Programming Parallelism with Futures in Lustre

Albert Cohen
INRIA Paris-Rocquencourt
DI, École normale supérieure
45 rue d'Ulm, 75230 Paris
albert.cohen@inria.fr

Léonard Gérard
Univ. Paris-Sud
DI, École normale supérieure
45 rue d'Ulm, 75230 Paris
leonard.gerard@ens.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
45 rue d'Ulm, 75230 Paris
marc.pouzet@ens.fr

ABSTRACT

Efficiently distributing synchronous programs is a challenging and long-standing subject. This paper introduces the use of *futures* in a LUSTRE-like language, giving the programmer control over the expression of parallelism. In the synchronous model where computations are considered instantaneous, futures increase expressiveness by decoupling the beginning from the end of a computation.

Through a number of examples, we show how to desynchronize long computations and implement parallel patterns such as fork-join, pipelining and data parallelism. The proposed extension preserves the main static properties of the base language, including static resource bounds and the absence of deadlock, livelock and races. Moreover, we prove that adding or removing futures preserves the underlying synchronous semantics.

Categories and Subject Descriptors

C 3 [**Real-time and embedded systems**]; D 3.2 [**Data-flow languages**]; D 3.4 [**Programming languages**]: Compilers, parallelism; F 1.2 [**Parallelism and concurrency**]

General Terms

Languages, Theory, Performance

Keywords

Synchronous languages; Block-diagrams; Semantics; Parallelism; Futures; Kahn process networks

1. INTRODUCTION

Synchronous languages are devoted to the design and implementation of embedded software. They are particularly successful for safety-critical real-time systems. They facilitate the parallel modular specification and formal verification of systems to the generation of target embedded code. The synchronous model is based on the hypothesis of a logical global time scale shared by all processes which compute and communicate wich each other instantaneously. This ideal model is then validated by computing the worst case execution time (WCET) of a single reaction. Nonetheless, global logical time may be difficult to preserve when the implementation is done on a parallel machine or performance is an issue. For example, when running a rare but long duration task concurrently with a frequent and faster task, the logical time step could naively be forced to be big enough for the longest task to fit in and short enough to keep up with the frequency of the small task. The classical solution is to decouple these tasks, running the long one accross several steps. This is usually stated as the problem of long duration tasks in the litterature [10].

Several approaches have been considered in the past, always using distribution as a means to decouple the tasks, be it explicit language constructs to call external distributed functions or automatic/guided repartition techniques. The current practice of distribution is mostly manual [1] with no warranty that it preserves the functional behavior of the model. We believe that decoupling should be explicitely controlled by the programmer, *within* the synchronous language itself as a programming construct. The distribution will then be done according to this decoupling. The natural expression of decoupling is given by the notion of *future* introduced in Act1 and MultiLisp [16] and present in modern languages like C++11, Java, F#.

A future a is the promise of the result of a computation. Whereas a call to $f(x)$ couples the computation of $f(x)$ and the return of the result y, the asynchronous call $\texttt{async}f(x)$ returns instantaneously a future a. Possibly latter on, when the actual result is needed, $!a$ will block until $f(x)$ has finished and return the result y. With the help of futures, we claim that synchronous languages are fit, not only to design the control and computations, but also to *program* the decoupling and distribution.

Contribution of the Paper.

In this paper, we consider a LUSTRE-like language extended with futures and explicit asynchronous function calls. This extension is modular and conservative w.r.t. the base language, in the following sense: a sequence of input/output values of the annotated (asynchronous) program is equal to the one of the unannotated one. In other words, the annotations preserve the original synchronous semantics. The implementation handles futures as a support library. They are treated like any value of an abstract type, the get operation $!y$ is translated to the library one, and an asynchronous call $\texttt{async}f(x)$ is a matter of wrapping it inside a concurrent task, managing inputs, and dealing with the filling of

EMSOFT'12, October 7–12, 2012, Tampere, Finland.

futures. The crucial memory boundedness of synchronous programs is preserved, as well as the ability to generate efficient sequential code for each separate process resulting from the distribution. The distributed program also stays free of deadlocks, livelocks and races.

To our knowledge, the use of futures in a synchronous language is unprecedented. We show via numerous examples how to desynchronize long-running computations, how to express pipelining, fork-join, and data-parallelism patterns. This way, we achieve much higher expressiveness than coordination languages with comparable static properties [12]. In particular, the language captures arbitrary data-dependent control flow and feedback. It also highlights the important reset operator, leveraging rarely exploited sources of data-parallelism in stateful functions.

Section 2 introduces our language proposal informally. Section 3 explores its expressiveness through numerous examples. Section 4 details its formal semantics. Section 5 discusses implementation and embedded design issues. Section 6 reviews related work, before we conclude in Section 7.

2. PRESENTATION

The language used in this paper is Heptagon, a synchronous data-flow language which extends LUSTRE with static parameters, automata, arrays, and an optimizing code generator [8]. A program defines infinite streams through sets of recursive equations. Usual data-types including arrays and records are implicitly lifted to streams. It follows closely the syntax of LUSTRE. This section recalls informally the main features of the language, then introduces the new elements to desynchronize programs and enable their parallel execution. Consider, for example:

```
node sum(x:int)=(y:int)
let
  y = x + (0 fby y);
tel
```

A chronogram of `sum`:

x	0	1	0	2	4	0	−2	−8	...
y	0	1	1	3	7	7	5	−3	...

```
class Sum {
  int m_y;
  void reset() {m_y = 0;}
  int step(int x) {
    int y = x + m_y;
    m_y = y;
    return y;
  }
}
```

The Heptagon code on the left declares a node `sum` that converts an integer input stream `x` to an output stream `y`. Each sample in `y` is declared to be the sum of `x` and the stream `0 fby y`, which consists of a zero followed by the samples in `y`. This amounts to introducing an intialized register that delays the stream `y` by one cycle. The chronogram shows the beginning of `y` derived from a random `x`.

