, σ is the constraint store and ω is the symbol table. We do not formally specify how site-specific computation progress components behave. Since they are site-specific, custom optimization of this component may differ as long as the reduction of the shared components remains the same. The global model is the set of those triples indexed by the sites : $\left\{\left\langle \mathcal{T}_s \,,\, \sigma_s \,,\, \omega_s \right\rangle\right\}_{s\in\mathbb{S}}$, where $\mathbb{S}$ is the set of sites.

Reductions are still written with an arrow

$$\left\{\left\langle \mathcal{T}_s \,,\, \sigma_s \,,\, \omega_s \right\rangle\right\}_{s\in\mathbb{S}} \rightarrow \left\{\left\langle \mathcal{T}'_s \,,\, \sigma'_s \,,\, \omega'_s \right\rangle\right\}_{s\in\mathbb{S}}$$

However, if a reduction affects a single site t, we may still drop the indexed set notation and write

$$\left\langle \mathcal{T} \,,\, \sigma \,,\, \omega \right\rangle \rightarrow \left\langle \mathcal{T}' \,,\, \sigma' \,,\, \omega' \right\rangle$$

to mean that only site t for some $t \in \mathbb{S}$ with state $\left\langle \mathcal{T} \,,\, \sigma \,,\, \omega \right\rangle$ is modified to state $\left\langle \mathcal{T}' \,,\, \sigma' \,,\, \omega' \right\rangle$.

We introduce a new notation to denote operations on the symbol table. $\omega \oplus (\mathbf{s}, e, V)$ tells us we *add* a symbol entry in the scope $\mathbf{s}$ with identifier e and variable set V, or if an entry with identifier e already exists we update the previous entry $(e, \bar{V})$ to $(e, \bar{V} \cup V)$. We also write $\omega \lhd (\mathbf{s}, f)$ to tell us we create a new scope from text in variable f whose parent is $\mathbf{s}$. The symbol $\mathbf{s}_0^\omega$ is a short cut to write the current scope (it is thus dependant of the stack frame status). $\bar{\mathbf{s}}$ is the merged map from $\mathbf{s}$ and its ancestors, entries duplicated in "youngest" tables overriding the entry in the ancestors. Thus $\overline{\mathbf{s}_0^\omega}$ is the current identifier visibility.

We informally extend the Oz language with symbols solely used in reduction. We allow any variable to be used as a Oz symbol. A standard reduction is thus converting an identifier symbol to a variable symbol. We also add an instruction $\diamond$, which specifies the end of the scope, or when the stack frame should pop out a scope.

Sharing constraints. Of course, as in CCP', any constraint known to a site can be shared by this site to one of its peers. In a full mesh network topology, this gives the following reductions.

$$\left\{\left\langle \mathcal{T}_s \,,\, \sigma_s \,,\, \omega_s \right\rangle\right\}_{s\in\mathbb{S}} \rightarrow \left\{\left\langle \mathcal{T}_s \,,\, \sigma_s \,,\, \omega_s \right\rangle\right\}_{s\in\mathbb{S}\setminus\{t\}} \cup \left\{\left\langle \mathcal{T}_t \,,\, \sigma_t \cup \{c_s\} \,,\, \omega_t \right\rangle\right\}$$

if $c_s \in \sigma_s$ for some $s \in \mathbb{S} \setminus \{t\}$.

Sharing symbol table entries. Constraints told to a site by one of its peers may be irrelevant if the peer has used different variables for its constraint. Thus correctly merging the symbol table to know what variable refers to the same entities is important. The merge of symbol table is what the following reduction proposes.

$$\left\{\left\langle\, \mathcal{T}_s \,,\, \sigma_s \,,\, \omega_s \,\right\rangle\right\}_{s\in\mathbb{S}} \to \left\{\left\langle\, \mathcal{T}_s \,,\, \sigma_s \,,\, \omega_s \,\right\rangle\right\}_{s\in\mathbb{S}\setminus\{t\}} \cup \left\{\left\langle\, \mathcal{T}_t \,,\, \sigma_t \,,\, \omega_t \oplus (\mathbf{s}, e, V) \,\right\rangle\right\}$$

if $t \in \mathbb{S}$ and (e, V) is an entry in scope $\mathbf{s}$ of ω_s for some $s \in \mathbb{S} \setminus \{t\}$.

Identifying variables from an equivalence class. The role of the symbol table, as explained for previous reductions, is to identify different variables from different peers that should actually have been the same. This is what proposes the next reduction.

$$\left\langle\, \mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \to \left\langle\, \mathcal{T} \,,\, \sigma \cup \{x = y\} \,,\, \omega \,\right\rangle$$

if there is an entry (e, V) in ω such that $x, y \in V$.

Equivalence classes do not overlap. The final set of equivalence classes should form a partition of used variables. Thus if a variable is in two equivalence classes, it is because the two classes are the same. Thus variables from both sets should be identified together[26].

$$\begin{aligned}&\left\langle\, \mathcal{T} \,,\, \sigma \oplus (\mathbf{s}, e, V) \oplus (\mathbf{s}', e', V') \,,\, \omega \,\right\rangle \\ &\quad \to \left\langle\, \mathcal{T} \,,\, \sigma \oplus (\mathbf{s}, e, V \cup V') \oplus (\mathbf{s}', e', V \cup V') \,,\, \omega \,\right\rangle \text{ if } V \cap V' \neq \{\}.\end{aligned}$$

Identifier to logical variable. Of course the main purpose of the symbol table is to map identifiers to variables. This what the following rule does.

$$\left\langle\, X \,,\, \sigma \,,\, \omega \,\right\rangle \to \left\langle\, \xi \,,\, \sigma \,,\, \omega \,\right\rangle$$

where X is an identifier and an entry (X, S) exists in $\overline{\mathbf{s}_0^\omega}$ with $\xi \in S$.

