Theorem. The (concurrent) category of computations $\mathcal{T}_{\mathcal{N}}$ of a Petri net $\mathcal{N}$ is a strict symmetric monoidal category.

Proof. This follows immediately from the Associativity, Identity, Parallel Associativity, Commutativity and Unit, and Functoriality axioms defining the equivalence relation $\equiv$.

Exercise. Prove that for any Petri net $\mathcal{N}$, any computation $m \xrightarrow{\alpha} m'$ has a [in general not unique] interleaving description as a sequential composition of basic transitions:

$$\left(m \xrightarrow{\alpha} m'\right) = \left(m = m_0 m_1 \xrightarrow{m_0 l_1} m_0 m_2' = m_0' m_2 \xrightarrow{m_0' l_2} m_0' m_3' = m_0'' m_3 \xrightarrow{m_0'' l_3} m_0''' m_4' = m_0''' m_4 \right.$$

$$\left. m_0^{,n} m_{n+1}' = m_0^{,n+1} m_{n+2} \xrightarrow{m_0^{,n+1} l_{n+2}} m_0^{,n+1} m_{n+3}' = m' \right)$$

where: $\left(m_i \xrightarrow{l_i} m_{i+1}'\right) \in \xrightarrow[\mathcal{N}]{}$, $1 \leq i \leq n+2$

Hint: Try induction on the depth of the proof tree for $m \xrightarrow{\alpha} m'$.

1. Concurrency in Petri Nets: Further Readings

José Meseguer

1. Petri net theory is a vast field with many applications. An excellent book explaining the main concepts, giving many examples, and showing how one can prove properties of Petri nets is [available electronically through Springer]:

Wolfgang Reisig, "Understanding Petri Nets", Springer-Verlag, 2013

2. The categorical semantics of the concurrent computations of a Petri net N as a strictly symmetric monoidal category $\mathcal{T}_N$ was first given in:

José Meseguer and Ugo Montanari, "Petri Nets are Monoids", Information and Computation, 88, 105-155, 1990.

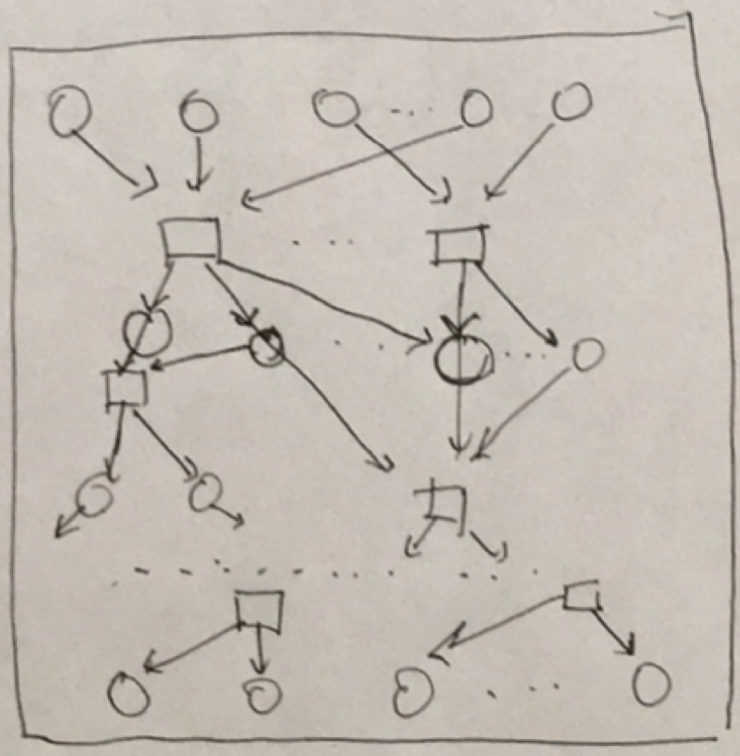
3. A logical and categorical semantics of Petri nets as theories in Girard's Linear Logic was given in:

Narciso Martí-Oliet and José Meseguer, "From Petri Nets to Linear Logic", Math. Struct. Comp. Sci., 1, 69-101, 1991.

4. A remarkable fact, showing that the notion of a Petri net computation as an arrow $u \xrightarrow{\alpha} v$ in the monoidal category $\mathcal{T}_N$ is very natural, was a proof that it essentially coincides with a partial order of events model of such computations by Best and Devillers called processes of a net, defined in:

Eike Best, Raymond Devillers, "Sequential and Concurrent Behaviour in Petri Net Theory", Theor. Comp. Sci, 55, 87-136, 1987.

Roughly speaking, a concurrent computation viewed as a process looks like:



where

○ are places

□ are transitions

The equivalence between a concurrent computation as an arrow $u \xrightarrow{\alpha} v$ in $\mathcal{T}_N$ and as a Best-Devillers process was established in:

Pierpaolo Degano, José Meseguer and Ugo Montanari, "Axiomatizing the algebra of net computations and processes," Acta Informatica, 33, 641-667, 1996.

where other, more detailed categorial models of the computations of a Petri net are defined using non-strict monoidal categories that allow permutations of tokens.

5. The Petri nets we have considered are the most basic kind, called Place/Transition nets [but further restrictions are also possible]. More expressive high-level nets such as: colored nets, and algebraic nets, where the tokens are not atomic, but can be algebraic data structures can express many more applications by finite nets. Furthermore, timed Petri nets can also express timed

computations.

A categorical semantics for all these more general Petri nets [and for the original Place/Transition nets] was given in the most general logical framework of rewriting logic [the semantic framework on which Maude is based, to be explained in upcoming lectures] by:

Mark-Oliver Stehr, José Meseguer and Peter C. Ölveczky, "Rewriting Logic as a Unified Framework for Petri Nets," in H. Ehrig et al. (Eds.), Unifying Petri Nets, Springer LNCS, 2128, 250-303, 2001.

2. Modeling Parallel Functional Programming

Functional programming describes deterministic computations, i.e., if f is a functional program and we provide input data, say, $(a_1, \ldots, a_n)$ to its arguments, then either $f(a_1, \ldots, a_n)$ does not terminate [loops], or

it terminates with a unique result, say, the data value b, so that $f(a_1, .., a_n) = b$.

The most popular notation to define functional programs in languages like Haskell, ML, and so on, is the Lambda Calculus, where a function of the form:

$$f : x \longmapsto exp(x)$$

is written as the λ-expression: $\lambda x . exp(x)$.

For example:

$$double = \lambda x . (x + x)$$

Function evaluation is accomplished by the β-reduction rule:

$$(\lambda x . t)\, u \longrightarrow t\{x \mapsto u\}$$

where t and u are meta-variables ranging over λ-expression, and $t\{x \mapsto u\}$ denotes the substitution in t of x by u. For example:

$$(\lambda x . (x + x))\, 5 \longrightarrow 5 + 5$$

Note that function application is denoted by empty syntax ——, so if t and u are λ-expressions, then $t\,u$ denotes the application of t to u, where u can also define a function, so that t may be a higher-order function.

Since name handling and substitutions are somewhat complex and tricky to implement, an even simpler calculus for functional programming without names and without quantifications about names [we view λx as a quantifier analogous to $\forall x$ for name x] is provided by the Spartan notation of Combinatory logic (CL). As a notation to define functions, CL is not human-friendly, but it is very much a machine-friendly notation. The point is that there is a translation