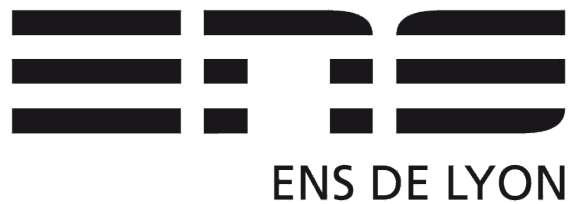
# Compilation and Program Analysis (#11) : Parallelism

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

1 Generalities on Parallelism

2 CCS: A Calculus for Communicating Systems

3 An introduction to Futures

4 Adding parallelism to Mini-while

5 A brief introduction to weak memory models

# Why parallelism?

1. To go faster
   Massive amount of computation, sometimes massively parallel, sometimes with complex parallelisation patterns
2. To handle large amount of data: big data-bases, consistency problems, synchronisation is crucial
3. To handle problems that are by nature parallel, from system interruption to online applications with several users/distributed data or decisions

# Different forms of parallelism

**Shared memory**

principle: processes can write and read data in common memory spaces example: threads in most languages generally you need a form of locking to be able to write things correctly or something similar (can be basic mutex or more complex locking like Java serialize)

**Message passing**

principle: communication between thread by sending/receiving messages several communication patterns exist synchronous/asynchronous/different send and receive primitives, etc.

**High-level programming models**

Can mix shared data and message passing or simply provide a high-level view on one of them, generally provides richer and safer way to compose computations

Example: parallel skeletons like map-reduce, actors, ...

**Parallel, concurrent, or distributed?**

1 Generalities on Parallelism

2 CCS: A Calculus for Communicating Systems

3 An introduction to Futures

4 Adding parallelism to Mini-while

5 A brief introduction to weak memory models

See on board:

- CCS: syntax of processes
- Operational Semantics as a Labelled Transition System
- On the equivalence of processes: bisimilarity

