Framework Development

Hans-Dieter A. Hiep

January 23, 2017

1 Introduction

This document comprises the development of the framework, to be implemented in Haskell. The sequel is organized as follows: section 2 introduces the preliminaries in detail, section 3 describes the most important structures that are used within the framework. Where possible, the document includes annotations with background information regarding design choices.

2 Preliminaries

Definition 1. Given a binary relation $R \subseteq X \times Y$. By *successor neighbourhood* xR we denote the set $xR = \{y \in Y \mid x\ R\ y\}$. By *predecessor neighbourhood* Ry we denote the set $Ry = \{x \in X \mid xRy\}$. In particular, a one-to-one binary relation has a singleton successor neighbourhood and a singleton predecessor neighbourhood.

Definition 2. A *directed graph* G is a pair (V, E) of a set of vertices V and a set of directed edges $E \subseteq V \times V$. The sets of outgoing edges out_v and incoming edges in_v parameterized by some vertex $v \in V$ are defined as usual. In particular, a *weighted directed graph* G is a triple (V, E, w) where $w : E \to \mathbb{R}$ is a weight function that assigns to every edge a non-negative weight. An *undirected graph* G is a directed graph where for every edge (u, v) there is also an edge (v, u). A *loop* is an edge (u, u).

Definition 3. An *independence relation* (or commutation relation) $I \subseteq S \times S$ is an irreflexive, symmetric relation over S. In particular the *independency graph* is (S, I) where I is an independence relation over S, and can be represented by an undirected graph with vertex set S and edge set I. The complement $D = S \times S \backslash I$ is called a *dependence relation*. The *dependency graph* (S, D) is defined similarly, and often depicted without loops. (Adopted from [DM97].)

Definition 4. A *monoid* $\mathcal{M} = (S, \bullet, \epsilon)$ is a set S together with some binary operation $\bullet : S \times S \to S$ for which the following properties holds:

1. associativity, for all $s, t, u \in S$ it holds that $(s \bullet t) \bullet u = s \bullet (t \bullet u)$;
2. identity, there exists an element $e \in S$, such that for every $a \in S$, it holds that $e \bullet a = a = a \bullet e$.

Definition 5. A *partially commutative monoid* $\mathcal{C}(I)$ is a monoid $(S, \bullet, \epsilon)$ with an independence relation $I \subseteq S \times S$ for which the following property holds:

3. Commutativity, for all $s, t \in S$ it holds that $(s, t) \in I \Rightarrow s \bullet t = t \bullet s$.

In particular, we call $\mathcal{C}(I)$ or $\mathcal{C}$ a *(totally) commutative monoid*, i.e. a monoid where the above property holds for any element, if I consists of every pair (s, t) of unique elements $s, t \in S$. We say that $\mathcal{C}(\emptyset) = \mathcal{M}$, i.e. a monoid is a partially commutative monoid where every element is dependent. (Inspired by [HI05].)

Definition 6. An *alphabet* Σ is a finite set of which its elements are called letters. By Σ^* we denote the set of all finite sequences, *words*, over Σ (cf. Kleene star). For any $x \in \Sigma^*$, by $|x|$ we denote the length of the word and $|x|_a$ denotes the number of occurrences of the letter a in x. By Σ^n we denote the set of all worlds over Σ with fixed length n, i.e. for every $x \in \Sigma^n$ it holds $|x| = n$.

Definition 7. For any n, an *adjacent transposition relation* $T_I \subseteq \Sigma^n \times \Sigma^n$ induced by an independence relation $I \subseteq \Sigma \times \Sigma$ is a symmetric relation over Σ^n, where for $u, v \in \Sigma^n$, any letters $a, b \in I$ and words $p, s \in \Sigma^*$:

$$u\ T_I\ v \Leftrightarrow u = pabs \text{ and } v = pbas.$$

That is, $u\ T\ v$ if v is u with a transposition of two adjacent letters. By the *permutation relation* P_I induced by an independence relation I we denote the reflexive transitive closure of T_I for any n, that is, $u\ P\ v$ if and only if there exists a possibly empty sequence $z_1, \ldots, z_n$ of words in Σ^* such that:

$$u\ T_I\ z_1\ T_I\ \ldots\ T_I\ z_n\ T_I\ v.$$

By $[u]_I$ we denote the equivalence class of u under some P_I induced by I, i.e.

$$[u]_I = \{v \in \Sigma^* \mid u\ P_I\ v\}.$$

Definition 8. A *Dyck language* over some two letter alphabet $\Sigma = \{[,]\}$ is defined as

$$\{u \in \Sigma^* \mid imb(u) = 0 \leq imb(v) \text{ for all prefixes } v \text{ of } u\},$$

where $imb : \Sigma^* \to \mathbb{Z}$ is defined as $imb(u) = |u|_[- |u|_]$, denoting the imbalance of opening brackets unmatched with closing brackets.

