Compilation and Program Analysis (#13) : Advanced parallelism

Ludovic Henrio

Master 1, ENS de Lyon et Dpt Info, Lyon1

2023-2024

What semantics for parallel programs

No big step semantics for parallelism

Denotational semantics difficult too because somehow big-step (see next slide)

Consequence: do small step semantics with interleaving between small steps

if P does $P \rightarrow P_1 \rightarrow P_2 \rightarrow ... \rightarrow P_n$ and Q does $Q_1 \rightarrow Q_1 \rightarrow Q_2 \rightarrow .. \rightarrow Q_n$ then $P||Q$ does the combination of the two. This is expressed by reordering processes $(P||Q \equiv Q||P)$and a simple rule:

$$\frac{P \rightarrow P'}{P||Q \rightarrow P'||Q}$$

This is most of the time sufficient but sometimes not enough, e.g. not directly adapted to weak memory models, no “true concurrency” of the form:

$$\frac{P \rightarrow P' \qquad Q \rightarrow Q'}{P||Q \rightarrow P'||Q'}$$

1. Semantics of parallelism
   - CCS: a classical comunication calculus

CCS syntax

- Channel names: a, b, c , . . .
- Co-names: $\bar{a}, \bar{b}, \bar{c}, \ldots$ (complementary « $\bar{\bar{a}}$ = a »)
- Silent action (unobservable): τ
- Actions: μ ::= $a \mid \bar{a} \mid \tau$
- Processes:

$$
\begin{array}{lcll}
P, Q & ::= & 0 & \text{inaction} \\
 & \mid & \mu.P & \text{prefix} \\
 & \mid & P \mid Q & \text{parallel} \\
 & \mid & P + Q & \text{(external) choice} \\
 & \mid & (\nu a)P & \text{restriction} \\
 & \mid & \mathrm{rec}_K P & \text{process } P \text{ with definition } K = P \\
 & \mid & K & \text{(defined) process name}
\end{array}
$$

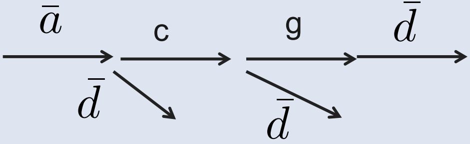
Intuitive Semantics from an "action" point of view

- a.P offers action a and then becomes P
- a.P+b.Q may offer either a and become P or b and become q
- $(\nu a)P$ may offer any action of P, except a
- P | Q may offer an action of P or of Q, but also if P offers a and Q offers a, they may synchronise (into a τ action)

We will use a labelled transition semantics for CCS processes.

A Micro-example

$$(\bar{a}.c.g)|\bar{d}$$



A tiny example

$$rec_{C1}(Tick.C1)$$

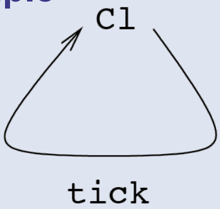


Figure: The transition graph for C1

Labelled graph

- vertices: process expressions
- labelled edges: transitions
- Each derivable transition of a vertex is depicted
- Abstract from the derivations of transitions

Exercise:
What are the possible traces (output sequences) of C1?

CCS : behavioural semantics (1)
Operators and rules

- Action prefix:

$$\frac{}{\mu.P \xrightarrow{\mu} P}$$

- Communication:

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

- Parallelism

$$\frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} \qquad \frac{Q \xrightarrow{\mu} Q'}{P|Q \xrightarrow{\mu} P|Q'}$$

CCS : behavioural semantics (2)
Operators and rules

- Non-deterministic choice

$$\frac{Q \xrightarrow{\mu} Q'}{P+Q \xrightarrow{\mu} Q'} \qquad \frac{P \xrightarrow{\mu} P'}{P+Q \xrightarrow{\mu} P'}$$

- Scope restriction

$$\frac{P \xrightarrow{\mu} P' \qquad \mu \neq a, \bar{a}}{(\nu a)P \xrightarrow{\mu} (\nu a)P'}$$

- Recursive definition

$$\frac{P[\mathrm{rec}_K P / K] \xrightarrow{\mu} P'}{\mathrm{rec}_K P \xrightarrow{\mu} P'}$$

Example

Apply semantic rules to infer one possible behaviour of:

$$\nu a.(a.b|(\overline{a}.c + \overline{a}.\overline{b}))$$

Are the following traces acceptable:

- $\tau.c.b$
- $a.\overline{a}.\tau$
- $\tau.\overline{b}.b$
- $\tau.\tau$

?

Notes on CCS and process algebras

- Different syntax exist, and plenty of variants, among them:
  - Different recursion: define process names
  - Guarded choice / guarded recursion
  - Passing data on channels ($\overline{a}(5) \mid a(x)$)
- More work on π-calculus than CCS. π-calculus is more or less CCS+sending channel names over channels.

  Scopes and restrictions become more complex. You need something like:
  $((\nu x)(P|Q)) \equiv ((\nu x)P)|Q$
- many research works on
  - equivalence relations between CCS/pi-calculus programs: Bisimulation and other equivalences.
  - typing the behaviour of programs using session types
  - security and probability aspects, etc.

Overall a vast topic

Cultural digression: About denotational semantics for parallelism

Recall: denotational semantics transforms a “program” into a “mathematical structure” It looks like big-step but it depends on the structure generated. Generating something like a trace is possible too. Trace semantics exist but is not exactly denotational. However there are ways to get something more denotational, among our recent works:

- LAGC semantics with Reiner Hahnle, Einar Broch Johnsen et al.: kind of small-step denotational by “concretizing” the semantics at some points
- itrees / ctrees with Yannick Zakowski (and others): Generate a tree (coinductive) structure similar to a trace to take into account inputs, impure effects ... and non-determinism. Coq development for reasoning on languages and compilers.