1 Generalities on Parallelism

2 CCS: A Calculus for Communicating Systems

3 An introduction to Futures
- Principles
- A case study from the literature: $\lambda$-calculus with futures

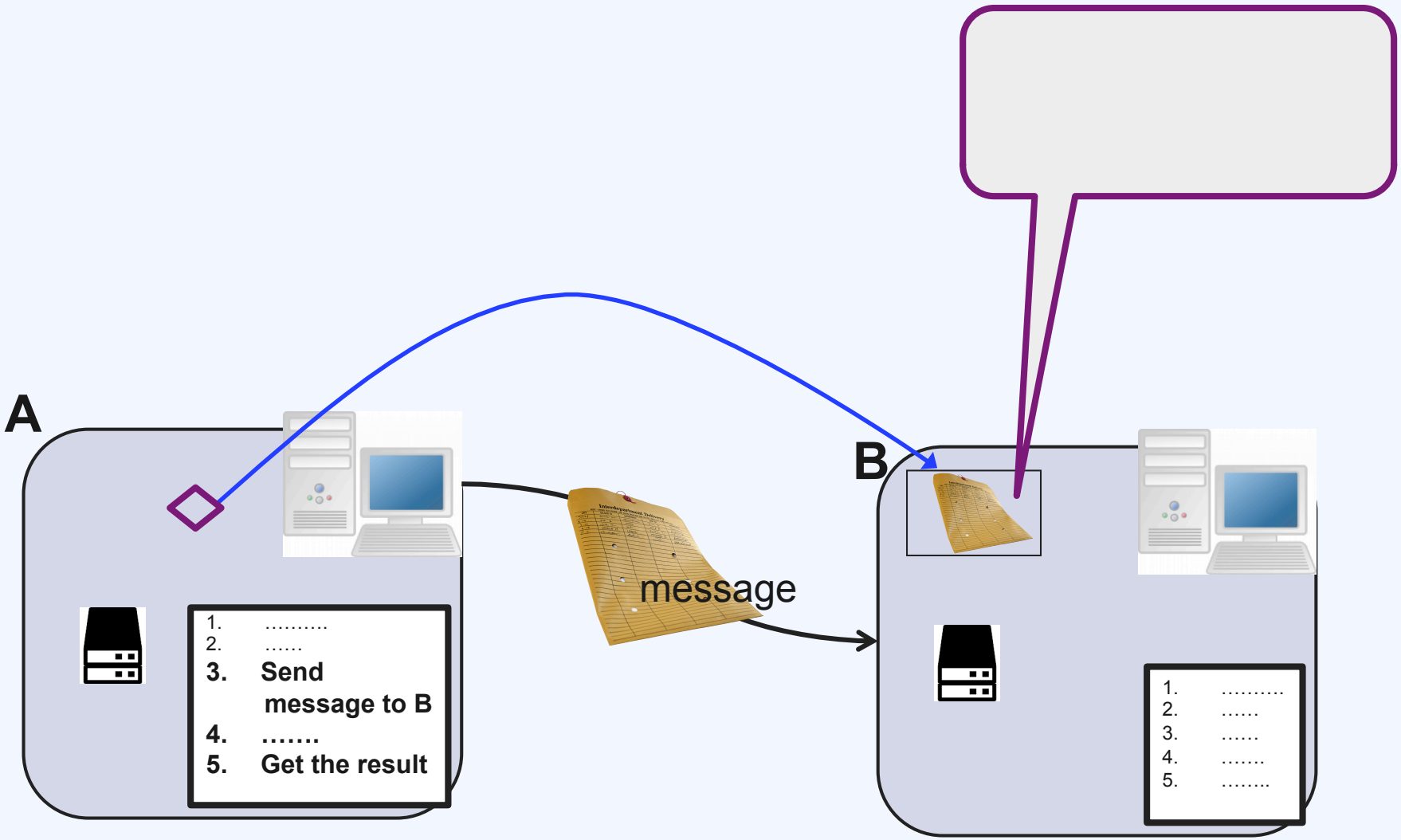
4 Adding parallelism to Mini-while

5 A brief introduction to weak memory models

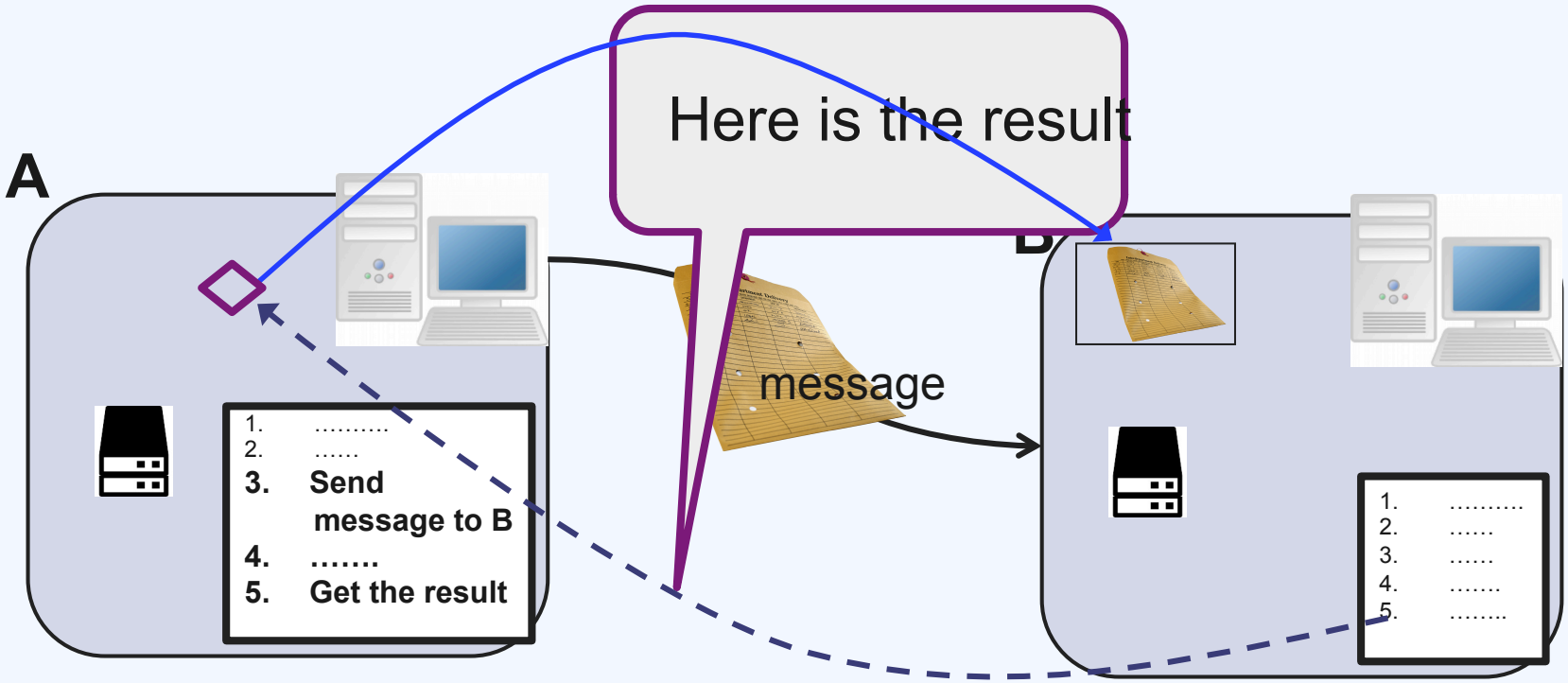
# 3 An introduction to Futures

- Principles
- A case study from the literature: $\lambda$-calculus with futures

# Requests and replies



# Requests and replies



## 3 An introduction to Futures

- Principles
- A case study from the literature: $\lambda$-calculus with futures

# A **simple** $\lambda$-calculus with futures: Syntax

Terms: $\lambda$-calculus + futures:

$$e ::= (e\ e') \mid \lambda x.\, e \mid x \mid get\ e \mid f \mid async(e)$$

$f$ appears during execution.

$v$ is a value (fully evaluated term), i.e. $v ::= f \mid \lambda x.\, e$.

We could add other values, e.g. int.

A configuration consists of

- futures: $fut(f)$ (unresolved) or $fut(f\ v)$ (resolved with a value)
- and tasks ($task(f\ \ e)$).

References:

- A more complete lambda calculus with futures can be found in: *Joachim Niehren, Jan Schwinghammer, Gert Smolka. A Concurrent Lambda Calculus with Futures. Theoretical Computer Science, 2006,*
