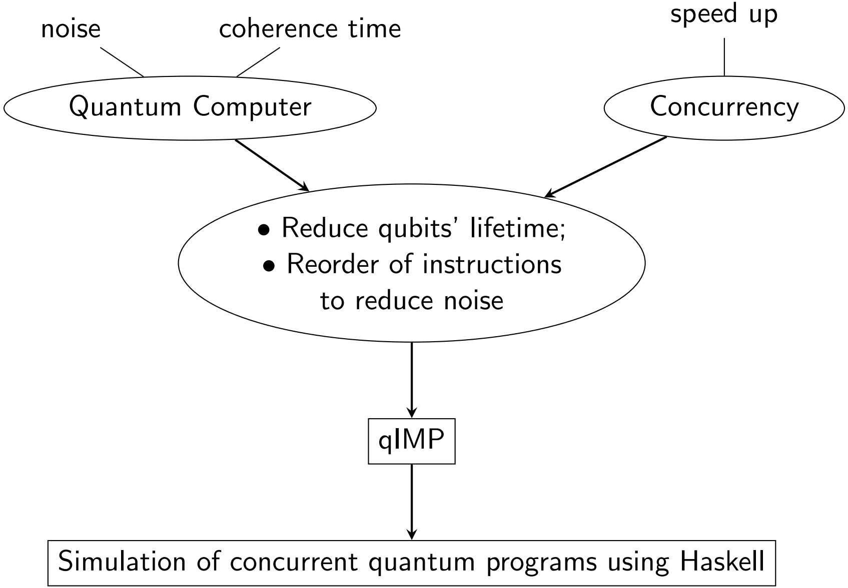
Towards Quantum Concurrency (pt1)

Inês Dias
Vitor Fernandes

INESC-TEC
University of Minho

March 3, 2023

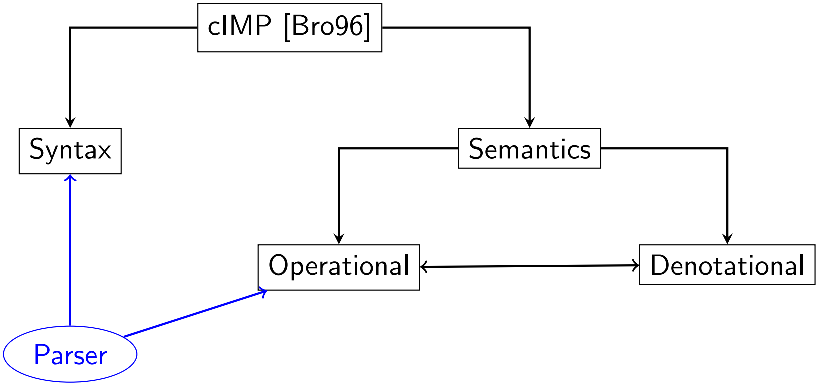
Motivation


noise
coherence time
speed up
Quantum Computer
Concurrency
• Reduce qubits' lifetime;
• Reorder of instructions to reduce noise
qIMP
Simulation of concurrent quantum programs using Haskell

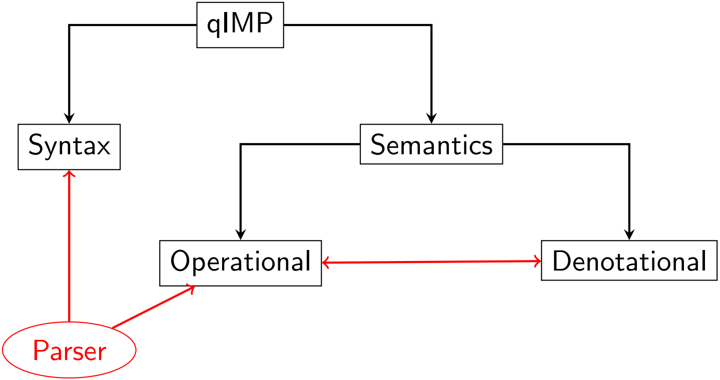
Intuitions

- Syntax
  - set of ‘words’ and operators
  - allows the construction of ‘sentences’
- Semantics
  - set of rules to evaluate ‘sentences’
  - gives meaning to the ‘sentences’
- Operational Semantics
  - ‘sentences’ as commands to be executed by a computer
- Denotational Semantics
  - ‘sentences’ as mathematical objects
- Parsing
  - checking whether a given ‘sentence’ is part of a certain language

Contextualization I



Contextualization II



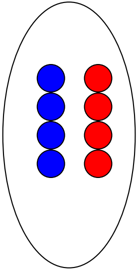
What is concurrency?