Weak memory models

Question:What are the possible results (value of `c`) for running these two threads in parallel (initially `a=b=c=0`)?

```
a=1;                    ||      b=1;
if (b==0) then                  if (a==0) then
  c++;                            c++;
```

Answer: It depends ... a lot, there are many ways to interpret the program

A memory model gives the semantics of memory accesses

Memory model – SC

<u>Sequential Consistency</u> :Each thread executes in its order, only interleaving occurs.
The granularity of atomic instructions already plays a major role (e.g. in $y = x$).

y = x	‖	x = 1
	‖	y = x

If initially $x = y = 0$, at the end $y = 1$.

Memory model – TSO

Total Store Ordering : E.g. x86, Writings are not observed immediately by other threads but locally it is consistent.

y = x	‖	x = 1
	‖	y = x

If initially $x = y = 0$, at the end $y = 0$ or $y = 1$.

Different weak memory models

- TSO is not sufficient to explain C, LLVM, ARM, etc.
- The hardware or the compiler may reorder memory operations according to rules. These rules typically state what are “independent read/writes”.
- Even single threaded programs are subject to re-ordering.
- Weak memory models allow for powerful optimisations but make programs difficult to verify/formalise.
- Example of complex and expressive memory model: Promising, Pomsets with predicate transformers

Different weak memory models

- TSO is not sufficient to explain C, LLVM, ARM, etc.
- The hardware or the compiler may reorder memory operations according to rules. These rules typically state what are “independent read/writes”.
- Even single threaded programs are subject to re-ordering.
- Weak memory models allow for powerful optimisations but make programs difficult to verify/formalise.
- Example of complex and expressive memory model: Promising, Pomsets with predicate transformers

$$\texttt{y = x} \;\Big\|\; \begin{array}{l}\texttt{z = y}\\ \texttt{x = 1}\end{array}$$

With promising, at the end, z can be 0 or 1. Promising can express C++ weak memory model.

Different approaches to implement languages

Classical compilation: as seen in course
Source-to-source compilation: compiling to assembler is tedious and restricted to one architecture. Many source-to-source compilers, e.g. language-to-C / language-to-Java
Libraries: No translation at all. Relies on software engineering expertise to implement rich libraries DSLs while staying in the restriction of the host language

Approaches can be mixed.

Domain specific language (DSL) are programming languages with a high level of abstraction, designed/optimized for a specific class of problems. They are often implemented using source-to source compilation and libraries.

Next: 2 examples.

ProActive:

A Java API + Tools for Parallel, Distributed Computing

A uniform framework: **An Active Object pattern**
A formal model behind: **Determinism (POPL'04)**

- **Programming Model (Active Objects):**
- **Asynchronous Remote Invocations, Wait-By-Necessity**
- **Groups, Mobility, Components, Security, Fault-tolerance, Load balancing**
- **Environment:**
  - **XML Deployment Descriptors, File Transfers**
  - **Interfaced with:** `rsh, ssh, LSF, PBS, Globus, Jini, SUN Grid Engine`

ObjectWeb
Open Source Middleware

Creating active objects

An object created with `A a = new A (obj, 7);`

can be turned into an active and remote object:

- **Object-based:**

```
a = (A)ProActive.turnActive (a, node);
```

- **Instantiation-based:**

```
A a = (A)ProActive.newActive(«A»,param,node);
```

`The "node" is the AO container.`

Remaining of the code unchanged → "Transparency"

Example 2: The Encore language approach

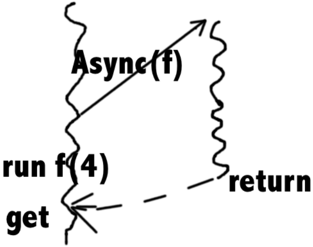
- A language with objects, futures, actors, etc. with a rich type system to optimise data access while preventing data-races.
- Many advanced features: Ownership types, parallel futures, forwarding, ...
- Compiled into C (source-to-source compilation)
- Relies on a specific C library, and an existing Actor library in C (pony) with a dedicated runtime (PonyRT) for final compilation and execution
- Tiny code example (observe the dedicated syntax):

```
defrun() : void {
    let fut = service.provide()
        client =new Client()
    in {
      client.send(get fut);
    ...
    }
}
```

On the interaction between optimisation, typing, semantics, and programming language design

Back to miniwhile with futures – 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)
)
```



Back to miniwhile with futures with a complex example

```
fut<int> foo(int x) {
 fut<int> r ;
 r=async(bar(x+1));
 return r
}
int bar(int x) { skip; return x*x }
{
 fut<int> z ; int y ;
 fut<fut<int>> x;
 x:=async(foo(2));
 y:=get(get(x));
 y:=y+1;
 z:=get(x);
}
```

Is this program well-typed? What are the possible flows of execution?

4 More on futures: Dataflow explicit futures

- Overview of future constructs
- Preliminary studies
- Dataflow explicit futures: principles
- Semantics of flows and `forward*`
- Implementation and evaluation of Flows in Encore

A few future constructs

- ABS

```
Fut<Int> f = o!add(2, 3); ❶
await f?; ❷
Int result = f.get; ❸
```

- Javascript **promises**

```
myFirstPromise.then((successMessage) => {
  console.log("Yay! " + successMessage);
});
```

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

- ProActive

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

A few future constructs

Parametric type

```
Fut<Int> f = o!add(2, 3); ❶
await f?; ❷
Int result = f.get; ❸
```

- Javascript **promises**

```
myFirstPromise.then((successMessage) => {
  console.log("Yay! " + successMessage);
});
```

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the “ask” import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

- ProActive

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

A few future constructs

Parametric type

```
Fut<Int> f = o!add(2, 3); ❶
await f?; ❷
Int result = f.get; ❸
```

Synchronous

- Javascript **promises**

```
myFirstPromise.then((successMessage) => {
  console.log("Yay! " + successMessage);
});
```

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

- ProActive

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                        null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

A few future constructs

Parametric type

- Javascript **promises**