Sequential composition. Let $\mathcal{S}$ and $\mathcal{T}$ be two (partially reduced) program texts. Then by putting $\mathcal{T}$ after $\mathcal{S}$, we mean $\mathcal{S}$ must be reduced before $\mathcal{T}$. An optional semicolon may be inserted between $\mathcal{S}$ and $\mathcal{T}$.

$$\left\langle\, \mathcal{S}\mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \to \left\langle\, \mathcal{S}'\mathcal{T} \,,\, \sigma' \,,\, \omega \,\right\rangle \text{ if } \left\langle\, \mathcal{S} \,,\, \sigma \,\right\rangle \to \left\langle\, \mathcal{S}' \,,\, \sigma' \,\right\rangle.$$

or

$$\left\langle\, \mathcal{S}\mathbf{;}\mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \to \left\langle\, \mathcal{S}'\mathbf{;}\mathcal{T} \,,\, \sigma' \,,\, \omega \,\right\rangle \text{ if } \left\langle\, \mathcal{S} \,,\, \sigma \,\right\rangle \to \left\langle\, \mathcal{S}' \,,\, \sigma' \,\right\rangle.$$

If $\mathcal{S}$ cannot be reduced any more because it is now a primitive, we can move on reducing $\mathcal{T}$.

$$\left\langle\, e\mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \to \left\langle\, \mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \text{ if } e \text{ is an expression.}$$

or

$$\left\langle\, e\mathbf{;}\mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \to \left\langle\, \mathcal{T} \,,\, \sigma \,,\, \omega \,\right\rangle \text{ if } e \text{ is an expression.}$$

Parentheses factorisation. Parenthesis factorisation allows to capture several sequential statements in a block and return the value of the last. With it, one can write sentences such as $x = (\ \mathit{operation1}(y); \mathit{operation2}(z)\ )$.

If $\mathcal{S}$ is program text, the main rule is :

$$\langle\, (\mathcal{S})\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, (\mathcal{S}')\,,\,\sigma\,,\,\omega\,\rangle \text{ if } \langle\, \mathcal{S}\,,\,\sigma\,\rangle \rightarrow \langle\, \mathcal{S}'\,,\,\sigma'\,\rangle,$$

but if $\mathcal{S} = e$ an expression, then the parentheses can be removed :

$$\langle\, (e)\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, e\,,\,\sigma\,,\,\omega\,\rangle \text{ if } e \text{ is an expression.}$$

Parallel composition. Let $\mathcal{T}_i$ for i ranging from 1 to n be (partially reduced) program texts. Parallel composition, unlike sequential composition, allows these program texts to be reduced in any order. It is done with the **concurrently** primitive.

$$\langle\, \textbf{concurrently do } \mathcal{T}_1 \ldots \textbf{do } \mathcal{T}_i \ldots \textbf{do } \mathcal{T}_n \textbf{ end}\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, \textbf{concurrently do } \mathcal{T}_1 \ldots \textbf{do } \mathcal{T}_i' \ldots \textbf{do } \mathcal{T}_n \textbf{ end}\,,\,\sigma'\,,\,\omega\,\rangle$$

if $\langle\, \mathcal{T}_i\,,\,\sigma\,\rangle \rightarrow \langle\, \mathcal{T}_i'\,,\,\sigma'\,\rangle$.

Like in sequential composition, when a program text $\mathcal{T}_i$ is done, it is discarded from the text.

$$\langle\, \textbf{concurrently do } \mathcal{T}_1 \ldots \textbf{do } \mathcal{T}_i \ldots \textbf{do } \mathcal{T}_n \textbf{ end}\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, \textbf{concurrently do } \mathcal{T}_1 \ldots \textbf{do } \mathcal{T}_{i-1} \textbf{ do } \mathcal{T}_{i+1} \ldots \textbf{do } \mathcal{T}_n \textbf{ end}\,,\,\sigma'\,\rangle$$

if $\mathcal{T}_i$ is only an expression e.

Ask. We do not allow any more to react on the store when a constraint or its negation are not entailed as it may lead to non-monotonic behaviour in the reductions of the program text if used incorrectly. The construct **react** without the **otherwise** keyword is still valid though.

If for i ranging from 1 to $n \in \mathbb{N}$, c_i is a predicate and T_i is a program text, we have the reduction :

$$\langle\, \textbf{react} \ldots \textbf{on } c_i \textbf{ do } \mathcal{T}_i \ldots \textbf{end}\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, \mathcal{T}_i\,,\,\sigma\,,\,\omega\,\rangle \text{ if } \sigma \models c_i.$$

For readability Roze should support also the oz-like **if**-**then**-**else** primitive. If as usual c is a predicate and $\mathcal{T}$ and $\mathcal{S}$ are program texts, it reduces as such :

$$\langle\, \textbf{if } c \textbf{ then } \mathcal{S} \textbf{ else } \mathcal{T} \textbf{ end}\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, \mathcal{S}\,,\,\sigma\,,\,\omega\,\rangle \text{ if } \sigma \models c,$$
$$\langle\, \textbf{if } c \textbf{ then } \mathcal{S} \textbf{ else } \mathcal{T} \textbf{ end}\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, \mathcal{T}\,,\,\sigma\,,\,\omega\,\rangle \text{ if } \sigma \models \neg c.$$

Tell. The **tell** from CCP' is also a valid expression in Roze. If c is a constraint and adding c does not result in an inconsistent store, then c is added.

$$\langle\, \textbf{tell}\,(c)\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \langle\, ,\,\sigma \cup \{c\}\,,\,\omega\,\rangle \text{ if } \sigma \cup \{c\} \not\models \bot,$$
$$\langle\, \textbf{tell}\,(c)\,,\,\sigma\,,\,\omega\,\rangle \rightarrow \textit{failure} \text{ if } \sigma \cup \{c\} \models \bot,$$

Again, for readability, when the constraint is an equality, we allow to omit to write **tell** $(\dots)$ around it on the condition to use $:=$ as the equality value.

$$\langle\, e{:=}e\,,\,\sigma\,,\,\omega\,\rangle \;\rightarrow\; \langle\; \textbf{tell}\,(\xi{=}e)\,,\,\sigma \cup \{c\}\,,\,\omega\,\rangle,$$

where e and e' are variables.