- Ability to perform different tasks at a time
  - is used to speed up processes
- Popularized within computer science in the 60s by Dijkstra [Dij65]
  - mutual exclusion
- Nowadays is ubiquitous
  - mobile apps

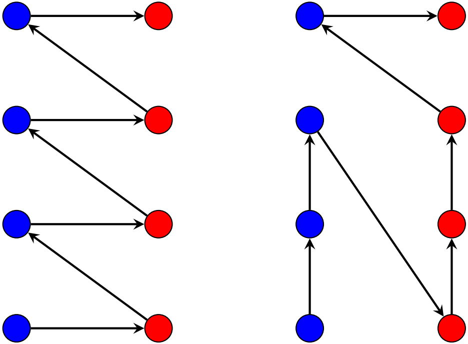
Usual Approach

Division of a task into independently small ones, which interact with each other and are performed in an interleaved way

Concurrent execution of a program



Task to be executed
divided into two sub-tasks



Examples of interleaved execution of the sub-tasks

cIMP

- Developed by Brookes [Bro96]
- Shared-variable model
- Fully abstract
  - operational and denotational semantics are equivalent

Syntax

$$B ::= tt \mid ff \mid \neg B \mid B_1 \& B_2 \mid E_1 \leq E_2$$

$$E ::= 0 \mid 1 \mid I \mid E_1 + E_2 \mid \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2$$

$$\texttt{C} ::= \texttt{skip} \mid I := E \mid \texttt{C}_1 ; \texttt{C}_2 \mid \texttt{C}_1 || \texttt{C}_2 \mid \textbf{if } B \textbf{ then } \texttt{C}_1 \textbf{ else } \texttt{C}_2 \mid \textbf{while } B \textbf{ do } \texttt{C}$$

Operational Semantics

Configuration

$\langle \texttt{C},\ s\rangle$, where C is a command and s is a state

- Booleans and expressions:
  - $\mathscr{B}[\![B]\!] = \{(s, v) \mid \langle B, s\rangle \rightarrow^* v\}$
  - $\mathscr{E}[\![E]\!] = \{(s, n) \mid \langle E, s\rangle \rightarrow^* n\}$
- Commands:

$$\frac{\langle E, s\rangle \rightarrow^* n}{\langle I := E, s\rangle \rightarrow \langle \texttt{skip}, [s \mid I = n]\rangle}$$

$$\frac{\langle \texttt{C}_1,\ s\rangle \rightarrow \langle \texttt{C}_1',\ s'\rangle}{\langle \texttt{C}_1 || \texttt{C}_2,\ s\rangle \rightarrow \langle \texttt{C}_1' || \texttt{C}_2,\ s'\rangle} \qquad \frac{\langle \texttt{C}_2,\ s\rangle \rightarrow \langle \texttt{C}_2',\ s'\rangle}{\langle \texttt{C}_1 || \texttt{C}_2,\ s\rangle \rightarrow \langle \texttt{C}_1 || \texttt{C}_2',\ s'\rangle}$$

$$\langle \textbf{while}\ \texttt{B}\ \textbf{do}\ \texttt{C},\ s\rangle \rightarrow \langle \textbf{if}\ \texttt{B}\ \textbf{then}\ \texttt{C};\textbf{while}\ \texttt{B}\ \textbf{do}\ \texttt{C}\ \textbf{else}\ \texttt{skip},\ s'\rangle$$

Example

$$\langle \texttt{x} := 1 || \texttt{x} := 2,\ [x = 0] \rangle$$

$$\langle \texttt{skip} || \texttt{x} := 2,\ [x = 1] \rangle \qquad \langle \texttt{x} := 1 || \texttt{skip},\ [x = 2] \rangle$$

$$\langle \texttt{skip} || \texttt{skip},\ [x = 2] \rangle \qquad \langle \texttt{skip} || \texttt{skip},\ [x = 1] \rangle$$

How to **implement** this language using HASKELL?