```
Fut<Int> f = ...
await f?;
Int result = f.get;
```

Coop multithreading

```
...stPromise.then((successMessage) => {
  console.log("Yay! " + successMessage);
});
```

Synchronous

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

- ProActive

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

A few future constructs

Parametric type

- Javascript **promises**

```
Fut<Int> f
await f?;
Int result = f.get;
```

Coop multithreading

```
stPromise.then((successMessage) => {
onsole.log("Yay! " + successMessage);
});
```

Synchronous

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the “ask” import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

- ProActive

Typed as future

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                                   null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

A few future constructs

Parametric type

- Javascript **promises**

```
Fut<Int> f
await f?;
Int result = f.get;
```

```
stPromise.then((successMessage) => {
  onsole.log("Yay! " + successMessage);
});
```

Coop multithreading

Synchronous

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

Typed as future

- ProActive

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

Typed as content

A few future constructs

Parametric type

- Javascript **promises**

```
Fut<Int> f
await f?;
Int result = f.get;
```

Coop multithreading

```
stPromise.then((successMessage) => {
  onsole.log("Yay! " + successMessage);
});
```

Synchronous

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

- ProActive

Typed as future

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                                    null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

Typed as content

Transparent

A few future constructs

Parametric type

- Javascript **promises**

```
Fut<Int> f = ...
await f?;
Int result = f.get;
```

Coop multithreading

```
...stPromise.then((successMessage) => {
  console.log("Yay! " + successMessage);
});
```

Asynchronous

Synchronous

- Akka: blocking or non-blocking

```
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

Typed as future

- ProActive

```
Worker worker = (Worker) PAActiveObject.newActive(Worker.class.getName(),
                              null);//constructor arguments
Value v1 = worker.foo(); //v1 is a future
Value v2 = b.bar(v1); //v1 is passed as parameter
```

Typed as content

Transparent

Classification of futures

	(a)synchronous	Typing	Data-flow synchronisation
ABS	Coop multithreading + synchronous	Parametric type	NO
ProActive	Synchronous + WBN	Content	YES
Encore	Coop multithreading + synchronous + asynchronous (->)	Parametric type	NO
Akka	synchronous + asynchronous	future	NO
Javascript	Asynchronous	No	YES
Java	synchronous	Parametric type	NO

An example …

Classification of futures

	(a)synchronous	Typing	Data-flow synchronisation
ABS	Coop multithreading + synchronous	Parametric type	NO
ProActive	Synchronous + WBN	Content	YES
Encore	Coop multithreading + synchronous + asynchronous (->)	Parametric type	NO
Akka	synchronous + asynchronous	future	NO
Javascript	Asynchronous	No	YES
Java	synchronous	Parametric type	NO

Implicit

An example …

Classification of futures

	(a)synchronous	Typing	Data-flow synchronisation
ABS	Coop multithreading + synchronous	Parametric type	NO
ProActive	Synchronous + WBN	Content	YES
Encore	Coop multithreading + synchronous + asynchronous (->)	Parametric type	NO
Akka	synchronous + asynchronous	future	NO
Javascript	Asynchronous	No	YES
Java	synchronous	Parametric type	NO

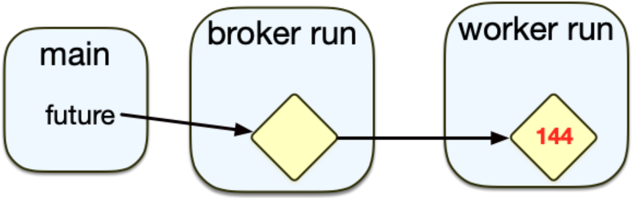
Implicit

Implicit

An example …

Introducing the problem with Future Nesting: Naive Broker

```
class Broker:
  fun run(f: int -> int, x: int): Future[int]
    let worker: Worker = select_worker()
    return async(worker.run(f, x))

fun main(): unit
  let broker: Broker = get_broker()
  let future: Future[Future[int]] = async(broker.run(fibonacci, 12))
  let result: int = get(get(future))
```

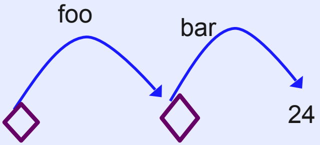


Explicit Futures à la ABS

```
main () {
  alpha=new B();
  fut<fut<int>> x=alpha.foo(2);
  y=x.get.get;
  y=y+1;
  fut<fut<int>> z=alpha.foo_fut(x.get)
}
```

explicit synchronisation BUT I should know how many futures are imbricated … typing and synchronisation

```
class B()
…
fut<int> foo(int x) {
  int t=x+1;
  fut<int> r=ANOTHERAO.bar(t);
  return r;
}
fut<int> foo_fut(fut<int> x) {
  int t=x.get;
  t=t+1;
  fut<int> r=ANOTHERAO.bar(t);
  return r;
}
```



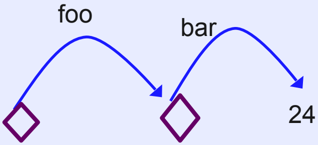
Explicit Futures à la ABS

```
main () {
  alpha=new B();
  fut<fut<int>> x=alpha.foo(2);
  y=x.get.get;
  y=y+1;
  fut<fut<int>> z=alpha.foo_fut(x.get)
}
```

explicit synchronisation BUT I should know how many futures are imbricated … typing and synchronisation

```
class B()
…
fut<int> foo(int x) {
  int t=x+1;
  fut<int> r=ANOTHERAO.bar(t);
  return r;
}
fut<int> foo_fut(fut<int> x) {
  int t=x.get;
  t=t+1;
  fut<int> r=ANOTHERAO.bar(t);
  return r;
}
```

EVEN MORE COMPLICATED

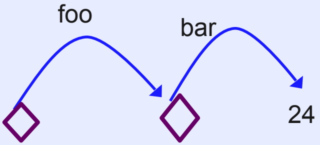


Explicit Futures à la ABS

```
main () {
  alpha=new B();
  fut<fut<int>> x=alpha.foo(2);
  y=x.get.get;
  y=y+1;
  fut<fut<int>> z=alpha.foo_fut(x.get)
}
```

explicit synchronisation BUT I should know how many futures are imbricated … typing and synchronisation