Procedure declaration. A declared procedure must be stored in the constraint store, and is assigned to the new variable p. To merge concurrent declarations of the procedure, an entry in the symbol table is added, whose identifier P is compile-time generated.

$$\begin{aligned}\langle\; &\textbf{def}\ \_\ (\ \ X_1,\dots,X_n\ \ )\ \textbf{by}\ \ \mathcal{S}\ \ \textbf{end}\ \ ,\,\sigma\,,\,\omega\,\rangle\\ &\rightarrow\; \langle\ ,\,\sigma \cup \{p = \mathit{text}(X_1,\dots,X_n)(\mathcal{S})\}\,,\,\omega \oplus (\mathbf{s}_0^{\omega}, P, \{p\})\,\rangle,\end{aligned}$$

where $\mathcal{S}$ is a program text and X_1, dots, X_n are identifiers.

Replacing the underscore '_' with an identifier is a shorthand notation to bind the procedure to the variable denoted by this identifier.

$$\begin{aligned}\langle\; &\textbf{def}\ \ f\ \ (\ \ X_1,\dots,X_n\ \ )\ \textbf{by}\ \ \mathcal{S}\ \ \textbf{end}\ \ ,\,\sigma\,,\,\omega\,\rangle\\ &\rightarrow\; \langle\, f{:=}\ \ \textbf{def}\ \_\ (\ \ X_1,\dots,X_n\ \ )\ \textbf{by}\ \ \mathcal{S}\ \ \textbf{end}\ \ ,\,\sigma\,,\,\omega\,\rangle\end{aligned}$$

where f is a variable, and $\mathcal{S}$, X_1, dots X_n are as above.

Procedure call. Procedure call does three things. First, it creates a new procedure scope, if one has not yet been created. Secondly, it adds to this new scope an entry $(X_i, \{\xi_i\})$ for each ith identifier X_i of the procedure and ith argument variable ξ_i.

$$\begin{aligned}\langle\, f\ (\ \ \xi_1,\dots,\xi_n\ \ )\ \,,\,\sigma \cup \{p = \mathit{procedure}(X_1,\dots,X_n)(\mathcal{S})\}\,,\,\omega\,\rangle\\ \rightarrow\; \langle\, \mathcal{S}^{\xi_1,\dots,\xi_n}_{X_1,\dots,X_n} \diamond\,,\,\sigma \cup \{p = \mathit{procedure}(X_1,\dots,X_n)(\mathcal{S})\}\,,\,\omega'\,\rangle\end{aligned}$$

where $X_1,\dots,X_n$ are identifier names, $\xi_1,\dots,\xi_n$ are logical variables, $\mathcal{S}$ is a partially reduced program text and $\mathcal{S}^{\xi_1,\dots,\xi_n}_{X_1,\dots,X_n}$ is $\mathcal{S}$ with free occurrences of identifier X_i replaced by logical variable ξ_i for $1 \le i \le n$.

If $\mathbf{s}_1$ is the newly created scope, the output symbol table is :

$$\omega' = (\omega \lhd (\mathbf{s}_0^{\omega}, p)) \oplus (\mathbf{s}_1, X_1, \xi_1) \oplus \dots \oplus (\mathbf{s}_1, X_n, \xi_n).$$

Remember that operation $\diamond$ is an operation that tells to pop out the stack frame. It is added at the end of the scope.

Note this operation formally converts every free identifier of the scope's program text into a variable. But in practise, conversion of free identifiers is done dynamically using the symbol table.

Variable declaration. Variable declaration is very similar to immediately call a newly declared anonymous procedure. But unlike procedure call, we are not given variable arguments, new variables λ_i are created.

$$\left\langle \textsf{let } X_1, \ldots, X_n \textsf{ st } \mathcal{S} \textsf{ end}\,,\, \sigma\,,\, \omega \right\rangle \rightarrow \left\langle \mathcal{S}^{\lambda_1,\ldots,\lambda_n}_{X_1,\ldots,X_n} \diamond\,,\, \sigma\,,\, \omega' \right\rangle$$

where $X_1, \ldots, X_n$ are identifier names, $\lambda_1, \ldots, \lambda_n$ are new unique logical variables of the concerned site, $\mathcal{S}$ is a partially reduced program text and $\mathcal{S}^{\lambda_1,\ldots,\lambda_n}_{X_1,\ldots,X_n}$ is $\mathcal{S}$ with free occurrences of identifier X_i replaced by logical variable λ_i for $1 \leq i \leq n$. If $\mathbf{s}_1$ is the newly created scope, the output symbol table is :

$$\omega' = (\omega \lhd (\mathbf{s}^{\omega}_0, p)) \oplus (\mathbf{s}_1, X_1, \lambda_1) \oplus \cdots \oplus (\mathbf{s}_1, X_n, \lambda_n).$$

End of scope

$$\left\langle \diamond\,,\, \sigma\,,\, \omega \right\rangle \rightarrow \left\langle\ ,\, \sigma\,,\, \omega \right\rangle$$

Pop out the last frame of the stack frame. The stack frame is not presented here in the triple as it is not shared among sites.

4.4 CRDTs in Roze

We provide a few CRDTs as primitives of Roze. Those CRDTs can be combined together to produce new CRDTs. For example, the modification-wins graph CRDT can be built by combining the modifications-wins set CRDT and a modification-wins boolean CRDT. We thus also provide combinations functions, that output a new CRDT from one, two or more input CRDTs. For clarity we provide minimal basic CRDTs. Actually, we provide two : the *modification-wins* boolean and *the growing-only counter*.