Goal

To determine the **final configuration** of a computation, given an initial configuration.

$$\boxed{\text{Initial Configuration}} \longrightarrow \boxed{\text{Final configuration}}$$

$$\langle \mathtt{a} := \mathtt{a} + 1,\ [a = 0] \rangle \qquad\qquad \langle \mathtt{skip},\ [a = 1] \rangle$$

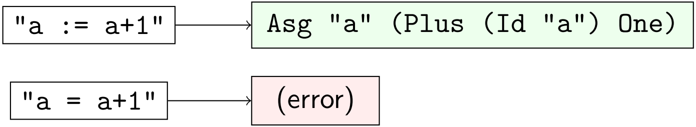
What do we need?

We need to implement: a **parser**; the **semantics**.

What do we need?

We need to implement: a **parser**; the **semantics**.

- Parser



What do we need?

We need to implement: a **parser**; the **semantics**.

- Parser

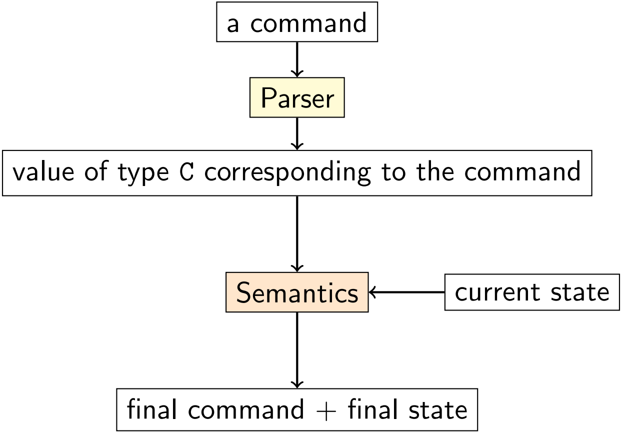
`"a := a+1"` → `Asg "a" (Plus (Id "a") One)`

`"a = a+1"` → (error)

- Semantics

command	`Asg "a" (Plus (Id "a") One)`	→	`Skip`
state	$[a{=}0]$		$[a{=}1]$

Our implementation in a nutshell



Syntax in HASKELL

`data B = BTrue | BFalse | Not B | And B B | Leq E E`

$B ::= tt \mid ff \mid \neg B \mid B_1 \& B_2 \mid E_1 \leq E_2$

A value of type B is a **boolean expression**.

`data E = Zero | One | Id String | Plus E E | IfTE_E B E E`

$E ::= 0 \mid 1 \mid I \mid E_1 + E_2 \mid \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2$

A value of type E is an **integer expression**.

`data C = Skip | Asg String E | Seq C C | Paral C C | IfTE_C B C C | WhDo B C`

$\mathtt{C} ::= \mathtt{skip} \mid I := E \mid \mathtt{C}_1 ; \mathtt{C}_2 \mid \mathtt{C}_1 || \mathtt{C}_2 \mid \textbf{if } B \textbf{ then } \mathtt{C}_1 \textbf{ else } \mathtt{C}_2 \mid \textbf{while } B \textbf{ do } \mathtt{C}$

A value of type C is a **command**.

Syntax in HASKELL (example)

data B = BTrue | BFalse | Not B | And B B | Leq E E

data E = Zero | One | Id String | Plus E E | IfTE_E B E E

data C = Skip | Asg String E | Seq C C | Paral C C | IfTE_C B C C | WhDo B C

Example of a command (a value of type C):

Asg 'a' (IfTE_E (Leq (Id 'a') (Id 'b')) Zero One)

$a := \text{if } (a \leq b) \text{ then } 0 \text{ else } 1$

Syntax in Haskell (with auxiliary data types)

```
data B = BTrue | BFalse | Not B | And B B | Leq E E

data E = Zero | One | Id String | Plus E E | IfTE_E B E E

data C = Skip | Asg String E | Seq C C | Paral C C | IfTE_C B C C |
         WhDo B C
```

Some auxiliary data types, useful for implementing the parser:

```
data BAux = BTrueAux | BFalseAux | NotAux BAux |
            AndAux BAux BAux | LeqAux E E | StrB String

data CAux = SkipAux | AsgAux String E | SeqAux CAux CAux |
            ParalAux CAux CAux | IfTE_CAux BAux CAux CAux |
            WhDoAux BAux CAux | StrC String
```

A parser for the language

- PARSEC is the HASKELL library used to implement our parsers.