- simple lambda calculus with futures has been used in *Fernandez-Reyes, K., Clarke, D., Castegren, E., Vo, H-P. Forward to a Promising Future. Coordination 2018*

# A **simple** $\lambda$-calculus with futures: Semantics

RED-LAMBDA

$$task(g\ E[(\lambda x.e)\ v]) \rightarrow task(g\ E[e\{v/x\}])$$

RED-ASYNC

$$\frac{\textit{fresh}\ f}{task(g\ E[async(e)]) \rightarrow fut(f)\ \ task(f\ e)\ \ task(g\ E[f])}$$

CONTEXT

$$\frac{cn \rightarrow cn'}{cn\ cn'' \rightarrow cn'\ cn''}$$

END-TASK

$$fut(f)\ task(f\ v) \rightarrow fut(f\ v)$$

RED-GET

$$task(f\ E[getf])fut(f\ v) \rightarrow task(f\ E[v])\ fut(f\ v)$$

Note: configurations identified modulo reordering of tasks / futures,

**What is E?**

# Evaluation contexts

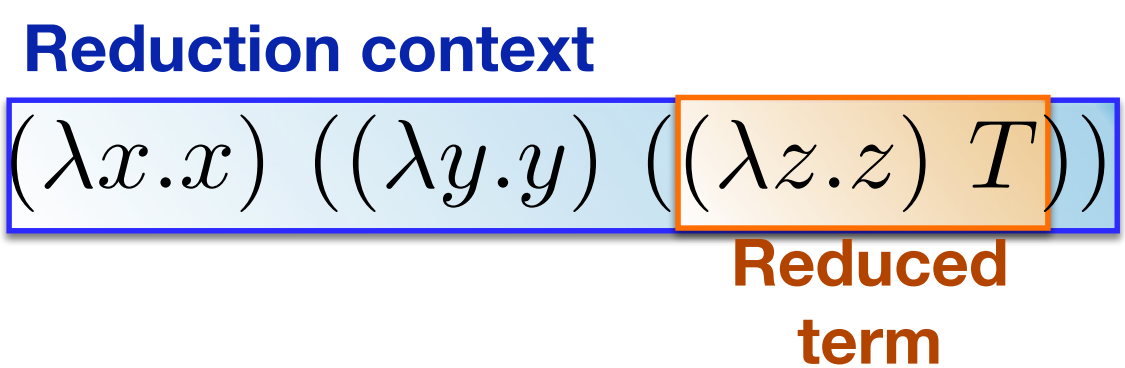
Evaluation contexts (sometimes called reduction contexts) used to focus on part of the configuration and reduce it. Compared to context rules they are more versatile: you can better choose what is in/out of the context.

For lambda-calculus with futures:

$$E ::= E\ e \mid v\ E \mid \bullet \mid get\ E$$

This ensures call-by-value.

$$E[e] = E\{\bullet \leftarrow e\}$$




$E = (\lambda x.x)\ ((\lambda y.y)\ (\bullet))$ and RED-LAMBDA can be applied.

# Example of lambda-fut evaluation

What is the initial configuration?

A task containing the program to be evaluated: $task(f\ e)$ where $e$ is the program and $f$ is a future that will never be used.