A third basic CRDT may be useful when doing combinations, the constant CRDT, but it does not need special support. As there is only one state in the constant CRDT, no commits are possible and a view of the CRDT is necessarly a constant function. Thus a variable bound to an atom is a constant CRDT. See Almeida et al. [2013] for more examples on what CRDTsÒ can be obtained from these basic building blocks.

We have already shown how to translate some of these CRDTs in logic predicates and to code them in CCP' in chapter 3. Here we shall give a semantic description of the behaviour of the two basic CRDTs and the CRDT combination functions.

4.4.1 Reduction of primitive CRDTs in Roze

The following reductions describe how the two primitive CRDTs of Roze, the modification-wins boolean and the growing-only counter are reduced.

Modification-wins boolean. A boolean is more commonly known as a truth value by logicians. This CRDT truth value is thus a modification-wins boolean. It can be viewed as a modification-wins CRDT Set that must

be subset of a singleton set. We already view the modification-wins set in 3.4. Here are the possible reductions.

Do not be afraid of some of the slang used in the following reductions, such as "for every integer $k > n$, we impose σ does not entail $crdtBoolValue(\xi, k)$ or its negation" which means that n is the maximal version number used by predicates $crdtBoolValue$ concerned with ξ and e in store σ.

- Creation of the boolean. We create a new variable ξ and bind it to the built-in constant that tells it is now a CRDT Boolean. We then assign it to the given input value. Since b is a boolean, $v = b$ part means we store predicate v in the store if $b = \top$ or $\neg v$ else.

  $$\langle\, \mathsf{crdt\_bool\_new}(b)\,,\, \sigma\,,\, \omega\,\rangle \\ \rightarrow\, \langle\, \xi\,,\, \sigma \cup \{\xi = crdtBool\} \cup \{crdtBoolValue(\xi, 0) = b\}\,,\, \omega\,\rangle,$$

  where ξ is a new unique (to the site) variable and b is a truth value, true ($\top$) or false ($\bot$).

- Set the boolean to True.

  $$\langle\, \mathsf{crdt\_bool\_set}(\xi\,, \top)\,,\, \sigma\,,\, \omega\,\rangle \\ \rightarrow\, \langle\, ,\, \sigma \cup \{crdtBoolValue(\xi, n+1)\}\,,\, \omega\,\rangle,$$

  where ξ is a variable and
  - $\sigma \models \xi = crdtBool$
    (i.e. ξ is bound to a boolean CRDT in the constraint store),
  - $n \in \mathbb{N}$ and $\sigma \models \neg crdtBoolValue(\xi, n)\}$
    (i.e. the previous version value of the boolean was false),
  - for every integer $k > n$, we impose σ does not entail $crdtBool(\xi, k)$ or its negation.

  $$\langle\, \mathsf{crdt\_set\_add}(\xi\,, \top)\,,\, \sigma\,,\, \omega\,\rangle \;\rightarrow\; \langle\, ,\, \sigma\,,\, \omega\,\rangle,$$

  where ξ is a variable and
  - $\sigma \models \xi = crdtBool$,
  - $n \in \mathbb{N}$ and $\sigma \models crdtBoolValue(\xi, n)\}$,
  - for every integer $k > n$, we impose σ does not entail $crdtBool(\xi, k)$ or its negation.

- Setting the boolean to false. We won't write the lengthy reduction here, as they are very similar to the two previous reductions. Just replace the occurrences of $\neg crdtBool$ by $crdtBool$ and conversely.

- Asking the value of the boolean. If we can tell the value of a boolean, we should also be able to ask for it. This is what the following reduction does.

  $$\langle\, \mathsf{crdt\_bool\_val}(\xi)\,,\, \sigma\,,\, \omega\,\rangle \;\rightarrow\; \langle\, b\,,\, \sigma\,,\, \omega\,\rangle,$$

  where ξ is a variable and

  - $\sigma \models \xi = \mathit{crdtBool}$,
  - $n \in \mathbb{N}$ and $\sigma \models \mathit{crdtBoolValue}(\xi, n) = b\}$ for some truth value b ($\top$ or $\bot$),
  - for every integer $k > n$, we impose σ does not entail $\mathit{crdtBool}(\xi, k)$ or its negation.

- $\top$-wins boolean. Sometimes a boolean that behaves differently may be needed. The $\top$-wins boolean is false ($\bot$) until someone declare it is true ($\top$). It has only two states : "still at the false value" and "now at the true value".

  It is fairly easier to define than the modification-wins boolean, but we do not need to add it as a primitive, as it can be easily defined using the modification-wins boolean. Indeed, if we allow declarations only with the initial value false ($\bot$) and we allow only to set the boolean to true ($\top$), then we have exactly the $\top$-wins boolean. Yet other variations may be declared using one of these two booleans, like a $\bot$-wins boolean.

Growing-only counter. The growing-only counter is a CRDT non-negative integer that can only be incremented. It is an easy CRDT to translate in logic terms, because the version number of the states equals the value the integer is supposed to be at.

- Creation of the growing-only counter. As for every CRDT, we simply create a new variable and bind it to a constant symbol that tells it is a CRDT.

  $$\langle\, \mathit{crdt\_uint\_new}\,,\, \sigma\,,\, \omega\,\rangle \\ \quad \to \langle\, \xi\,,\, \sigma \cup \{\xi = \mathit{crdtUint}\} \cup \{\mathit{crdtUintValue}(\xi) \geq 0)\}\,,\, \omega\,\rangle,$$

  where ξ is a new unique variable.