3 Framework

We investigate five structures: that of states, that of messages, that of events, that of process descriptions and that of simulations. We assume that P is a finite set of processes, and that a network is a directed graph (P, C), consisting of processes as vertices and channels $C \subseteq P \times P$ as directed edges. We assume that every process has an implicit channel to itself, i.e. $(p, p) \in C$ for every $p \in P$.

3.1 Events

Definition 9. An *event* is parameterized on the message space M and state space Σ, happens at some process p, and is either:

1. an *internal* event, with a next state $\sigma \in \Sigma$;
2. a *send* event, with sent message $\mu \in M$, intended received $q \in P$, and next state $\sigma \in \Sigma$;
3. a *receive* event, with received message $\mu \in M$, original sender $q \in P$, and next state $\sigma \in \Sigma$;
4. a *crash* event.

The set of events is denoted E.

We deviate from the definition in [Fok13], since we explicitly require crashes happen as events. In a Byzantine setting, Byzantine processes can only send arbitrary messages after a crash event has occurred.

We say that any monoid on E is an execution, typically represented by a sequence of events. For sake of simplicity, we assume every execution is finite—although we will encounter possibly infinite executions, it is not practically necessary to simulate these infinite executions event by event.

Events can independently occur at different processes, hence we define a relation on the events in an execution specifying which events must happen before others.

Definition 10. The *causal order* is the smallest transitive dependence relation $\prec\ \subseteq E \times E$ such that for $a \prec b$:

1. if a and b happen at the same process, and a occurs before b, then $a \prec b$;
2. if a is a send event and b is a corresponding receive event, then $a \prec b$.

The last property deserves some attention. What does it mean that a send event corresponds to a receive event? We illustrate this using the following example. Let different kinds of punctuation indicate different corresponding events.

$$[\ \ (\ \ \{\ \ \{\ \ ]\ \ \}\ \ )\ \ [\ \ (\ \ \}\ \ )\ \ ]$$

So in this example, there are three different correspondences. We make precise what we mean by corresponding, below. Internal process state is irrelevant for matching corresponding events. In the diagram above, we assume that the brackets, parens and braces are mutually different correspondences (i.e. brackets do not correspond to parens, et cetera). Open marks indicates a send event and a closed mark indicates a receive event.

Definition 11. A send event $s \in E$ *corresponds to* a receive event $r \in M$ if and only if:

1. the message μ is the same at s and r,
2. the intended receiver at s is the same as the process where r happens,
3. the original sender at r is the same as the process where s happens.

Example 12.

$t =$	0	1	2	3	4	5	6	7	8	9	A	B
	[				]			[				]
		(					)		(		)	
			{	{		}				}		

We show the corresponding messages in a separate row. Do the first and third mark correspond? For brackets and parens: two send events never match. In the case of braces, however, the messages seemingly correspond. However, the second and the third brace also correspond according to our definition, and so does the first and second brace correspond to the fourth brace. Thus, we have four correspondences. For brackets and parens, we have three correspondences (why?). To demonstrate that this is wrong, consider

$$[\quad]\quad]$$

which has two correspondences: the first send is received both by the second and third event. Clearly, the last receive event should never happen without a corresponding send event.

> *Philosophical note*: Can receive events happen before send events? Consider two processes, p and q. Suppose q has knowledge of the inner working of p. Now p sends a message containing its new state, and progresses to that new state and pauses. Only when p wakes up, after a while, will it send a message to q. However, when q receives the first message by p it can simulate the inner working of p and thus know that it will receive a message from p. Accordingly, q receives the message that will be send by p, but before the actual message is sent. After a while, p wakes up and actually sends the message to q. Is it a realistic assumption that messages are never received before they are sent? May processes simulate each other in this way? What if a process is simulated that has in the mean time crashed? What about abstract simulation, where only an abstract part of the other machine is simulated, or speculative simulation, where non-deterministically multiple possible states are simulated until enough evidence is gathered to conclude the actual state, or undoable simulation, where every consequence of a faulty simulation can be undone?

Now consider a related problem: suppose one is walking past the events from left to right, is it possible to say at any point in time which sent messages are not received? We will comment on this problem later on.

3.2 Representations of free partially commutative monoids

Informally, a free object is a collection of elements where only the algebraic properties hold. In particular a free monoid $F\mathcal{M}(\Sigma)$ is the set of all words of some generating alphabet Σ. A free totally commutative monoid $F\mathcal{C}(\Sigma)$ is the set of all finite multi-sets consisting of letters of a generating alphabet Σ. A free partially commutative monoid $F\mathcal{C}(\Sigma, I)$ is the set of equivalence classes $[x]_I$ induced by the independence relation $I \subseteq \Sigma \times \Sigma$.