EVEN MORE COMPLICATED




```
class B()
…
fut<int> foo(int x) {
  int t=x+1;
  fut<int> r=ANOTHERAO.bar(t);
  return r;
}
fut<int> foo_fut(fut<int> x) {
  int t=x.get;
  t=t+1;
  fut<int> r=ANOTHERAO.bar(t);
  return r;
}
```

Method duplicated
(typing + synchronisation)

In ProActive: easier to write, everything hidden

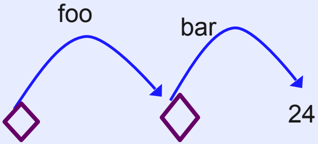
```
main () {
  alpha=new B();
  int x=alpha.foo(2);
  y=x+1;
  int z=alpha.foo(x)          }
```

transparent future access (too simple?)

No useless synchronisation

```
class B()
…
int foo(int x) {
  int t=x+1;
  int r=ANOTHERAO.bar(t);
  return r;
}
```

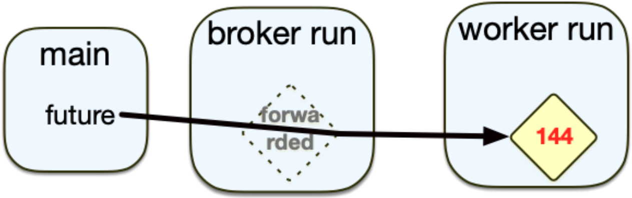
A single method



No easy way to know that there might be a synchronisation here ... and **perhaps a deadlock**

Future Nesting: Forwarding Broker [Fernandez-Reyes et al. 2018]

```
class Broker:
 fun run(f: int -> int, x: int): int
  let worker: Worker = select_worker()
  forward(worker.run(f, x)) --Delegate resolution of current future

fun main(): unit
 let broker: Broker = get_broker()
 let future: Future[int] = async(broker.run(fibonacci, 12))
 let result: int = get(future)
```



Deadlock analysis for transparent futures (with Uni Bologna)

- Behavioural types allows detecting deadlock in ABS
- Extension to transparent first-class futures is not trivial
- Because of the **data-flow** nature: an **unbound** number of method behaviours may have to be **unfolded** at the synchronization point

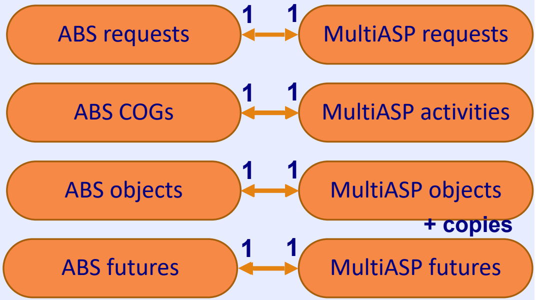
Terminal recursive functions

- We exhibit an analysis for transparent futures
  - Harder than for explicit futures
  - Even more useful as deadlocks are more difficult to find manually

ProActive backend for ABS
Using multi-active objects

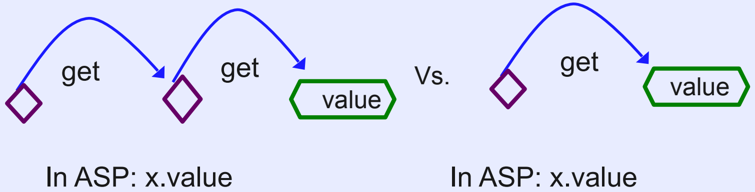
- Systematic translation of cooperative active objects into multi-threaded active objects
  - Instantiation on ABS and ProActive specifically
  - Faithful simulation
- Show the expressiveness of multiactive objects
- Show the differences btw active object languages

SHALLOW TRANSLATION



Futures: Dataflow synchronization ≠ explicit (control-flow) synchronization

- The translation simulates all possible ABS executions), *except:*
  - If a future value is a future (too strong restriction)
  - Not observable in MultiASP



- In ABS one can observe the end of a method execution, in ASP one can only observe the availability of some data

So, there are two kinds of futures: explicit or implicit

- Explicit
  - Control-flow synchronisation
  - Parametric type
  - Get (and await)
- Implicit
  - Data-flow synchronisation (wait-by-necessity)
  - No future type
  - No syntax for synchronisation

[Survey of Active Object Languages, ACM Comp Survey 2017]

Well … **NOT EXACTLY!**

A summary of problems with classical explicit futures

Godot [Fernandez-Reyes et al 2019]

Godot: All the Benefits of Explicit and Implicit Futures, Fernandez-Reyes K., Clarke D., Henrio L., Broch Johnsen E., Wrigstad T., ECOOP 2019

- The Future Type Proliferation Problem leading to the nesting of future types in case of delegated calls
- The Future Reference Proliferation Problem referring to the possibly long chain of future references that has to be followed to reach the resolved future
- The Fulfilment Observation Problem referring to the fact that the events observed with data-flow and with control-flow synchronisations are not the same

What makes the difference between future constructs?