- Incrementation of the counter. This is again very simple. We know $\mathit{crdtUintValue}(\xi)$ is larger than some n and we cannot yet tell it is larger than $n+1$. We add the constraint that it is larger than $n+1$.

  $$\langle\, \mathit{crdt\_uint\_inc}(\xi)\,,\, \sigma\,,\, \omega\,\rangle \to \langle\, ,\, \sigma \cup \{\mathit{crdtUintValue}(\xi) \geq n{+}1\}\,,\, \omega\,\rangle,$$

  where ξ is a variable and
  - $\sigma \models \xi = \mathit{crdtUint}$,
  - $n \in \mathbb{N}$ and $\sigma \models \mathit{crdtUintValue}(\xi) \geq n\}$,
  - for every integer $k > n$, we impose σ does not entail $\mathit{crdtUintValue}(\xi) \geq k$ or its negation.

- Value of the counter. This is again very simple. We simply choose the largest n so that we can deduce $\mathit{crdtUintValue}(\xi) \geq n$ to replace the CRDT variable.

  $$\langle\, \mathit{crdt\_uint\_val}(\xi)\,,\, \sigma\,,\, \omega\,\rangle \;\to\; \langle\, n\,,\, \sigma\,,\, \omega\,\rangle,$$

  where ξ is a variable and

– $\sigma \models \xi = crdtUint$,
– $n \in \mathbb{N}$ and $\sigma \models crdtUintValue(\xi) \geq n\}$,
– for every integer $k > n$, we impose σ does not entails $crdtUintValue(\xi) \geq k$ or its negation.

4.4.2 Combining CRDTs

We now overview how to combine CRDTs. This is a direct translation of the techniques seen in section 2.5 into a programming language.

We assume CRDT procedures of a CRDT $\mathcal{C}$ are stored in a record tool kit. We use an abuse of notation and write $\mathcal{C}$ also for this record. Such a record has the form $\mathcal{C} = \mathit{crdt}(\mathit{new} : n_{\mathcal{C}}, \mathit{commits} : c_{\mathcal{C}}, \mathit{views} : v_{\mathcal{C}})$, where $\mathcal{C}.\mathit{new}$ is a procedure to initialise the CRDT, $\mathcal{C}.\mathit{commits}$ is a record whose field are the various commits procedures that can be applied to the CRDT and where $\mathcal{C}.\mathit{views}$ is a record whose fields are the various views that a user can have on the CRDT.

We won't provide the code that gives the correct CRDT record of a combined CRDT from the two initial CRDT records. This code is tedious to understand but it is not insightful. Rather, we provide the reduction of the call to create a new combined CRDT, to commit to a combined CRDT or to view a combined CRDT.

As we did not provide the construction of the category of functor $\mathsf{Fun}(\mathcal{C}, \mathcal{D})$ in section 2.5, we will not provide the corresponding reductions in this section either, even though it is a valuable way to combine CRDTs.

Cartesian product. The Cartesian product is the easiest combination. It actually does not need a specific language support. Using two distinct CRDTs in the store is actually using their Cartesian product. Indeed performing a commit on one is like performing a commit on the couple by leaving one in the same state and committing the other. Merging the constraint store will merge both CRDTs at the same time. So the Cartesian product is simply using two distinct CRDT variables and no special syntactic or semantic support is needed for that.

Lexicographic union. We introduce the lexicographic union of two CRDTs before the lexicographic product because it is fairly more easier to describe. So let $\mathcal{C}$ and $\mathcal{D}$ be two CRDTs. We want to compute $\mathcal{C} \sqcup \mathcal{D}$.

- Creation of the CRDT. The creation of the CRDT is straightforward. The CRDT $\mathcal{C}$ and $\mathcal{D}$ are created and saved in a record. So far, so good, we could say we built up $\mathcal{C} \times \mathcal{D}$. We simply add a predicate, *crdtLexUnionIsOnRight* that takes the variable of CRDT lexicographic union and tells whether we use objects of CRDT $\mathcal{D}$ or not. As the creation, we use objects of the category $\mathcal{C}$, so the predicate must not be put in the store. The reduction is :

$$\langle\, \mathit{crdt\_lexUnion\_new}(\mathcal{C}, \mathcal{D})\ ,\ \sigma\ ,\ \omega\ \rangle \rightarrow \langle\, (\xi = \mathit{crdtLexUnion}(\mathcal{C}.\mathit{new}, \mathcal{D}.\mathit{new})\ ;\ \xi)\ ,\ \sigma\ ,\ \omega\ \rangle$$

  where ξ is a fresh variable and $\mathcal{C}.\mathit{new}$ is the initialisation function of the CRDT $\mathcal{C}$ and $\mathcal{D}.\mathit{new}$ the one of $\mathcal{D}$.

- Commits. If the CRDT is in its "left" states, then its possible commits are the commits of the CRDT $\mathcal{C}$ and a special commit to change to the "right" states. In the "right" states, the possible commits are the commits of the CRDT $\mathcal{D}$. "Moving to the right" could be done by adding a unary relation symbol *crdtLexUnionIsOnRight*. The reduction of this particular commit is

$$\langle\, \mathit{crdt\_lexUnion\_changeRight}(\xi)\,,\,\sigma\,,\,\omega\,\rangle \\ \rightarrow\, \langle\, \xi\,,\,\sigma \cup \{\mathit{crdtLexUnionIsOnRight}(\xi)\}\,,\,\omega\,\rangle,$$

  where $\sigma \models \xi = \mathit{crdtLexUnion}(\xi_{\mathcal{C}}, \xi_{\mathcal{D}})$.
  The reduction of the "left" commit is simply an indirect call to the reduction of a commit of the CRDT $\mathcal{C}$.

$$\langle\, \mathit{crdt\_lexUnion\_commitLeft}(\xi\,,\mathcal{C}\,,\,i_{\mathcal{C}}\,,e_{\mathcal{C}})\,,\,\sigma\,,\,\omega\,\rangle \\ \rightarrow\, \langle\, \mathcal{C}\,.\,\mathit{commits}.\,i_{\mathcal{C}}(\xi_{\mathcal{C}}\,,e_{\mathcal{C}})\,,\,\sigma\,,\,\omega\,\rangle$$

  where
  - $\sigma \models \xi = \mathit{crdtLexUnion}(\xi_{\mathcal{C}}, \xi_{\mathcal{D}})$,
  - $\mathcal{C} = \mathit{crdt}(\mathit{new} : n_{\mathcal{C}}, \mathit{commits} : c_{\mathcal{C}}, \mathit{views} : v_{\mathcal{C}})$ is a CRDT record,
  - $i_{\mathcal{C}}$ is a record index of the record of commit procedures $\mathcal{C}\,.\,\mathit{commits}$,
  - $e_{\mathcal{C}}$ is an argument for the commits $\mathcal{C}\,.\,\mathit{commits}.i_{\mathcal{C}}$,
  - $\sigma \not\models \mathit{crdtLexUnionIsOnRight}(\xi)$.