Example 1: a parser for assignments (I := E)

```
pCAsg = do
    i <- parseIde
    string ':='
    e <- parseExp
    return (AsgAux i e)
```

A parser for the language

- PARSEC is the HASKELL library used to implement our parsers.

Example 1: a parser for assignments (I := E)

```
pCAsg = do
    i <- parseIde
    string ':='
    e <- parseExp
    return (AsgAux i e)
```

- Parser **parseIde** returns the parsed identifier (**i** is an identifier)

A parser for the language

- PARSEC is the HASKELL library used to implement our parsers.

Example 1: a parser for assignments (I := E)

```
pCAsg = do
    i <- parseIde
    string ':='
    e <- parseExp
    return (AsgAux i e)
```

- Parser **parseIde** returns the parsed identifier (**i** is an identifier)
- Parser **string ':='** parses the string ':='

A parser for the language

- PARSEC is the HASKELL library used to implement our parsers.

Example 1: a parser for assignments (I := E)

```
pCAsg = do
    i <- parseIde
    string ':='
    e <- parseExp
    return (AsgAux i e)
```

- Parser **parseIde** returns the parsed identifier (**i** is an identifier)
- Parser **string ':='** parses the string ':='
- Parser **parseExp** returns the parsed expression (**e** is an expression)

A parser for the language

- PARSEC is the HASKELL library used to implement our parsers.

Example 1: a parser for assignments (I := E)

```
pCAsg = do
    i <- parseIde
    string ':='
    e <- parseExp
    return (AsgAux i e)
```

- Parser **parseIde** returns the parsed identifier (**i** is an identifier)
- Parser **string ':='** parses the string ':='
- Parser **parseExp** returns the parsed expression (**e** is an expression)

Thus: parser **pCAsg** parses an assignment **I := E** and returns the corresponding value of type `CAux`.

A parser for the language

Example 2: a parser for commands

```
pC = try(pCSeq) <|> try(pCParal) <|> try(pCSkip) <|>
     try(pCAsg) <|> try(pCIf) <|> try(pCWhile) <|> pCParen
```

Each of the parsers in the definition of `pC` parses a different ‘type’ of command:

- `pCSeq` parses a **sequence of commands** (e.g. ‘`a:=1 ; b:=a`’)
- `pCParal` parses a **parallel composition of commands** (e.g. ‘`a:=1 || b:=a`’)
- `pCParen` parses a **command inside parentheses** (e.g. ‘`(skip)`’)
- ...

A parser for the language

Example 2: a parser for commands

```
pC = try(pCSeq) <|> try(pCParal) <|> try(pCSkip) <|>
     try(pCAsg) <|> try(pCIf) <|> try(pCWhile) <|> pCParen
```

- First, parser `pCSeq` is tried.

A parser for the language

Example 2: a parser for commands

```
pC = try(pCSeq) <|> try(pCParal) <|> try(pCSkip) <|>
     try(pCAsg) <|> try(pCIf) <|> try(pCWhile) <|> pCParen
```

- First, parser `pCSeq` is tried.
  - If it **succeeds** in parsing the input, `pC` returns the value returned by `pCSeq`.

A parser for the language

Example 2: a parser for commands

```
pC = try(pCSeq) <|> try(pCParal) <|> try(pCSkip) <|>
     try(pCAsg) <|> try(pCIf) <|> try(pCWhile) <|> pCParen
```

- First, parser `pCSeq` is tried.
  - If it **succeeds** in parsing the input, `pC` returns the value returned by `pCSeq`.
  - Otherwise, **pCParal** is applied, and so on.