Statically, typing makes the difference

Claim:

At runtime
Control-flow vs data-flow synchronisation

(and of course:

Synchronous vs asynchronous vs cooperative scheduling)

Idea: DeF – Explicit Futures with data-driven synchronisation

Type futures, but less strictly than in ABS

Flow<<>>

Static and Typing:

- No imbricated Flow<<>>
- It is always possible to put a A when a Flow<<A>> is expected

Runtime:

- get fetches future value until a non-future value is obtained

An example …

DeF: data-flow explicit futures

```
main () {
  alpha=new B();
  flow<<int>> x=alpha.foo(2);
  y=x.get;
  y=x+1;
  int z=alpha.foo(x)
}
```

explicit future access … but a SINGLE ACCESS!
Synchronisation visible but simple

No synchronisation here

```
class B()
…
flow<<int>> foo(flow<<int>> x) {
  int t=x.get;
  t=t+1;
  flow<<int>> r=ANOTHERAO.bar(t);
  return r;
}
```

DeF: data-flow explicit futures

```
main () {
  alpha=new B();
  flow<<int>> x=alpha.foo(2);
  y=x.get;
  y=x+1;
  int z=alpha.foo(x)
}
```

explicit future access … but a SINGLE ACCESS!
Synchronisation visible but simple

```
class B()
…
flow<<int>> foo(flow<<int>> x) {
  int t=x.get;
  t=t+1;
  flow<<int>> r=ANOTHERAO.bar(t);
  return r;
}
```

No synchronisation here

Typing of method a bit strange (avoidable if we change the typing rule for return)

DeF: data-flow explicit futures

```
main () {
  alpha=new B();
  flow<<int>> x=alpha.foo(2);
  y=x.get;
  y=x+1;
  int z=alpha.foo(x)
}
```

explicit future access … but a SINGLE ACCESS!
Synchronisation visible but simple

```
class B()
…
flow<<int>> foo(flow<<int>> x) {
  int t=x.get;
  t=t+1;
  flow<<int>> r=ANOTHERAO.bar(t);
  return r;
}
```

No synchronisation here

We declare that the method accepts a future as parameter and performs the **synchronisation if necessary** else the get does nothing

Typing of method a bit strange (avoidable if we change the typing rule for return)

DeF: data-flow explicit futures

```
main () {
  alpha=new B();
  flow<<int>> x=alpha.foo(2);
  y=x.get;
  y=x+1;
  int z=alpha.foo(x)
}
```

explicit future access … but a SINGLE ACCESS!
Synchronisation visible but simple

```
class B()
…
flow<<int>> foo(flow<<int>> x) {
  int t=x.get;
  t=t+1;
  flow<<int>> r=ANOTHERAO.bar(t);
  return r;
}
```

No synchronisation here

A single method that CAN receive a future
Code reuse

We declare that the method accepts a future as parameter and performs the **synchronisation if necessary** else the get does nothing

Typing of method a bit strange (avoidable if we change the typing rule for return)

One step further: terminal recursive asynchronous functions

In ABS

```
Int fact(Int n, Int r){
 Fut<Int>x; Int m;
 if (n==1) return r else {
   r = r*n;
   x = this.fact(n-1,r); m = await x;
   return m }}
```

Await? This method has to be unscheduled/rescheduled. **safe?**
Cooperative scheduling necessary

In ASP

```
Int fact(Int n, Int r){
 Int y;
 if (n == 1) return r else {
   r = r*n;
   y = this.fact(n-1,r); return y }}
```

Similar to sequential code, no synchronisation (**except if n is a future)**

In DeF

```
Fut«Int» fact(Int n, Int r) {
 Fut«Int» y;
  if (n == 1) return r   else {
   r = r*n;
   y = this.fact(n-1,r);  return y }}
```

Body as expected (no synchronisation)
No coop scheduling
No deleg

An implementation in Encore – explicit futures

```
active class B
  def bar(t: int): int
    t * 2
  end

  def foo(x: int): Fut[int]
    val t = x + 1
    val beta = new B()
    beta!bar(t)
  end

  --we need this function as foo
  --cannot take both fut and int
  def foo_fut(x: Fut[int]): Fut[int]
    this.foo(get(x))
  end
end

active class Main
  def main(): unit
    val alpha = new B()
    val x: Fut[Fut[int]] = alpha!foo(1)
    val y: int = get(get(x)) + 1
    val z: Fut[Fut[int]] = alpha!foo_fut(get(x))
    println(get(get(z))) --10
  end
end
```

An implementation in Encore – dataflow futures (flow)

```
active class B
 def bar(t: int): int
  t * 2
 end

 def foo(x: Flow[int]): Flow[int]
  val t = get*(x) + 1
  val beta = new B()
  beta!!bar(t)
 end
end
```

```
active class Main
 def main(): unit
  val alpha = new B()
  val x: Flow[int] = alpha!!foo(1)
       --this lifts 1 from int to Flow[int]
  var y: int = get*(x) + 1
  val z: Flow[int] = alpha!!foo(x)
  println(get*(z)) --10
 end
end
```

Synchronization (why is it called control and data flow?)

- Explicit futures

  `Future[Future[int]]` $\Rightarrow$ `get(get())`

  - Synchronization resolved by <u>end of computation</u> (control-flow)

- Dataflow explicit futures

  `Flow[int]` $\Rightarrow$ `get*()`

  - Synchronization resolved by <u>availability of data</u> (dataflow)

4 More on futures: Dataflow explicit futures

The Godot Hypothesis

The Godot Hypothesis

When working with dataflow explicit futures, `forward*` is equivalent to `return`.

Outline:

1. Semantics of `return`
2. Introduction to bisimulation
3. Semantics of `forward*` and equivalence proof

Flowing Broker – semantics

```
class Broker:
  fun run(f: int -> int, x: int):
      Flow[int]
    let worker = select_worker()
    return async*(worker.run(f, x))

fun main(): unit
  let broker: Broker = get_broker()
  let flow: Flow[int] = async*(
    broker.run(fibonacci, 12)
  )
  let result: int = get*(flow)
  println(result)
```

`flow`$_0$ (main thread)

$\boxed{\text{computing main()}}$

Flowing Broker – semantics

```
class Broker:
  fun run(f: int -> int, x: int):
      Flow[int]
    let worker = select_worker()
    return async*(worker.run(f, x))

fun main(): unit
  let broker: Broker = get_broker()
  let flow: Flow[int] = async*(
    broker.run(fibonacci, 12)
  )
  let result: int = get*(flow)
  println(result)
```

$\texttt{flow}_0$ (main thread)