  The reduction of a "right" commit *crdt_lexUnion_commitRight* is defined in a similar way, but when $\sigma \models \mathit{crdtLexUnionIsOnRight}(\xi)$.
- The views, in the "left" states (*crdt_lexUnion_viewLeft*) or in the "right" states (*crdt_lexUnion_viewRight*), are as well defined in a very similar manner as the "left" or "right" commit of a Lexicographic union-combined CRDT.
- $\top$-wins boolean. Curiously enough, the $\top$-wins boolean can also be defined as the lexicographic product of the constant CRDT *false* and the constant CRDT *true* :

$$\bot \sqcup \top$$

Lexicographic product. The lexical product is more difficult to understand. Be sure to watch again figure 2.3. Again, let $\mathcal{C}$ and $\mathcal{D}$ be two CRDTs. We of course need to create a new instance of the CRDT $\mathcal{C}$.

The figure suggests that each new state in $\mathcal{C}$ brings a new reinitialised instance of $\mathcal{D}$, and this is true. When committing a new state of the instance of $\mathcal{C}$, a new instance of $\mathcal{D}$ must be pointed to. Thus we use a version counter of the states of the instance of $\mathcal{C}$ thanks to the primitive non-negative integer CRDT we described above. To each value of the counter is associated a unique instance of $\mathcal{D}$. When committing for the instance of $\mathcal{C}$, the counter is incremented, and to the new value is associated a freshly created instance of $\mathcal{D}$.

- Creation of the CRDT.

$$\begin{aligned}\langle\, \mathit{crdt\_lexProduct\_new}(\mathcal{C},\mathcal{D})\,,\,\sigma\,,\,\omega\,\rangle \\ \rightarrow\ \langle\, (\ \textbf{tell}\,(\ \xi{=}\ \mathit{crdtLexProduct}(\mathcal{C}\,.\,\mathit{new})\ )\ ; \\ \textbf{tell}\,(\ \mathit{crdtLexProductVersion}(\xi)\ =\ \mathit{crdt\_uint\_new}\ )\textbf{;} \\ \textbf{tell}\,(\ \mathit{crdtLexProductRight}(\xi,0)\ =\mathcal{D}\,.\,\mathit{new}\ )\ ;\ \xi)\,,\,\sigma\,,\,\omega\,\rangle\end{aligned}$$

  where ξ is a fresh variable and $\mathcal{C}\,.\,\mathit{new}$ is the initialisation function of the CRDT $\mathcal{C}$ and $\mathcal{D}\,.\,\mathit{new}$ the one of $\mathcal{D}$.

- Commits. The commit of the lexicographic product corresponds to the commits of $\mathcal{C}$ and the commits of $\mathcal{D}$. In figure 2.3, we showed that the lexicographic product of $\mathcal{C}$ and $\mathcal{D}$ can be seen as $\mathcal{C}$ with a inner copy of $\mathcal{D}$ in each of its objects. Therefore one must be able to move in the copy of $\mathcal{D}$ or to move in $\mathcal{C}$ and use a new copy of $\mathcal{D}$. Let us begin with the easy commit reduction, the one on the instance of $\mathcal{D}$.

$$\begin{aligned}\langle\, \mathit{crdt\_lexProduct\_commitRight}(\xi,\mathcal{D},\,i_{\mathcal{D}},e_{\mathcal{D}})\,,\,\sigma\,,\,\omega\,\rangle \\ \rightarrow\ \langle\, \mathcal{D}\,.\,\mathit{commits}.\,i_{\mathcal{D}}(\xi_{\mathcal{D}},e_{\mathcal{D}})\,,\,\sigma\,,\,\omega\,\rangle\end{aligned}$$

  where
  - $\sigma \models \xi = crdtLexUnion(\xi_{\mathcal{C}})$,
  - $\mathit{crdt\_uint\_val}(crdtLexProductVersion(\xi)) = n$
  - $crdtLexProductRight(\xi, n) = \xi_{\mathcal{D}}$
  - $\mathcal{D} = \mathit{crdt}(\mathit{new} : n_{\mathcal{D}}, \mathit{commits} : c_{\mathcal{D}}, \mathit{views}\ : v_{\mathcal{D}})$ is a CRDT record,
  - $i_{\mathcal{D}}$ is a record index of the record of commit procedures $\mathcal{D}\,.\,\mathit{commits}$,
  - $e_{\mathcal{D}}$ is an argument for the commits $\mathcal{D}\,.\,\mathit{commits}.i_{\mathcal{D}}$.