A parser for the language

Example 2: a parser for commands

```
pC = try(pCSeq) <|> try(pCParal) <|> try(pCSkip) <|>
     try(pCAsg) <|> try(pCIf) <|> try(pCWhile) <|> pCParen
```

- First, parser `pCSeq` is tried.
  - If it **succeeds** in parsing the input, `pC` returns the value returned by `pCSeq`.
  - Otherwise, **pCParal** is applied, and so on.

In sum: parser `pC` checks if the input begins with a command and, if so, returns a value of type `CAux` corresponding to that command.

Semantics in HASKELL - using a scheduler

$$\frac{\langle \mathtt{C}_1,\ s\rangle \rightarrow \langle \mathtt{C}_1',\ s'\rangle}{\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_1' || \mathtt{C}_2,\ s'\rangle} \quad \frac{\langle \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_2',\ s'\rangle}{\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_1 || \mathtt{C}_2',\ s'\rangle}$$

- If $\langle \mathtt{C}_1,\ s\rangle$ and $\langle \mathtt{C}_2,\ s\rangle$ are not terminated configurations, there are two branches of execution for $\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle$.

Semantics in Haskell - using a scheduler

$$\frac{\langle \mathtt{C}_1,\ s\rangle \rightarrow \langle \mathtt{C}_1',\ s'\rangle}{\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_1' || \mathtt{C}_2,\ s'\rangle} \qquad \frac{\langle \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_2',\ s'\rangle}{\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_1 || \mathtt{C}_2',\ s'\rangle}$$

- If $\langle \mathtt{C}_1,\ s\rangle$ and $\langle \mathtt{C}_2,\ s\rangle$ are not terminated configurations, there are two branches of execution for $\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle$.
- Then, we need a **scheduler** to decide which branch of execution $\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle$ follows.

Semantics in HASKELL - using a scheduler

$$\langle \mathtt{C}_1 || \mathtt{C}_2,\ s \rangle \rightarrow ?$$

Semantics in HASKELL - using a scheduler

$$\langle \texttt{C}_1 || \texttt{C}_2,\ s\rangle \rightarrow ?$$

```
nextStepSch (Paral c1 c2) s = do
    x <- sched
    if (x==0) then (fst c1 c2 s) else (snd c1 c2 s)
```

- `sched` is a **scheduler**
- `x` is a pseudo-random integer (0 or 1)

Semantics in HASKELL - using a scheduler

$$\langle \texttt{C}_1 || \texttt{C}_2,\ s \rangle \to ?$$

```
nextStepSch (Paral c1 c2) s = do
    x <- sched
    if (x==0) then (fst c1 c2 s) else (snd c1 c2 s)
```

- `sched` is a **scheduler**
- `x` is a pseudo-random integer (0 or 1)
- if $\texttt{x} = 0$ then $\langle \texttt{C}'_1 || \texttt{C}_2,\ s' \rangle$ is returned:

$$\frac{\langle \texttt{C}_1,\ s \rangle \to \langle \texttt{C}'_1,\ s' \rangle}{\langle \texttt{C}_1 || \texttt{C}_2,\ s \rangle \to \langle \texttt{C}'_1 || \texttt{C}_2,\ s' \rangle}$$

Semantics in HASKELL - using a scheduler

$$\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow ?$$

```
nextStepSch (Paral c1 c2) s = do
    x <- sched
    if (x==0) then (fst c1 c2 s) else (snd c1 c2 s)
```

- `sched` is a **scheduler**
- `x` is a pseudo-random integer (0 or 1)
- if $\mathtt{x} = 0$ then $\langle \mathtt{C}'_1 || \mathtt{C}_2,\ s'\rangle$ is returned:

$$\frac{\langle \mathtt{C}_1,\ s\rangle \rightarrow \langle \mathtt{C}'_1,\ s'\rangle}{\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}'_1 || \mathtt{C}_2,\ s'\rangle}$$

- else $\langle \mathtt{C}_1 || \mathtt{C}'_2,\ s'\rangle$ is returned:

$$\frac{\langle \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}'_2,\ s'\rangle}{\langle \mathtt{C}_1 || \mathtt{C}_2,\ s\rangle \rightarrow \langle \mathtt{C}_1 || \mathtt{C}'_2,\ s'\rangle}$$