**What is the behaviour of $\left(\lambda x.\, get(async((\lambda y.\, y + y)\ x))\right)\ 3$?**

**Suppose we have a print operation in the language of the form `print "A";e`. Write a simple program that can print either first "A" then "B" or first "B" then "A". add get to the program so that only one output is possible.**

Hint: Define a term $e_A$ of the form $e_A =$ print "A"; $1$ and call it asynchronously.

Let is a classical construct, easy ro define in lambda calculus.

1 Generalities on Parallelism

2 CCS: A Calculus for Communicating Systems

3 An introduction to Futures

4 Adding parallelism to Mini-while
- Shared memory
- Asynchronous function calls and futures
- Typing futures in mini-while
- Type safety?

5 A brief introduction to weak memory models

## 4 Adding parallelism to Mini-while

- Shared memory
- Asynchronous function calls and futures
- Typing futures in mini-while
- Type safety?

# Mini-While Syntax (OLD) 1/2

Expressions:

$$\begin{array}{rcll} e & ::= & c \,|\, e + e \,|\, e \times e \,|\, ... & \\ & | & x & \textit{variable} \end{array}$$

Statements:

$$\begin{array}{rcll} S(Smt) & ::= & x := expr & \text{assign} \\ & | & x := f(e_1) & \text{simple function call} \\ & | & skip & \text{do nothing} \\ & | & S_1 ; S_2 & \text{sequence} \\ & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\ & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop} \end{array}$$

# Mini-While Syntax (OLD) 2/2

Programs with function definitions and global variables

$$
\begin{array}{rcll}
Prog & ::= & D \;\; FunDef \;\; Body & \text{Program} \\
Body & ::= & D; S & \text{Function/main body} \\
D & ::= & var\ x : \tau | D; D & \text{Variable declaration} \\
FunDef & ::= & \tau\ f(x_1 : \tau_1)\ Body; return\ e & \\
 & | & FunDef \;\; FunDef & \text{Function def}
\end{array}
$$

# Structural Op. Semantics (SOS = small step) for mini-while (OLD – no fun)

$$(x := a, \sigma) \to \sigma[x \mapsto Val(a, \sigma)]$$

$$(\texttt{skip}, \sigma) \to \sigma$$

$$\frac{(S_1, \sigma) \to \sigma'}{\big((S_1; S_2), \sigma\big) \to (S_2, \sigma')} \qquad \frac{(S_1, \sigma) \to (S_1', \sigma')}{\big((S_1; S_2), \sigma\big) \to (S_1'; S_2, \sigma')}$$

$$\frac{Val(b, \sigma) = tt}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \to (S_1, \sigma)}$$

$$\frac{Val(b, \sigma) = ff}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \to (S_2, \sigma)}$$

# Mini-while + shared memory

Add parallel composition to statements

$$S ::= ...|S||S'$$

And 2 reduction rules for parallelism:

PARALLEL1

$$\frac{(S_1, \sigma) \to (S_1', \sigma')}{(S_1||S_2, \sigma) \to (S_1'||S_2, \sigma')}$$

PARALLEL2

$$\frac{(S_2, \sigma) \to (S_2', \sigma')}{(S_1||S_2, \sigma) \to (S_1||S_2', \sigma')}$$

And 2 special cases to garbage collect finished threads:

ENDTASK1

$$(skip||S_2, \sigma) \to (S_2, \sigma')$$

ENDTASK2

$$(S_1||skip, \sigma) \to (S_1, \sigma')$$

# Example mini-while shared memory

**Compute the semantics of**:

- $x := 0; (x := 2 || \texttt{while}\ x < 3\ \texttt{do}\ x := x + 1\ \texttt{done})$

Note: $||$ is quite an impractical construct for programming. One typically prefer a spawn statement: but note that such a statement can create an unbounded amount of tasks!

## 4 Adding parallelism to Mini-while

- Shared memory
- Asynchronous function calls and futures
- Typing futures in mini-while
- Type safety?