  Now we proceed with the other commit reduction, the one on the instance of $\mathcal{C}$.

$$\begin{aligned}\langle\, \mathit{crdt\_lexProduct\_commitLeft}(\xi,\mathcal{C}\,,\,i_{\mathcal{C}}\,,e_{\mathcal{C}})\,,\,\sigma\,,\,\omega\,\rangle \\ \rightarrow\ \langle\, (\ \mathcal{C}\,.\,\mathit{commits}.\,i_{\mathcal{C}}(\xi_{\mathcal{C}},e_{\mathcal{C}})\textbf{;} \\ \mathit{crdt\_uint\_inc}(\ \mathit{crdtLexProductVersion}(\xi)\ )\textbf{;} \\ \mathit{crdtLexProductRight}(\xi,\ \mathit{crdt\_uint\_val}(\ \mathit{crdtLexProductVersion}(\xi)\ ))\ =\mathcal{D}\,.\,\mathit{new}\ \textbf{;}\ )\,, \\ \sigma\,,\,\omega\,\rangle\end{aligned}$$

  where
  - $\sigma \models \xi = crdtLexUnion(\xi_{\mathcal{C}})$,
  - $\mathcal{C} = \mathit{crdt}(\mathit{new} : n_{\mathcal{C}}, \mathit{commits} : c_{\mathcal{C}}, \mathit{views}\ : v_{\mathcal{C}})$ is a CRDT record,
  - $i_{\mathcal{C}}$ is a record index of the record of commit procedures $\mathcal{C}\,.\,\mathit{commits}$,
  - $e_{\mathcal{C}}$ is an argument for the commits $\mathcal{C}\,.\,\mathit{commits}.i_{\mathcal{C}}$.

- The views of the lexicographic product are the view of the CRDT $\mathcal{C}$ and $\mathcal{D}$ by selecting the appropriate component first. For example, a view on $\mathcal{D}$ is explained by the reduction :

$$\begin{aligned}\langle\, \mathit{crdt\_lexProduct\_viewLeft}(\xi,\mathcal{C}\,,\,i_{\mathcal{C}})\,,\,\sigma\,,\,\omega\,\rangle \\ \rightarrow\ \langle\, \mathcal{C}\,.\,\mathit{views}.\,i_{\mathcal{C}}(\xi_{\mathcal{C}})\,,\,\sigma\,,\,\omega\,\rangle\end{aligned}$$

  where

- $\sigma \models \xi = crdtLexUnion(\xi_{\mathcal{C}})$,
- $\mathsf{crdt\_uint\_val}(crdtLexProductVersion(\xi)) = n$
- $crdtLexProductRight(\xi, n) = \xi_{\mathcal{D}}$
- $\mathcal{D} = \mathsf{crdt}(\mathsf{new} : n_{\mathcal{D}}, \mathsf{commits} : c_{\mathcal{D}}, \mathsf{views} : v_{\mathcal{D}})$ is a CRDT record,
- $i_{\mathcal{D}}$ is a record index of the record of view procedures $\mathcal{D}.\mathsf{views}$.

$\mathsf{crdt\_lexProduct\_viewRight}$ is defined similarly.

4.5 Garbage collection

Since in Roze, the constraint store is *monotonic*, it converges toward the use of infinite memory. Some small specific but amazingly efficient garbage collection can be done, like removing the constraint $x \geq k$ from the store when adding the constraint $x \geq n$ with $n > k$ since the former can be deduced from the latter.

A more generic garbage collection could be explored for CRDT by using a *lattice* rather than a semi-lattice. A lattice is a set with two semi-lattice structures such that the two binary operators of these structures, the joint $\vee$ and the meet $\wedge$, obeys absorption laws :

$$x \vee (x \wedge y) = x, \qquad x \wedge (x \vee y) = x.$$

If we translate the lattice into a poset (with the order induced by the joint $\vee$), then into a category as done in (2.5) of chapter 2, then the meet corresponds to the *product* of two state objects. The product has been defined in (2.10) and is illustrated by figure 2.1. It is the dual notion of the coproduct defined in (2.12).

If the joint $\vee$ corresponds to a CRDT merge operator, the meet $\wedge$ corresponds to the state obtained by applying operations observed by both replicas on the input state of the CRDT. Thus if we can compute the meet of the state of each replica of the CRDT, we could also collect garbage operations observed by the meet. In other words, the state of the instance of a CRDT resulting of operations observed by every replicas need not to be CRDTs. Only modifications resulting from the application of operations we are unsure whether other sites have observed it should be made "CRDT"-mergeable. In yet other terms, we could use the meet of the states of each replica (or some other state we are sure to be dominated by the meet) as the common basis to build the CRDT upon.

An algorithm doing this garbage collection on a few basic CRDTs is yet to be found. Even more promising is the possibility to have an algorithm that do garbage collection on a combined CRDT based on the garbage collection algorithms of each CRDT used to do the combination. This should be possible because as we can compute the coproduct of a combined Category from the coproducts of the categories used to do the combination, it is possible to compute the product of the combined categories from the product of the categories used to do the combination.

Conclusion

This thesis worked on two goals. The main goal was to combine conflict-free replicated data types with concurrent constraint programming to provide an asynchronously distributed programming language, transparent to the code writer. A secondary goal was to convert those conflict-free replicated data types from the semi-lattice formalism to the formalism of Categories.

Conclusion on category theory

The secondary goal about categories is not tied to the first about combining CRDTs and CCP'. We just stated the most interesting difference between a lattice and a CRDT category : "Unlike the semi-lattice formalism, nothing prevents to have more than one arrow between two states". This means that with categories, you may take into account the state of an instance of a CRDT, but you can also consider how the *transition* between states is performed.

For example, we saw the notion of joint in a semi-lattice corresponds to the notion of coproduct in a Category. Actually, there is a more general definition of the coproduct that takes arrows into account. Its name is the *colimit*. The colimit is performed not over a set of objects of the category, but over a sub-diagram of that category. By a sub-diagram we mean the image of a "tiny" category into the considered category by a functor. Thus the colimit object must not only have arrows from the objects of the sub-diagram, these arrows must also commute with the arrows of the sub-diagram. In other words, we can do a merge of states and a transition between theses states.