A representation of a free monoid is simply Σ^*. A representation for free totally commutative monoids can be found within a representation of free partially commutative monoids, where by definition I is fixed. A suitable representation for free partially commutative monoids are labelled directed graphs [Per85], as follows:

For each word $s \in \Sigma^*$ we construct a dependency graph, with as vertices the integers $1, \dots, |s|$, where each vertex i is labelled by the i-th letter s_i. Recall that D is the dependency relation induced by I. There is an edge (i, j) if and only if (1) $i < j$, and (2) $(s_i, s_j) \in D$, and (3) for all k such that $i < k < j$ we have $s_i \neq s_k$. There are at most $|\Sigma| \cdot |s|$ edges in this graph. Two free partially commutative monoids are equal if and only if their representation as dependency graphs are isomorphic with respect to labelling.

Example 13. Consider the independency relation $I = \{(a, b), (b, a), (c, d), (d, c)\}$. It has an induced dependency relation $D = \{(a, a), (a, c), (a, d), (b, b), (b, c), (b, d), (c, c), (d, d)\}$. Let $s = abdcbaa$. We construct a dependency graph as follows: for each vertex from left to right, we scan towards the right, creating edges if the corresponding labels are related in D. We stop scanning either when we reach the end of the list or after we reached a vertex with the same label.

An alternative representation for free totally commutative monoids is that of the function space $\mathbb{N}^\Sigma$, i.e. a mapping from every letter of the alphabet to some natural number. Intuitively, we count the number of occurrences of each element. Hence this representation is also called the multi-set representation.

A free partially commutative monoids has an alternative representations as a tuple of subsets of the alphabet, i.e. $2^\Sigma \times 2^\Sigma \times \cdots \times 2^\Sigma$. Within each block, letters can freely commute. We will use this representation for computing the normal form. There are two normal forms for free partially commutative monoids. We will only consider the normal form by Foata. Two elements of a free partially commutative monoid are equal if and only if they have the same unique normal form. Clearly, normal forms always preserve the length of the original word.

TODO

3.3 Dyck monoids

Definition 14. An *involution relation* $J \subseteq S \times S$ is an irreflexive, symmetric, one-to-one relation over some set S.

Every function $f : S \to S$ that is an involution, i.e. $f(f(a)) = a$, is also an involution relation. However, involution relations are more general, e.g. take $S = \{-, 0, +\}$ and let $-$ and $+$ be related, but 0 is unrelated. An element a related by an involution relation to some other element a^{-1} are also called the inverses of each other, [DLM08].

Example 15. $S = \{\downarrow, -, 0, +, \uparrow\}$, with $-$ and $+$ related, $\uparrow$ and $\downarrow$ related.

The notation of two elements a and a^{-1} suggest that aa^{-1} cancel, i.e. $aa^{-1} = 1$. However, due to symmetry, we also have $a^{-1}a = 1$. In particular, with our problem in mind, we do not want that $a^{-1}a = 1$ cancel, i.e. a send must cancel a receive, but a receive must never cancel a send. We thus have a stronger relation:

Definition 16. A *signed involution relation* $\mapsto \subseteq S \times S$ is an irreflexive, one-to-one relation over some set S, such that the following property holds:

1. for all $s, t \in S$, if $s \mapsto t$ then not $t \mapsto u$ for any $u \in S$.

This relation is stronger than the involution relation J, since every involution is also a a signed involution but the converse need not hold. We think of the signed involution relation as three partitions of S. We have one partition of *passive* elements that are unrelated. The second partition is consists of *negative* elements (the union of all predecessor neighbourhoods, i.e. all first components in the pair) and the third partition consists of *positive* elements (the union of all successor neighbourhoods, i.e. all second components of the pair). The number of positive elements is always the same as the number of negative elements, by one-to-one.

Example 17. $S = \{\downarrow, -, 0, +, \uparrow\}$ and $- \mapsto +$, $\downarrow \mapsto \uparrow$. Here the negative elements are $\{\downarrow, -\}$ and the positive elements are $\{+, \uparrow\}$ and the passive elements are $\{0\}$.

We will now consider the main structure for events:

Definition 18. A *Dyck monoid* $\mathcal{D}(J)$ is a partially commutative monoid $\mathcal{C}(I) = (S, \bullet, \epsilon)$ with a signed involution relation $\mapsto \subseteq S \times S$ for which the following properties holds:

1. the independence relation I is induced by the dependence relation $D = \{(s, t) \mid s \text{ equals } t \text{ or } s \mapsto t\text{or } t \mapsto s \text{ for all } s, t \in S\}$;
2. for any $s, t \in S$, if $s \mapsto t$ then $s \bullet t = \epsilon$.