$\boxed{\text{computing main()}}$

$\texttt{flow}_1$

$\boxed{\text{computing } \texttt{broker.run()}}$

Flowing Broker – semantics

```
class Broker:
  fun run(f: int -> int, x: int):
      Flow[int]
    let worker = select_worker()
    return async*(worker.run(f, x))

fun main(): unit
  let broker: Broker = get_broker()
  let flow: Flow[int] = async*(
    broker.run(fibonacci, 12)
  )
  let result: int = get*(flow)
  println(result)
```

$\texttt{flow}_0$ (main thread)

waiting for $\texttt{flow}_1$

$\texttt{flow}_1$

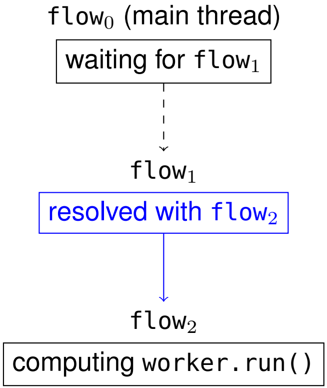
computing `broker.run()`

$\texttt{flow}_2$

computing `worker.run()`

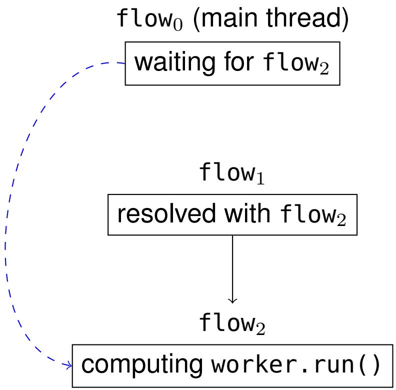
Flowing Broker – semantics

```
class Broker:
  fun run(f: int -> int, x: int):
      Flow[int]
    let worker = select_worker()
    return async*(worker.run(f, x))

fun main(): unit
  let broker: Broker = get_broker()
  let flow: Flow[int] = async*(
    broker.run(fibonacci, 12)
  )
  let result: int = get*(flow)
  println(result)
```



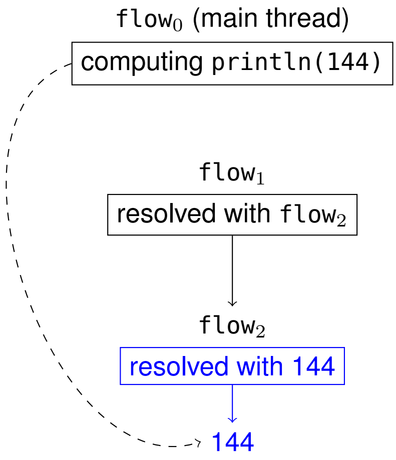
Flowing Broker – semantics

```
class Broker:
 fun run(f: int -> int, x: int):
    Flow[int]
   let worker = select_worker()
   return async∗(worker.run(f, x))

fun main(): unit
 let broker: Broker = get_broker()
 let flow: Flow[int] = async∗(
   broker.run(fibonacci, 12)
 )
 let result: int = get∗(flow)
 println(result)
```



Flowing Broker – semantics

```
class Broker:
  fun run(f: int -> int, x: int):
     Flow[int]
    let worker = select_worker()
    return async*(worker.run(f, x))

fun main(): unit
  let broker: Broker = get_broker()
  let flow: Flow[int] = async*(
    broker.run(fibonacci, 12)
  )
  let result: int = get*(flow)
  println(result)
```

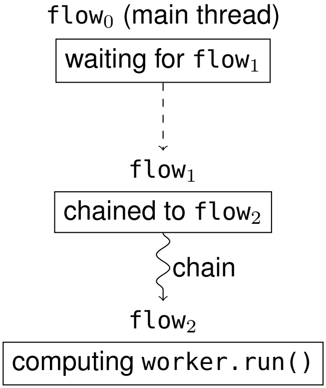


Semantics with forward*

Forward is a construct that does a shortcut (exists in Encore and illustrated above).

With dataflow futures it works more or less the same (see next slides).

```
class Broker:
 fun run(f: int -> int, x: int):
     Flow[int]
   let worker = select_worker()
   forward* async*(worker.run(f, x))

fun main(): unit
 let broker: Broker = get_broker()
 let flow: Flow[int] = async*(
   broker.run(fibonacci, 12)
 )
 let result: int = get*(flow)
 println(result)
```



Flow/return semantics

`flow`$_0$ (main thread)

waiting for `flow`$_1$

`flow`$_1$

resolved with `flow`$_2$

`flow`$_2$

computing `worker.run()`

forward* semantics

`flow`$_0$ (main thread)