# OLD SOS with functions (1/2)

Runtime configuration

(`Optional-Statement`, `Call-Stack`, `Stack`, `Store`):

$$cn ::= (S, Ctx, \Sigma, sto) \mid (Ctx, \Sigma, sto)$$

$$(x := e, Ctx, \Sigma, sto) \rightarrow (Ctx, \Sigma, sto[\Sigma(x) \mapsto Val(e, sto \circ \Sigma)])$$

$$\frac{(S_1, Ctx, \Sigma, sto) \rightarrow (Ctx, \Sigma', sto')}{\big((S_1; S_2), Ctx, \Sigma, sto\big) \rightarrow (S_2, Ctx, \Sigma', sto')}$$

$$\frac{(S_1, Ctx, \Sigma, sto) \rightarrow (S_1', Ctx, \Sigma', sto')}{\big((S_1; S_2), Ctx, \Sigma, sto\big) \rightarrow (S_1'; S_2, Ctx, \Sigma', sto')}$$

+ rules for if, skip, and while

# OLD SOS with functions (2/2)

CALL

$$\frac{bind_3(f, e_1..e_n, \Sigma, sto) = (S', \Sigma', sto')}{(x := f(e_1); S, Ctx, \Sigma, sto) \to (S', (\Sigma, x := R(f); S) :: Ctx, \Sigma', sto')}$$

$Ctx$ is a list of $(Stack, Stm)$. $x := f(e_1); S$ is the whole current statement (imposed by the syntax). $R(f)$ is a marker that remembers the name of the function called

$bind_3(f, e_1, \Sigma, sto) = (S_f, \Sigma', sto[\ell_1 \mapsto v_1])$ if $body(f) = D_f; S_f$,

$params(f) = [x_1]$, $Vars(D_f) = \{y_1..y_k\}$

$\ell_1$ fresh, $\ell'_1..\ell'_k$ fresh $Val(e_1, sto \circ \Sigma) = v_1$,

$\Sigma' = \Sigma[x_1 \mapsto \ell_1, y_1 \mapsto \ell'_1..y_k \mapsto \ell'_k]$.

$$\frac{v = Val(ret(f), sto \circ \Sigma')}{((\Sigma, x := R(f); S) :: Ctx, \Sigma', sto) \to (S, Ctx, \Sigma, sto[\Sigma(x) \mapsto v])}$$

# Futures: syntax and principles

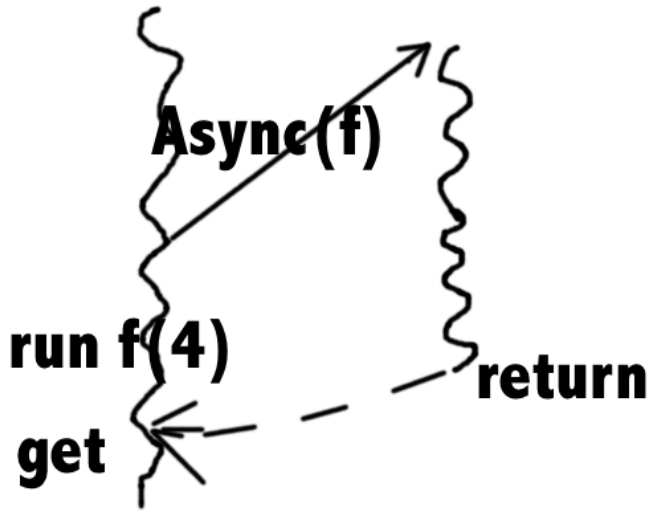
Statements:

| | | | |
|---|---|---|---|
| $S(Smt)$ | $::=$ | $x := expr$ | assign |
| | $\mid$ | $x := f(e_1)$ | simple function call |
| | $\mid$ | $x :=$ *Async*$(f(e_1))$ | Asynchronous function call |
| | $\mid$ | $x := get(e)$ | future access (synchronisation) |
| | $\mid$ | $skip$ | do nothing |
| | $\mid$ | $S_1; S_2$ | sequence |
| | $\mid$ | `if` $b$ `then` $S_1$ `else` $S_2$ | test |
| | $\mid$ | `while` $b$ `do` $S$ `done` | loop |

# Example (informally)

```
int f (int x) (
  int z;
  z:=x+x
) return z

(
  int x,y;
  fut<int> t;
  t:=Async(f(3));
  y:=f(4);
  x:=get(t)
)
```



# Design choice: no global state

We have the choice between

1. Having a global state and allow race-condition between tasks. Versatile and looks like C threads. Drawback: data-races.
2. Giving up on the global state to recover a more predictable semantics: no race between two tasks writing to the same memory.

The latter corresponds to MiniC: we have no global variable.
We specify the semantics for the second solution. To implement the first solution, a global memory should be added to the configuration.