A Dyck monoid can be seen as a partially commutative partially inverse monoid [DLM08]. Two elements may cancel each other out, but the order in which they cancel is important, i.e. all elements x commute if and only if there is no other element with which x cancels (from either the left or the right).

In the case of an empty signed involuation relation, where we have no element related and all elements are passive, then by the first condition a Dyck monoid is a totally commutative monoid (since the dependency is only reflexive) and the second condition does not apply.

Example 19. Consider the previous example: $[\ (\ \{\ \{\ ]\ \}\ )\ [\ (\ \}\ )\ ]$. We let $[\mapsto]$, $(\mapsto)$, and $\{\mapsto\}$.

1. We can freely commute non-matching marks. Hence the following:

$$\begin{array}{cccccccccccc}
[& (& \{ & \{ &] & \} &) & [& (& \} &) &] \\
[&] & (& \{ & \{ & \} &) & [& (& \} &) &] \\
[&] & [& (& \{ & \{ & \} &) & (& \} &) &] \\
[&] & [&] & (& \{ & \{ & \} &) & (& \} &) \\
[&] & [&] & (&) & \{ & \{ & \} & (& \} &) \\
[&] & [&] & (&) & (& \{ & \{ & \} & \} &) \\
[&] & [&] & (&) & (&) & \{ & \{ & \} & \}
\end{array}$$

2. If we have two consequtive elements, where the element on the left is negative and the element on the right is positive, the two can be eliminated:

$$\begin{array}{cccccccccccc}
[&] & [&] & (&) & (&) & \{ & \{ & \} & \} \\
 & & [&] & (&) & (&) & \{ & \{ & \} & \} \\
 & & & & (&) & (&) & \{ & \{ & \} & \} \\
 & & & & & & (&) & \{ & \{ & \} & \} \\
 & & & & & & & & \{ & & & \}
\end{array}$$

Ultimately leading to the empty word ϵ.

Note that this example already suggests the existence of a reduction relation and possibly some *normal form* if we impose some order on the elements. We will investigate this later. Also, the reader might be wondering what the correspondence between ϵ and a Dyck language is!

Revisiting our earlier question: can we walk from the left to the right and say which messages are not yet received, i.e. say which open markers are not yet closed? To answer that question, we need to find out what a good representation of a Dyck monoid is. We will consider two representations: that of free Dyck monoids induced by any signed involution relation. We must also keep in mind that we do not want to construct incorrect instances, where e.g. [ ]] is an element, with more receive events than corresponding send events: we thus also consider a representation that only admits negative (or only positive) elements.

Similar to the Foata normal form, the underlying alphabet of a Dyck monoid is totally ordered. Given $\leq_\Sigma$, we impose the following induced total order $\leq$, and let $\pi(t) = s$ be the left-projection of the relation $s \mapsto t$:

$$s \leq t \iff \begin{cases} s, t \text{ are negative} & s \leq_\Sigma t \\ s \text{ is negative and } t \text{ positive} & s \leq_\Sigma \pi(t) \\ s, t\text{are positive} & \pi(s) \leq_\Sigma \pi(t) \\ s \text{ is positive and } t \text{ negative} & \pi(s) \leq_\Sigma t \\ s, t \text{ are passive and} & s \leq_\Sigma t \\ s \text{ is passive and } t \text{ not} & \text{never} \\ t \text{ is passive and } s \text{ not} & \text{always} \end{cases}$$

Or we simply use the order of the negative element to order elements within a Dyck monoid, and let passive elements be ordered after all negative and positive elements.

Technical point: in the implementation we will construct sequences from the back to the front. In this case, it must be possible to have a negative number of open marks, since from the back to the front we will most likely encounter receive events first. However, the algorithm must always progress to the end of the list: it is an error if the count is still negative at the end, since that means that in the sequence there were more receive events than send events.

References

[DLM08] Volker Diekert, Markus Lohrey, and Alexander Miller, *Partially commutative inverse monoids*, Semigroup Forum, vol. 77, Springer, 2008, pp. 196–226.

[DM97] Volker Diekert and Yves Métivier, *Partial commutation and traces*, Handbook of formal languages, Springer, 1997, pp. 457–533.

[Fok13] Wan Fokkink, *Distributed algorithms: An intuitive approach*, MIT Press, 2013.

[HI05] Toshihiro Hamachi and Kunio Inoue, *Embedding of shifts of finite type into the dyck shift*, Monatshefte für Mathematik **145** (2005), no. 2, 107–129.

[Per85] Dominique Perrin, *Words over a partially commutative alphabet*, Combinatorial algorithms on words, Springer, 1985, pp. 329–340.