Semantics in HASKELL - example

bigStepToStr c s = list of final configurations that $\langle c,\ s\rangle$ can evolve to:

```
bigStepToStr "a:=0 || a:=1" [("a",5)] =
= [(Paral Skip Skip,[("a",1)]),(Paral Skip Skip,[("a",0)])]
```

$\langle \texttt{skip}||\texttt{skip},\ [a = 1]\rangle$ $\langle \texttt{skip}||\texttt{skip},\ [a = 0]\rangle$

Semantics in HASKELL - example

bigStepToStr c s = list of final configurations that $\langle c,\ s\rangle$ can evolve to:

```
bigStepToStr "a:=0 || a:=1" [("a",5)] =
= [(Paral Skip Skip,[("a",1)]),(Paral Skip Skip,[("a",0)])]
```

$\langle \texttt{skip}||\texttt{skip},\ [a=1]\rangle$ $\langle \texttt{skip}||\texttt{skip},\ [a=0]\rangle$

---

bigStepSchToStr c s calculates the final configuration that $\langle c,\ s\rangle$ evolves to (using a scheduler):

```
*Semantics_lingSimpl> bigStepSchToStr "a:=0 || a:=1" [("a",5)]
(Paral Skip Skip,[("a",1)])
*Semantics_lingSimpl> bigStepSchToStr "a:=0 || a:=1" [("a",5)]
(Paral Skip Skip,[("a",0)])
```

Going quantum

Syntax

$$\texttt{C} ::= \texttt{skip} \mid \texttt{U}(\tilde{\texttt{q}}) \mid \texttt{C}_1; \texttt{C}_2 \mid \texttt{C}_1 || \texttt{C}_2 \mid \texttt{M}(\texttt{q}) \rightarrow (\texttt{C}_1, \texttt{C}_2)$$

- $\texttt{skip}$: absence of action
- $\texttt{U}(\tilde{\texttt{q}})$: application of a unitary operation U to qubits $\tilde{\texttt{q}}$
- $\texttt{C}_1; \texttt{C}_2$: sequential composition of two commands
- $\texttt{C}_1 || \texttt{C}_2$: parallel composition of two commands
- $\texttt{M}(\texttt{q}) \rightarrow (\texttt{C}_1, \texttt{C}_2)$: measurement of qubit $\texttt{q}$ followed by the execution of $\texttt{C}_1$ if we read $|0\rangle$ or the execution of $\texttt{C}_2$ otherwise

Semantics

Configuration

$\langle \texttt{C},\ v \in \mathbb{C}^{2^n} \rangle$, where $\texttt{C}$ is a command and v is a unit vector in $\mathbb{C}^{2^n}$

$$\frac{\langle \texttt{C}_1,\ v \rangle \longrightarrow \sum_i p_i \cdot \langle \texttt{C}_\texttt{i},\ v_i \rangle}{\langle \texttt{C}_1 || \texttt{C}_2,\ v \rangle \longrightarrow \sum_i p_i \cdot \langle \texttt{C}_\texttt{i} || \texttt{C}_2,\ v_i \rangle} \qquad \frac{\langle \texttt{C}_2,\ v \rangle \longrightarrow \sum_j p_j \cdot \langle \texttt{C}_\texttt{j},\ v_j \rangle}{\langle \texttt{C}_1 || \texttt{C}_2,\ v \rangle \longrightarrow \sum_j p_j \cdot \langle \texttt{C}_1 || \texttt{C}_\texttt{j},\ v_j \rangle}$$

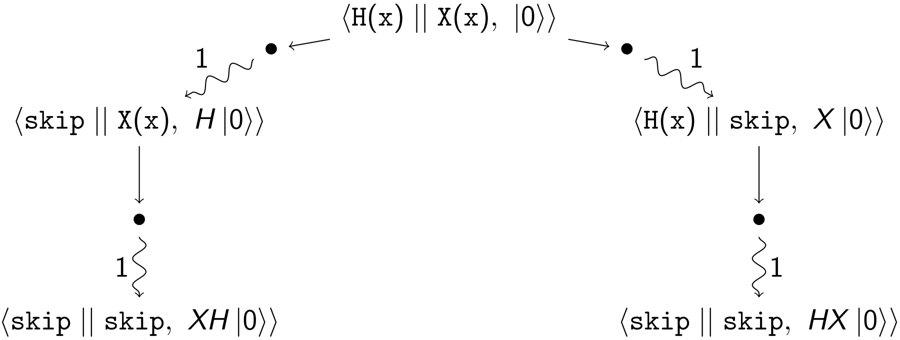
$$\langle \texttt{U}(\tilde{\texttt{q}}),\ v \rangle \longrightarrow 1 \cdot \langle \texttt{skip},\ U(\tilde{q})(v) \rangle$$