waiting for `flow`$_1$

`flow`$_1$

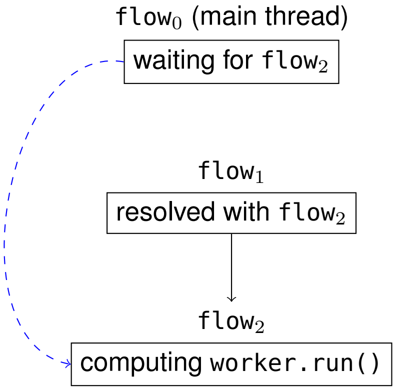
chained to `flow`$_2$

chain

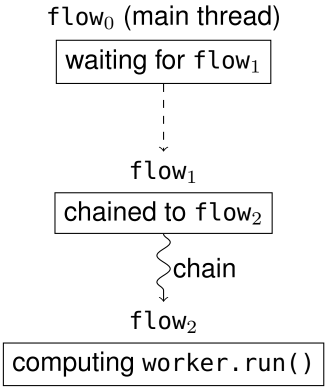
`flow`$_2$

computing `worker.run()`

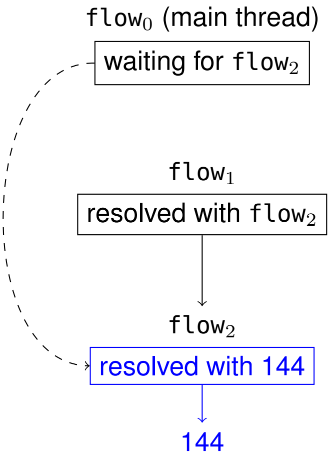
Flow/return semantics



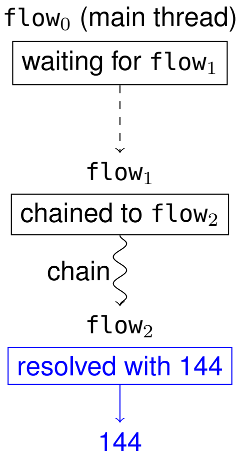
forward* semantics



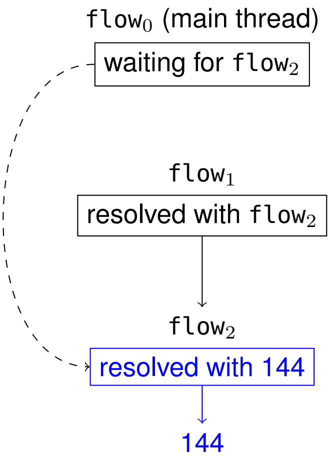
Flow/return semantics



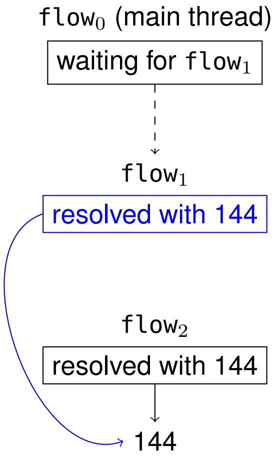
forward* semantics



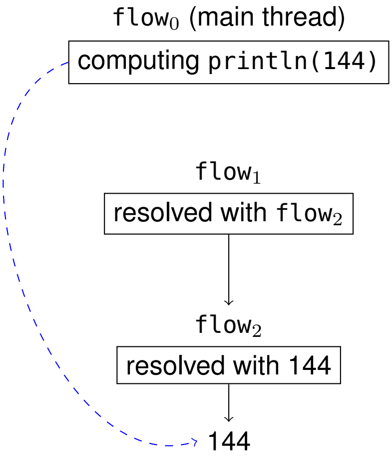
Flow/return semantics



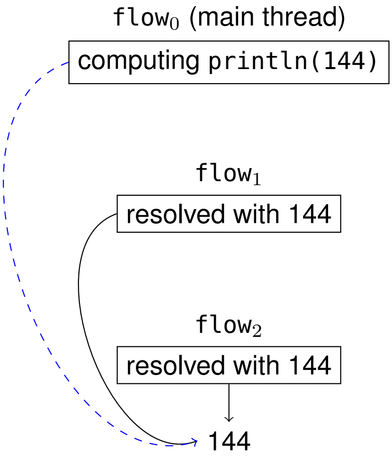
forward* semantics



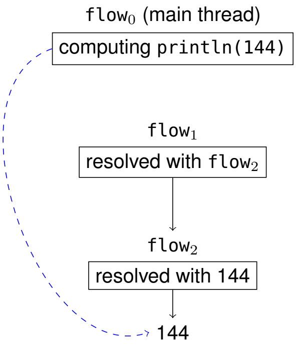
Flow/return semantics



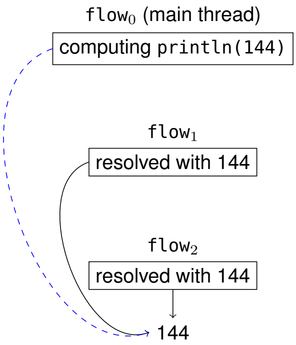
forward* semantics



Flow/return semantics



forward* semantics

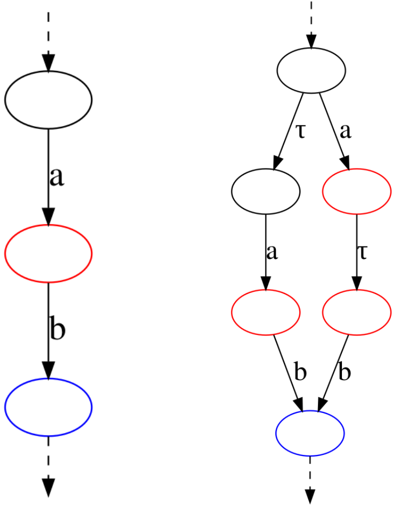


Lemma: preservation of sequences

For all sequence of flows in a program, there is a sequence of flows with the same source and same destination in this program with `forward*` replaced with `return`

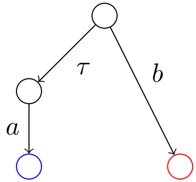
Branching bisimulation

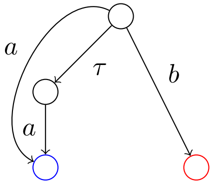
- Tool to compare semantics of transition systems based on a relation $\mathcal{R}$ between states
- Taus are non-observable internal events
- Strong > branching > weak bisimulation



Branching bisimulation (briefly)

Weak: If $s \mathcal{R} s'$ and $s \xrightarrow{\alpha} t$, there has to exist t' such that $s' \xrightarrow{\tau}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}^{*} t'$ and $t \mathcal{R} t'$.
Branching: Doing a tau stays in the same equivalence class.





No branching bisimulation here, just a weak bisimulation.

`return` and `forward*`

- We prove a branching bisimulation. Are considered τ transitions:
  - Updates of the chains in the `forward*` case
  - Updates of the `gets` in the `return` case

Theorem

When working with dataflow explicit futures, `forward*` and `return` are *observably* equivalent.

