Compilation and Program Analysis (#11) : Parallelism

Ludovic Henrio

Master 1, ENS de Lyon et Dpt Info, Lyon1

2023-2024

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

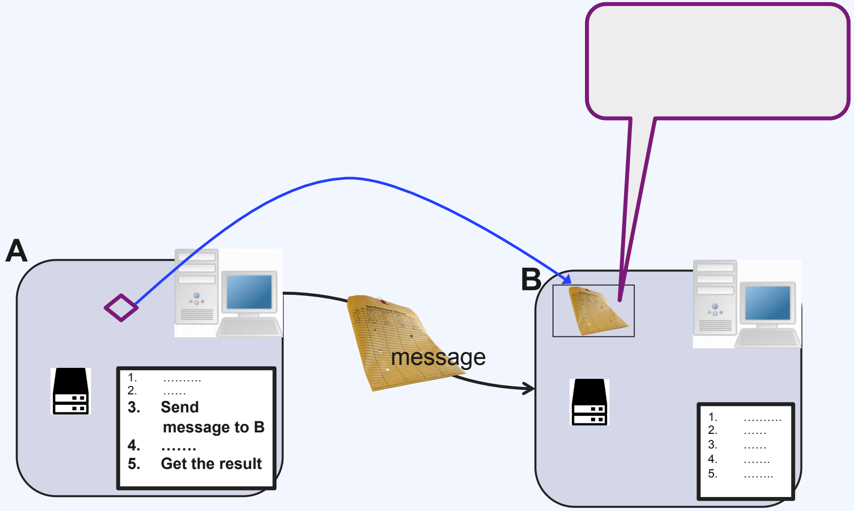
Example: parallel skeletons like map-reduce, actors, ...

Parallel, concurrent, or distributed?

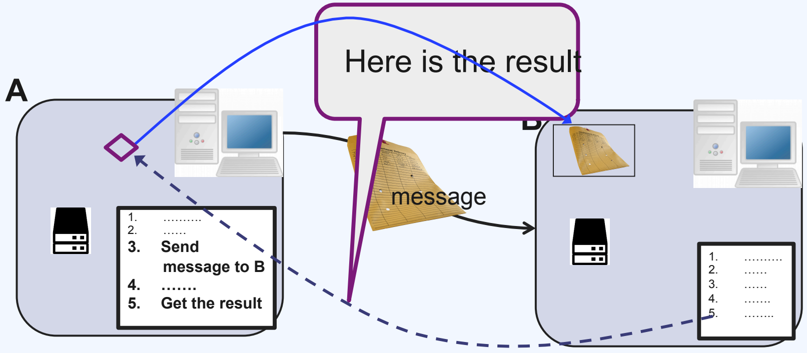
Objectives of this course

- Study and write semantics for parallel programs.
- Study one particular construct, futures, that we will implement in the practical session
- Focus on mini-while and futures.
- “Next” course we will see more general things on semantics for parallelism and more advanced features on futures.

Requests and replies



Requests and replies



A **simple** λ-calculus with futures: Syntax

Terms: λ-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** λ-calculus with futures: Semantics

RED-LAMBDA

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

RED-ASYNC

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

CONTEXT

$$\frac{cn \Rightarrow cn'}{cn\ cn'' \Rightarrow cn'\ cn''}$$

END-TASK

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

RED-GET

$$task(f\ E[get f])fut(f\ v) \Rightarrow task(f\ E[v])\ fut(f\ v)$$

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.

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

Reduction context

$$\boxed{(\lambda x.x)\ ((\lambda y.y)\ (\boxed{(\lambda z.z)\ T}))}$$

Reduced term

$E = (\lambda x.x)\ ((\lambda y.y)\ (\bullet))$ 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 $\big(\lambda x.\, get(async((\lambda y.\, y + y)\ x))\big)\ 3$?
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.

Mini-While Syntax (OLD) 1/2

Expressions:

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

Statements:

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$x := f(e_1,..,e_n)$	simple function call
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	$\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$	test
	$\mid$	$\texttt{while}\ b\ \texttt{do}\ S\ \texttt{done}$	loop

Mini-While Syntax (OLD) 2/2

Programs with function definitions and global variables

$Prog$	$::=$	$D\ \ FunDef\ \ Body$	Program
$Body$	$::=$	$D; S$	Function/main body
D	$::=$	$var\ x : \tau \vert D; D$	Variable declaration
$FunDef$	$::=$	$\tau\ f(x_1 : \tau_1, .., x_n : \tau_n)\ Body; return\ e$	
	$\vert$	$FunDef\ \ FunDef$	Function def

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

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

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

$$\frac{(S_1, \sigma) \Rightarrow \sigma'}{\big((S_1; S_2), \sigma\big) \Rightarrow (S_2, \sigma')} \quad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{\big((S_1; S_2), \sigma\big) \Rightarrow (S_1'; S_2, \sigma')}$$