# Future semantics for mini-while (1/3)

- Syntax: $F$, $G$ range over future identifiers. Values ($v$) can be future identifiers.
- Configurations:

$$\mathit{cn} ::= (S, Ctx, \Sigma, sto)_F \mid fut(F, v) \mid \mathit{cn} \;\; \mathit{cn}'$$

  Configurations are identified modulo reordering of tasks.
- The sequential reduction is lifted to our pool of thread straightforwardly:

$$\frac{(S, Ctx, \Sigma, sto) \to (S', Ctx', \Sigma', sto')}{(S, Ctx, \Sigma, sto)_F \; \mathit{cn} \to (S', Ctx', \Sigma', sto')_F \; \mathit{cn}}$$

# Future semantics for mini-while (2/3)

A naive solution for asynchronous function call:

ASYNC-CALL (BAD)

$$\frac{bind_3(f, e_1, \Sigma, sto) = (S', \Sigma', sto') \qquad G \text{ fresh future}}{\begin{array}{l}((x := \textit{Async}(f(e_1)); S), Ctx, \Sigma, sto)_F \ cn \rightarrow \\ (x := G; S, Ctx, \Sigma, sto)_F \ (S', \emptyset, \Sigma', sto')_G \ cn\end{array}}$$

**Problem**: we have lost the return expression that was somehow remembered by $R(f)$ in the synchronous call. We do not know what to fill the future $G$ with.

# Future semantics for mini-while (3/3)

A possible solution: we store the data in the call-stack:

$Ctx$ is hence a list of $(Stack, Stm)$, possibly with $return(e)$ as the last element of the list.

ASYNC-CALL

$$\frac{bind_3(f, e_1, \Sigma, sto) = (S', \Sigma', sto') \qquad G \text{ fresh future}}{((x := \textit{Async}(f(e_1)); S), Ctx, \Sigma, sto)_F \ cn \rightarrow ((x := G; S), Ctx, \Sigma, sto)_F \ (S', [return(ret(f))], \Sigma', sto')_G \ cn}$$

End of function execution and future access:

FUT-RESOLVE

$$\frac{\textbf{On board}}{\textbf{On board}} \rightarrow fut(F, v) \ cn$$

GET

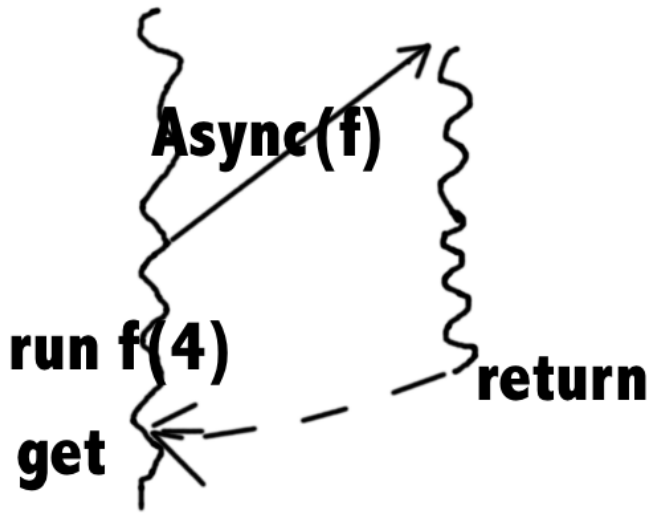
$$\frac{\textbf{On board}}{\textbf{On board}}$$

# Example (semantics)

**use the semantics to evaluate the previous example**

```
int f (int x) (
  int z;
  z:=x+x
) return z

(
  int x,y;
  fut<int> t;
  t:=Async(f(3));
  y:=f(4);
  x:=get(t)
)
```



## 4 Adding parallelism to Mini-while

- Shared memory
- Asynchronous function calls and futures
- Typing futures in mini-while
- Type safety?

# Base Type System (OLD)

From declarations we infer $\Gamma : Var \to Basetype$ with a judgment $\to_d$. From program we infer a function table $\Gamma_f : FuncName \to (\tau_1..\tau_n \to \tau)$ with a judgment $\to_f$ Then a typing judgment for expressions is $\Gamma \vdash e : \tau \in Basetype$. Typing of statements has the form : $\Gamma, \Gamma_f \vdash S$.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, \Gamma_f \vdash S_1 \quad \Gamma, \Gamma_f \vdash S_2}{\Gamma, \Gamma_f \vdash S_1; S_2}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma, \Gamma_f \vdash x := e}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma, \Gamma_f \vdash S_1 \quad \Gamma, \Gamma_f \vdash S_2}{\Gamma, \Gamma_f \vdash \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma, \Gamma_f \vdash S}{\Gamma, \Gamma_f \vdash \texttt{while } b \texttt{ do } S \texttt{ done}}$$

# Type function calls and typing program

To type a program we type all method bodies:

$$\frac{\begin{array}{c} Fundef \rightarrow_f \Gamma_f \\ \forall(\tau\ f(x_1 : \tau_1)\ D_f; S_f; return\ e \in Fundef). \\ \Gamma_g + \Gamma_l \vdash e : \tau \wedge \Gamma_l, \Gamma_f \vdash S_f \text{ with } x_1 : \tau_1; D_f \rightarrow_d \Gamma_l \\ D_m \rightarrow_d \Gamma_m \qquad \Gamma_m, \Gamma_f \vdash S \end{array}}{Fundef\ D_m; S}$$

CALL

$$\frac{\Gamma_f(f) = \tau_1 \rightarrow \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash x : \tau}{\Gamma, \Gamma_f \vdash x := f(e_1)}$$

$+$ merges and overwrite variable declarations, for overriding variables (local over global).