Conclusion on combining CRDTs and CCP'

We did not achieve every detail of this main goal, as we did not provide how to build the CRDT of mappings $\mathsf{Fun}(\mathcal{A}, \mathcal{C})$ from the CRDT $\mathcal{C}$ and the fixed set $\mathcal{A}$ in order to avoid to overload more an already heavily loaded in concepts thesis. This is not a tricky issue but it might be hard to understand for the person without practise in category theory. The elements of the partial map Crdt are all possible partial mappings from elements of $\mathcal{A}$ to $\mathcal{C}$.

With this final combination done, a good future work would be to write down a general method to convert a CRDT into a logic form. To do that we would therefore like to convert any CRDT into a combination of the basic building blocks. So we first need to prove that any CRDT can be build upon a few of these basic building blocks, if this is true.

CRDT transparency

A final possible future work is about how to have an easy-to-use CRDT. In this version of Roze, CRDTs are clear to the programmer. We given him the creation, commits and view procedures. But a programmer does not want to code using CRDTs. he wants to add, subtract or multiply integers, no a value from the Cartesian product of twice $\mathbb{N}$'s. A way to split more clearly the view type and the CRDT type in Roze is probably a good idea. It would need work to translate operations on the viewed type into operations on the CRDT.

Notes

[1]See Niehren et al. [2006].
[2]See Gilbert and Lynch [2002].
[3]See Zawirski et al. [2011].
[4]See Zawirski et al. [2013].
[5]See Zawirski et al. [2011].
[6]See Mattern [1989].
[7]The reader with some knowledge in mathematics will observe that there exists no loop of states dominating each other, by the transitivity and antisymmetry properties of an order relation.
[8]See Zawirski et al. [2012].
[9]See Zawirski et al. [2011].
[10]See Almeida et al. [2013].
[11]See Adámek et al. [1990].
[12]It is not important to know *what* element is the arrow. Remember that in category theory, it is the interactions between the studied objects that matters
[13]Distinction between different effects on a state is already taken care of, as they bring to different states. Thus transitions are in different arrow sets.
[14]This actually leads to the same state up to a change to another state through an isomorphism. An isomorphism is an arrow f such that there is an arrow g that "undo" f, i.e. such that $f \circ g$ and $g \circ f$ are identity arrows.Since in the case of CRDT categories the sole way to verify this is when f input and output objects are the same, there cannot be two distinct objects that are isomorphic. Thus "up to change to another state through an isomorphism" doesn't mean anything and can be dropped out.
[15]An abelian group is a group whose binary operation is commutative, i.e. $x + y = y + x$ for every element x and y in the group.
[16]We could use more than one arrow rather than only $m_{C,D}$, but this would involve more complicated mathematics so that we can change slightly the behaviour of $\mathcal{C}(D, D')$ to take into account what arrow was used to move from a *left* space to a *right* space.
[17]See [Rossi et al., 2006, p.486–493].
[18]See Saraswa et al. [1991].
[19]See Rautenberg [2006].
[20]Beside the fact that cc is net very readable for large softwares.
[21]See Van Roy and Haridi [2004].
[22]See Van Roy and Haridi [2004].
[23]Observe we mixed up identifiers and variables in figure 3.1 in chapter 3. Although the programmers does not write variables but identifiers, identifiers are replaced with variables, then operations can be applied on the variables.
[24]This notion is of course borrowed from set theory. Let $\sim$ be an equivalence relation for the set A. That is for every a, b and c in A : $a \sim a$ (reflexivity), $a \sim b \wedge b \sim c \implies a \sim c$ (transitivity) and $a \sim b \implies b \sim a$ (symmetry) hold. Then we can build a "quotion set" made of equivalence classes. In this case the equivalence class of an element $a \in A$ is the set $\{b \in A | b \sim a\}$. By the properties of an equivalence relation, if $a \sim b$ then equivalence class of a and b is the same, else they are disjoint. In this case the set of variable is quotiented by the equivalence relation that identifies variable from different sites but with the same semantics meaning.
[25]See Haridi et al. [1999].
[26]This is the transitivity of the equivalence relation.

Bibliography

Jiří Adámek, Herrlich Horst, and George E. Strecker. *Abstract and Concrete Categories : The Joy of Cats*. John Wiley and Sons, 1990. URL `http://katmat.math.uni-bremen.de/acc/`.

Paulo Sérgio Almeida, Carlos Baquero, and Alcino Cunha. Composing lattices and crdts. 2013. URL `http://www.dagstuhl.de/mat/Files/13/13081/13081.BaqueroCarlos.Slides.pdf`.

Seth Gilbert and Nancy Lynch. Brouwer's conjecture and the feasibility of consistent available partition tolerant web. *SIGACT News*, 33(2):48–51, 2002.

Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, 1999.

Friedemann Mattern. Virtual times and global states of distributed systems. pages 215–226, 1989.

J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science.*, 364(3):338–356, 2006.

Wolfgang Rautenberg. *A Concise Introduction to Mathematical Logic*. Springer, 2006.

Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

Vijay A. Saraswa and Rinard Martin. Concurrent constraint programming. *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 232–245, 1990.

Vijay A. Saraswa, Rinard Martin, and Prakash Panangaden. Semantic fundations of concurrent constraint programming. *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 333–352, 1991.

Gert Smolka. A foundation for higher-order concurrent constraint programming. *1st International Conference on Constraints in Computational Logics*, 1994.

Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

Marek Zawirski, Annette Bieniusa, Valter Balegas, Nuno Preguiça, Sérgio Duarte, Marc Shapiro, and Carlos Baquero. Conflict-free replicated data types. pages 386–400, 2011.

Marek Zawirski, Annette Bieniusa, Valter Balegas, Nuno Preguiça, Sérgio Duarte, Marc Shapiro, and Carlos Baquero. Geo-replication all the way to the edge. 2012.

Marek Zawirski, Annette Bieniusa, Valter Balegas, Nuno Preguiça, Sérgio Duarte, Marc Shapiro, and Carlos Baquero. Strong eventual consistency, conflict-free replicated data types. 2013.