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

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

Mini-while + shared memory

Add parallel composition to statements

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

And 2 reduction rules for parallelism:

PARALLEL1
$$\frac{(S_1,\sigma) \Rightarrow (S_1',\sigma')}{(S_1||S_2,\sigma) \Rightarrow (S_1'||S_2,\sigma')}$$

PARALLEL2
$$\frac{(S_2,\sigma) \Rightarrow (S_2',\sigma')}{(S_1||S_2,\sigma) \Rightarrow (S_1||S_2',\sigma')}$$

And 2 special cases when a parallel task finishes:

ENDTASK1
$$\frac{(S_1,\sigma) \Rightarrow \sigma'}{(S_1||S_2,\sigma) \Rightarrow (S_2,\sigma')}$$

ENDTASK2
$$\frac{(S_2,\sigma) \Rightarrow \sigma'}{(S_1||S_2,\sigma) \Rightarrow (S_1,\sigma')}$$

Example mini-while shared memory

Compute the semantics of:

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

A final note: $||$ is not often practical in the syntax one would prefer a spawn statement (different because spawn can create infinitely many tasks).

OLD SOS with functions (1/2)

Runtime configuration
$(\texttt{Optional-Statement}, \texttt{Call-Stack}, \texttt{Stack}, \texttt{Store})$:

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

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

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

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

+ rules for if, skip, and while

OLD SOS with functions (2/2)

CALL

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

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

$bind_3(f, e_1..e_n, \Sigma, sto) = (S_f, \Sigma', sto[\ell_1 \mapsto v_1..\ell_n \mapsto v_n])$ if $body(f) = D_f; S_f$,
$params(f) = [x_1]$, $Vars(D_f) = \{y_1..y_k\}$
ℓ_1 fresh, $\ell'_1..\ell'_k$ fresh $Val(e_1, sto \circ \Sigma) = v_1$,
$\Sigma' = \Sigma[x_1 \mapsto \ell_1, y_1 \mapsto \ell'_1..y_k \mapsto \ell'_k]$.

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

Futures: syntax and principles

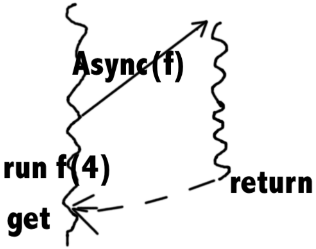
Statements:

$S(Smt)$	$::=$	$x := expr$	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. Have a global state and allow race-condition between tasks. Versatile and looks like C threads. Drawback: data-races.
2. Forget the global state that we had in function evaluation to have a more predictable semantics: no races between two tasks writing on the same memory.

The second case corresponds to MiniC where 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.
We thus suppose from now on that there is no global variable.

Future semantics for mini-while (1/3)

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

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

  Configurations are identified modulo reordering of tasks.
  Note: compared to λ-calculus+fut we do not put unresolved futures in the configuration (we could).
- Sequential reduction rules adapted straightforwardly:

$$\frac{(S, Ctx, \Sigma, sto) \Rightarrow (S', Ctx', \Sigma', sto')}{(S, Ctx, \Sigma, sto)_F\ \textit{cn} \Rightarrow (S', Ctx', \Sigma', sto')_F\ \textit{cn}}$$

$$\frac{(S, Ctx, \Sigma, sto) \Rightarrow (Ctx', \Sigma', sto')}{(S, Ctx, \Sigma, sto)_F\ \textit{cn} \Rightarrow (Ctx', \Sigma', sto')_F\ \textit{cn}}$$

Future semantics for mini-while (2/3)

A naive solution for asynchronous function call:

$$\textsc{Async-call (BAD)}\quad \frac{bind_3(f, e_1, \Sigma, sto) = (S', \Sigma', sto') \qquad G \text{ fresh future}}{\begin{array}{c}(x := \textit{Async}(f(e_1)); S, Ctx, \Sigma, sto)_F\ \textit{cn} \Rightarrow \\ (x := G; S, Ctx, \Sigma, sto)_F\ (S', \emptyset, \Sigma', sto')_G\ \textit{cn}\end{array}}$$

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: add $return(e)$ to the valid Ctx and store the returned expression in the call-stack:
Ctx is a list of $(Stack, Stm)$ possibly with $return(e)$ as the last element of the list.

$$\textsc{Async-call}\quad \frac{bind_3(f, e_1, \Sigma, sto) = (S', \Sigma', sto') \qquad G \text{ fresh future}}{\begin{array}{c}(x := \textit{\textbf{Async}}(f(e_1)); S, Ctx, \Sigma, sto)_F \ \textit{\textbf{cn}} \Rightarrow \\ (x := G; S, Ctx, \Sigma, sto)_F \ (S', return(ret(f)), \Sigma', sto')_G \ \textit{\textbf{cn}}\end{array}}$$

End of function execution and future access:

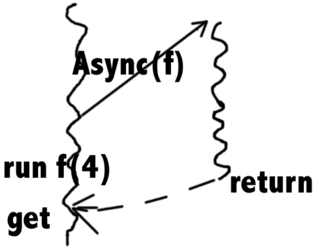
$$\textsc{Fut-resolve}\quad \frac{\textbf{On board}}{\begin{array}{c}\textbf{On board}\\ \Rightarrow fut(F, v)\ \textit{\textbf{cn}}\end{array}} \qquad\qquad \textsc{Get}\quad \frac{\textbf{On board}}{\textbf{On board}}$$

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)
)
```



Base Type System (OLD)

From declarations we infer $\Gamma : Var \rightarrow Basetype$ with a judgment $\rightarrow_d$. From program we infer a function table $\Gamma_f : FuncName \rightarrow (\tau_1..\tau_n \rightarrow \tau)$ with a judgment $\rightarrow_f$ 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:

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

CALL

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

$+$ 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`

We check structural type equivalence.

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

Typing rules for futures

On board

ASYNC

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

GET

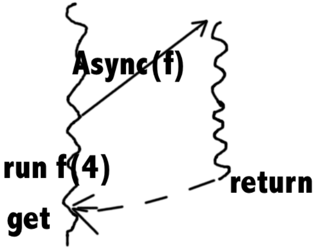
$$\frac{\ldots\ldots\ldots}{\Gamma, \Gamma_f \vdash x := get(e)}$$

Example (type)

Type the previous example (skip the typing of f)

```
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)
)
```



Preservation

Definition:

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

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

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:

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

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 (Γ_f added). We can define:

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

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

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).

Now: How to prove that small step semantics preserves well-typed configurations?

About Progress

State a progress property This entails absence of deadlock

Write a program that can deadlock

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.

formalise all this if we have the time

Structure and approach

Approach restricted to future of integers, typed as a new type `futint`. Corresponds to `fut<int>`. Functions that can be called asynchronously all have a single parameter of type `int` and return an `int`.

- An extended syntax (`get` and `async` and `futint` type) **Provided**
- A source-to-source transformation **Provided**
- A typing visitor: type `get` and `async` **To do**
- A dedicated library using C threads to implement `Async` and `Get` **To do**

Approach

A library-based approach.

- Program constructs are not implemented by the compiler, but just as a call to a dedicated library,
- We still do some program transformation, to trigger the right call to the library,
- Library could be direct system calls (here it is “just” “low-level” C library calls).

Source-to-source transformation

Done by `MiniCPPListener.py`

- add pointers, especially pointer-to-function
- Import the right library (`futurelib.h`) and add a cleanup phase

A practical example:

```
int main(){
  futint fval;
  int val;
  fval = Async(functi,123)
      ;
  println_int(0);
  return 0;
}
```

$\longrightarrow$

```
int main(){
  futint fval;
  int val;
  fval = Async(&functi
       ,123);
  println_int(0);
  freeAllFutures();
  return 0;
}
```

Typing futures

To do: add typing of async and Get in your type-checker: Your new typing visitor should now have:

```
def visitAsyncFuncCall(self, ctx):
  ...
```

Also type the instruction `Get`

```
    def visitGetCall(self, ctx):


            ...
```

What are the typing rules for Async and Get in our particular case?

The futurelib library: `futurelib.h` and `futurelib.c`

```
typedef struct {
    int Id;
    int Value;
    int resolved;
    pthread_t tid;
} FutureInt;
typedef FutureInt* futint;
FutureInt *fresh_future();
void print_futureInt(FutureInt *fut);
void free_future(FutureInt *fut);
void resolve_future(FutureInt *fut, int val);
int Get(FutureInt *fut);
FutureInt *Async(int (*fun)(int), int p);
void freeAllFutures();
```

We use a naive future dictionary (a large array of references)

Bonus: Threads in C

```
int pthread_create(pthread_t *thread, const pthread_attr_t *
    attr, void *(*start_routine) (void *), void *arg);
```

If attr is NULL, then the thread is created with default attributes.
A possible call:

```
  int err = pthread_create(&tid, NULL, &runTask, (args));
```

Notice the function pointer (explanation on board). `tid` can be associated with a future **why?**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

with a non-NULL `value_ptr` argument, the value passed to `pthread_exit()` by the terminating thread shall be made available in the location referenced by `value_ptr`.
usage:

```
  pthread_join(tid, NULL);
```

Summary: what have we seen

- Overview of concepts for parallelism and asynchronous tasks with futures.
- Designed a semantics for MiniWhile with thread and concurrent memory accesses.
- Designed a semantics for MiniWhile with futures and no concurrent memory access.
- A type system for futures
- Different ways to implement languages (beyond pure compilation solution)
- Illustrated by the extension of our MiniC language with futures

Next course is more advanced notions for parallelism, semantics and futures.