Semantic rules

$$\textsc{Assign}\quad \frac{[\![e]\!]_{a+\ell} = w \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{a \rangle F\ f(\{\ell \mid x = e\ ; s\}\#\overline{q}) \to a' \rangle F\ f(\{\ell' \mid s\}\#\overline{q})}$$

$$\textsc{Invk-Async}\quad \frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \mathrm{bind}(m, \overline{w}) = q' \qquad f' \text{ fresh}}{a \rangle F\ f(\{\ell \mid x = !\mathtt{m}(\overline{v})\ ; s\}\#\overline{q}) \to a \rangle F\ f(\{\ell \mid x = f'\ ; s\}\#\overline{q})\ f'(q')}$$

$$\textsc{Invk-Sync}\quad \frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \mathrm{bind}(m, \overline{w}) = q'}{a \rangle F\ f(\{\ell \mid x = \mathtt{m}(\overline{v})\ ; s\}\#\overline{q}) \to a \rangle F\ f(q'\#\{\ell \mid x = \mathtt{m}(\overline{v})\ ; s\}\#\overline{q})}$$

$$\textsc{Return-Async}\quad \frac{[\![v]\!]_{a+\ell} = w}{a \rangle F\ f(\{\ell \mid \mathtt{return}\ v\ ; s\}) \to a \rangle F\ f(w)}$$

$$\textsc{Return-Sync}\quad \frac{[\![v]\!]_{a+\ell'} = w}{a \rangle F\ f(\{\ell' \mid \mathtt{return}\ v\ ; s\}\#\{\ell \mid x = \mathtt{m}(\overline{v})\ ; s'\}\#\overline{q}) \to a \rangle F\ f(\{\ell \mid x = w\ ; s'\}\#\overline{q})}$$

$$\textsc{Get-Future}\quad \frac{[\![v]\!]_{a+\ell} = f'}{a \rangle F\ f(\{\ell \mid y = \mathtt{get*}\ v\ ; s\}\#\overline{q})\ f'(w') \to a \rangle F\ f(\{\ell \mid y = \mathtt{get*}\ w'\ ; s\}\#\overline{q})\ f'(w')}$$

$$\textsc{Get-Data}\quad \frac{[\![v]\!]_{a+\ell} = b}{a \rangle F\ f(\{\ell \mid y = \mathtt{get*}\ v\ ; s\}\#\overline{q}) \to a \rangle F\ f(\{\ell \mid y = b\ ; s\}\#\overline{q})}$$

Additional rules for forward∗

Forward-Async

$$\frac{[\![v]\!]_{a+\ell} = f'}{a \rangle F\ f(\{\ell \mid \texttt{forward}*\ v\ ; s\}) \to a \rangle F\ f(\texttt{chain}\ f')}$$

Forward-Sync

$$\frac{[\![v]\!]_{a+\ell} = w}{a \rangle F\ f(\{\ell \mid \texttt{forward}*\ v\ ; s\}\#q\#\overline{q}) \to a \rangle F\ f(\{\ell \mid \texttt{return}\ w\ ; s\}\#q\#\overline{q})}$$

Forward-Data

$$\frac{[\![v]\!]_{a+\ell} = b}{a \rangle F\ f(\{\ell \mid \texttt{forward}*\ v\ ; s\}) \to a \rangle F\ f(b)}$$

Chain-Update

$$\frac{}{a \rangle F\ f(\texttt{chain}\ f')\ f'(w) \to a \rangle F\ f(w)\ f'(w)}$$

4 More on futures: Dataflow explicit futures

- Overview of future constructs
- Preliminary studies
- Dataflow explicit futures: principles
- Semantics of flows and `forward*`
- Implementation and evaluation of Flows in Encore

Implementing flows

Early attempt: flows from futures

- Attempt by [Fernandez-Reyes et al, 2019] in Scala, as a library
- Mostly working, no support for parametric types (type system limitation)

Implementing flows

Early attempt: flows from futures

- Attempt by [Fernandez-Reyes et al, 2019] in Scala, as a library
- Mostly working, no support for parametric types (type system limitation)

Our implementation

- Implementation of flows in a fork of the Encore compiler
- Flows added directly in the type system, compiler modified
- Support for parametric types (except in corner cases)!

Encore and flows

- Encore already had control-flow futures and `forward`
- Active object language: future nesting is ubiquitous
- Compiler is simple: $\sim$ 20k Haskell lines

Encore and flows

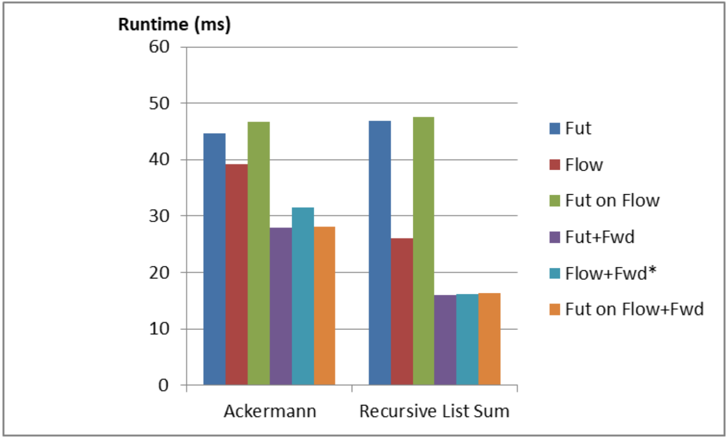
- Encore already had control-flow futures and `forward`
- Active object language: future nesting is ubiquitous
- Compiler is simple: $\sim$ 20k Haskell lines

Futures from flows

After the implementation of flows:

- Implementation of control-flow futures on top of data-flow ones
- A wrapper class prevents flows from collapsing [Fernandez-Reyes et al 2019]

Benchmarking Flow in Encore



Conclusion

Conclusion on DeF

- We proved that `forward*` and `return` are observably equivalent
  - `return` vs `forward` is just a matter of optimization with flows
- Flows are competitive with regular explicit futures
- A language with native flows can provide regular futures as a library

Today's course summary

- An introduction to CCS
- Brief introduction to weak memory models
- Brief introduction to different ways to implement languages
- Advanced futures, typing, semantics, and properties