Note: recall there is no global variable in our current setting.

# Adding future types

Previous type syntax:

$$\tau ::= int \mid bool$$

New types can be futures:

$$\tau ::= int \mid bool \mid fut < \tau >$$

Future types can be declared: `fut<int> x,y`

**On board**: try to design rules for typing async and get.

# Typing rules for futures

**On board**

ASYNC

$$\frac{\ldots\ldots}{\Gamma, \Gamma_f \vdash x := \mathit{Async}(f(e_1))}$$

GET

$$\frac{\ldots\ldots\ldots}{\Gamma, \Gamma_f \vdash x := get(e)}$$

## 4 Adding parallelism to Mini-while

- Shared memory
- Asynchronous function calls and futures
- Typing futures in mini-while
- Type safety?

# Preservation

Definition:

consider a well typed program $Prog$, a configuration *cn* reachable by executing $Prog$, we have

$$cn \to cn' \wedge \Gamma_f, \boldsymbol{\Gamma_{fut}} \vdash cn \implies \exists \Gamma'_{fut},\ \Gamma_f, \boldsymbol{\Gamma'_{fut}} \vdash cn'$$

**What is a well-typed configuration?** i.e. define the assertion $\Gamma_f, \boldsymbol{\Gamma_{fut}} \vdash cn$ What is $\boldsymbol{\Gamma_{fut}}$?

## Definition (Configuration typing (very OLD))

$$\Gamma \vdash (S, \sigma) \iff (\Gamma \vdash S \wedge \forall x. \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau)$$

Now we have:

$$cn ::= (S, Ctx, \Sigma, sto)_F \mid fut(F, v) \mid cn\ \ cn'$$

With $Ctx$ of the form $(\Sigma, S) :: .. :: (\Sigma_n, S_n) :: return(e)$ (no $return(e)$ for the main task).

# Well-typed configuration

Suppose $\Gamma, \Gamma_f \vdash (S, \sigma)$ defined similarly to before ($\Gamma_f$ added). We can define:

$$\frac{\boldsymbol{\Gamma}, \Gamma_f \vdash (S, sto \circ \Sigma) \qquad Ctx = (\Sigma_0, S_0) :: .. :: (\Sigma_n, S_n) :: return(e) \qquad \forall i \in [1..n].\, \boldsymbol{\Gamma_i}, \Gamma_f \vdash (S_i, sto \circ \Sigma_i) \qquad \boldsymbol{\Gamma_n} \vdash e : \Gamma_{fut}(F)}{\Gamma_f, \Gamma_{fut} \vdash (S, Ctx, \Sigma, sto)_F}$$

$$\frac{\emptyset \vdash v : \Gamma_{fut}(F)}{\Gamma_f, \Gamma_{fut} \vdash fut(F, v)} \qquad \frac{\Gamma_f, \Gamma_{fut} \vdash \mathit{cn} \qquad \Gamma_f, \Gamma_{fut} \vdash \mathit{cn}'}{\Gamma_f, \Gamma_{fut} \vdash \mathit{cn}\ \ \mathit{cn}'}$$

**Problem (same as with functions):** $\boldsymbol{\Gamma}$, $\boldsymbol{\Gamma_i}$ are undefined. It is the environment that types the considered statement, i.e. the typing environment of the function that contains the considered statement. We can for example annotate configurations with the name of the function that is currently evaluated and recover the typing environment (existence of a typing environment is sufficient).

**What's a proof scheme to establish preservation?**

# On the topic of progress

**State a progress property** Note: this entails absence of deadlock

**Can we deadlock? Can we livelock? What if we have globals?**

Alternatively, we can state a weaker progress property:

> *Any well-typed configuration that cannot progress is either a final configuration or exhibits a cycle of dependencies between futures: there is a list of future identifiers such that the task responsible for computing $F_1$ is performing a get on future $F_2$, ... the task responsible for computing $F_n$ is performing a get on future $F_0$.*

In other words: typing rules out all kinds of stuck configurations, except cycles of futures.