$$\langle \texttt{M}(\texttt{q}) \rightarrow (\texttt{C}_1, \texttt{C}_2),\ v \rangle \longrightarrow p_0 \cdot \langle \texttt{C}_1,\ v_0 \rangle + p_1 \cdot \langle \texttt{C}_2,\ v_1 \rangle$$

Notation

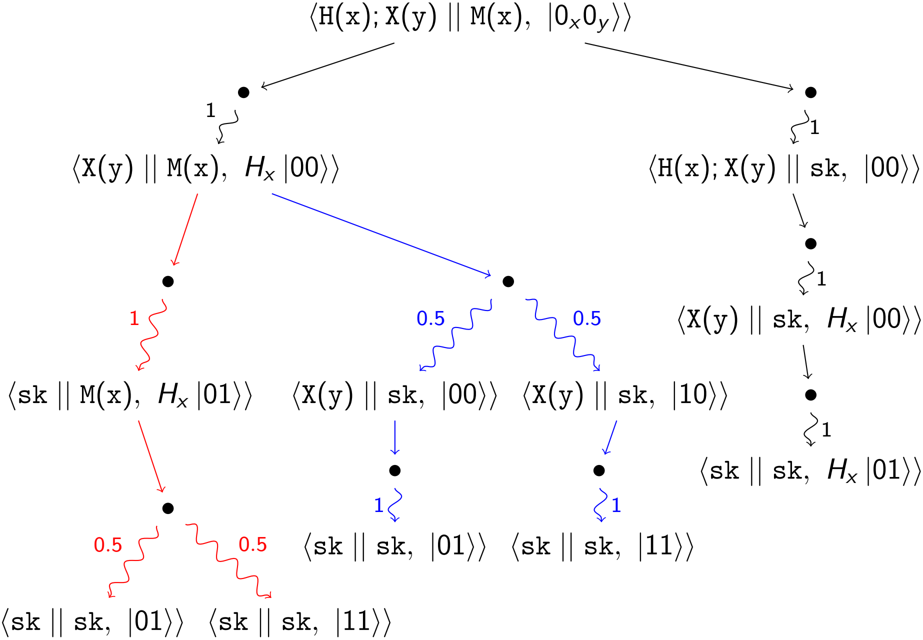
$$\langle \texttt{C},\ |0\rangle\rangle \longrightarrow \bullet \overset{p}{\rightsquigarrow} \langle \texttt{C}',\ |0\rangle\rangle$$

Example



Notation next example

- We write:
  - $\texttt{sk}$ for $\texttt{skip}$
  - $\texttt{M(q)}$ for $\texttt{M(q)} \rightarrow (\texttt{skip}, \texttt{skip})$


⟨H(x); X(y) || M(x), |0x0y⟩⟩
1
1
⟨X(y) || M(x), Hx |00⟩⟩
⟨H(x); X(y) || sk, |00⟩⟩
1
0.5
0.5
1
⟨X(y) || sk, Hx |00⟩⟩
⟨sk || M(x), Hx |01⟩⟩
⟨X(y) || sk, |00⟩⟩
⟨X(y) || sk, |10⟩⟩
1
⟨sk || sk, Hx |01⟩⟩
1
1
0.5
0.5
⟨sk || sk, |01⟩⟩
⟨sk || sk, |11⟩⟩
⟨sk || sk, |01⟩⟩
⟨sk || sk, |11⟩⟩

Bibliography I

Stephen Brookes.
Full abstraction for a shared-variable parallel language.
Information and Computation, 127(2):145–163, 1996.

Edsger W. Dijkstra.
Solution of a problem in concurrent programming control.
Commun. ACM, 8(9):569, 1965.

Thank you for your attention