1 Generalities on Parallelism

2 CCS: A Calculus for Communicating Systems

3 An introduction to Futures

4 Adding parallelism to Mini-while

5 A brief introduction to weak memory models

# Sequential consistency

Question: What are the possible results for the pair $(r1, r2)$ from an initial state where $(x = y = 0)$?

```
// Thread 1          ||          // Thread 2
x = 1                ||          y = 1
r1 = y               ||          r2 = x
```

# Sequential consistency

Question: What are the possible results for the pair $(r1, r2)$ from an initial state where $(x = y = 0)$?

```
// Thread 1          ||          // Thread 2
x = 1                ||          y = 1
r1 = y               ||          r2 = x
```

A sequentially consistent memory model answers as one would expect: the valid outputs are $(0, 1)$, $(1, 0)$, or $(1, 1)$.

# Sequential consistency

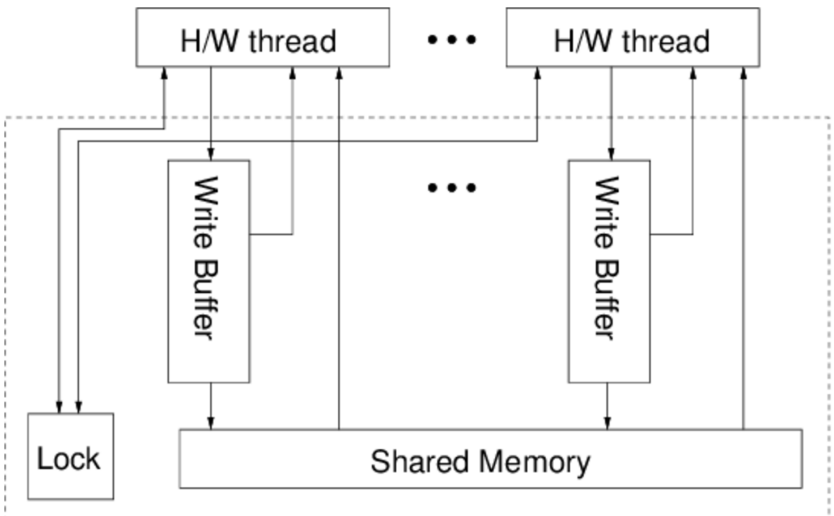
Question: What are the possible results for the pair $(r1, r2)$ from an initial state where $(x = y = 0)$?

```
// Thread 1          ||          // Thread 2
x = 1                ||          y = 1
r1 = y               ||          r2 = x
```

A sequentially consistent memory model answers as one would expect: the valid outputs are $(0, 1)$, $(1, 0)$, or $(1, 1)$.
Unfortunately, some memory models allow more behaviors: they are qualified of "weak"!

# Total Store Ordering (TSO) 1/2

Except real memories are more complex, and abstractions have (sometimes) a cost![1]




Writing to the shared memory is costly, an architecture such as x86 can't afford it. What does it mean for our memory model?

[1] Image taken from "Relaxed-Memory Concurrency and Verified Compilation".

# Total Store Ordering (TSO) 2/2

```
// Thread 1          ||          // Thread 2
x = 1                ||          y = 1
r1 = y               ||          r2 = x
```

Under a TSO model (running this example on an x86 processor for instance):

- we can observe $(0, 0)$ in the previous litmus test!
- but from a thread-local perspective, instructions appear in order.
- synchronization instructions: `fence` flushes the buffer.

# Beyond TSO: ARM, RiscV,...

These models allow some intra-thread reorderings: for instance, two consecutive stores to different locations can be reordered.[2]
Considering again initially $a = b = 0$:

thread A

```
1  store 1, @a
2  store 1, @b
```

thread B

```
1  %1 = load @b
2  %2 = load @a
```

On an ARM architecture, $(1, 0)$ is a possible outcome!

[2] Picture taken from Nicolas Chappe's PhD manuscript.

# But what about programming languages?

They could chose to be SC for ease of programming... But then the compiler will have to insert enough fences to prevent weak behaviors on the target architecture.
A very complex design point: C/C++, Java, or OCaml5 for instance all three land onto different compromises.
And compiler's intermediate representations have the toughest job of all: they must reasonably encapsulate source languages memory models, while being ready to efficiently compile to all architectures.
Modern research pieces to capture these models formally:

- The design of "A Promising Semantics for Relaxed-Memory Concurrency" (Kang et al.).
- At LIP, Nicolas Chappe's PhD defended last month on modelling llvm IR with threads.