Synchronous stream programs can be compiled to scalar sequential code with internal state, such as the code on the right. Here, `reset` intitializes the internal state. Once `reset` is called, each call to `step` takes the next input sample, modifies the state, and produces the next output sample.

```
node period<< n :int | (n > 0) >> () = (c :bool)
var cpt, next_cpt :int;
let
  next_cpt = if (cpt = n) then 1 else (cpt + 1);
  cpt = 1 fby next_cpt;
  c = (cpt = 1);
tel
```

This second example declares `period` with no input, but a static parameter `n` required to be positive. It uses two local streams `cpt` and `next_cpt`. The equations are recursive and their relative order is not relevant. Usual constructs like conditional expressions (`if then else`) and comparisons (`=`) are lifted to streams by applying them pointwise. The following chronogram shows the beginning of the streams defined by the equations in `period<<3>>()`, the application of `period` with static parameter `3`:

next_cpt	2	3	1	2	3	1	2	3	...
cpt	1	2	3	1	2	3	1	2	...
c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	...

Two additional operators allow finer control on streams. `x when c` is the sampling of `x` by the Boolean stream `c`; it is the stream made of the elements of `x` for which the matching element of `c` is *true*. `merge c x1 x2` is the lazy combination of `x1` and `x2`: it produces the stream made of elements of `x1` when the matching element of `c` is *true* and `x2` when `c` is *false*. `whenot` stands for `when not`. Consider the following diagram with `x` as input. Notice that `y` equates `x`.

c	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
x	0	1	2	3	4	5	6	...
x1 = x when c	0	.	.	3	.	.	6	...
x2 = x whenot c	.	1	2	.	4	5	.	...
y = merge c x1 x2	0	1	2	3	4	5	6	...

We say that `x1` is on clock `c`, `x2` on clock `not c` and `x` on the base clock. Clock `c` is made of ticks, which are logical instants associated with the truth values of the Boolean stream `c`. We also say that the elements of `x1` are present on the ticks of its clock `c`, and absent at any other logical instant. Note that clocks are inferred by a static analysis called clock calculus. They give activation conditions and so are used to build the control flow of the generated program [5].

2.1 A Motivating Example

The following example models a classical use case for these operators. [11] The node `slow` implements an expensive operation (reduced here to a mere addition to simplify the exposition). The output is required at a higher rate than `slow` allows for. Between these precise values, interpolation is done using a fast function `fast`. Below, on the left side the source code with its chronogram, and on the right side the compilation of the node `slow_fast` into JAVA code. Note that `ys` is on clock `big` since it is the first argument of `merge`, thus `slow` and the register defining `ys` are activated only when `big` is true. `v` is on the base clock and so is updated at each tick. It contains the last value of `y`.

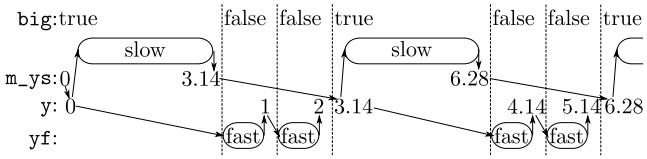
```
node fast
  (i :float) = (o :float)
let
  o = i +. 1.
tel
node slow
  (i :float) = (o :float)
let
  o = i +. 3.14
tel
node slow_fast()=(y :float)
var big :bool;
    ys, yf, v :float;
let
  big = period<<3>>();
  ys= 0. fby slow(y when big);
  yf = fast(v whenot big);
  y = merge big (ys) (yf);
  v = 0. fby y;
tel
```

big	*true*	*false*	*false*	*true*	*false*	...
ys	0.0	.	.	3.14	.	...
yf	.	1.0	2.0	.	4.14	...
y	0.0	1.0	2.0	3.14	4.14	...
v	0.0	0.0	1.0	2.0	3.14	...

```
class Slow_fast {
  Period period;
  Slow slow; Fast fast;
  float m_ys, m_v;
  Slow_fast() { /*...*/ }
  float step () {
    float y, yf;
    boolean big;
    big = period.step();
    if (big) {
      y = m_ys;
      m_ys = slow.step(y);
    } else {
      yf = fast.step(m_v);
      y = yf;
    }
    m_v = y;
    return y;
  }
  public void reset () {
    period.reset();
    slow.reset();
    fast.reset();
    m_ys = m_v = 0.f;
  }
}
```

Considering dataflow dependences, `y` depends on the value of `y` at the previous tick and of `ys` three ticks before. This should allow `slow` to last over three ticks of `fast` without slowing down the duration of an instant. Unfortunately, with the traditional compilation of LUSTRE into single-loop code, the generated code prevents ticks from overlapping, and the compilation of delayed streams like `ys` require the memory to be updated before the end of the tick. We have framed in the JAVA code the update of the memory storing the result of `slow`.

The diagram below details the consequence of this compilation strategy. It represents the computation of `slow_fast` progressing over physical time from left to right. The dashed vertical lines represent the frontiers separating ticks. Computation is depicted by the bubbles and arrows are data-dependences. Ticks are logical time, but in real-time systems this logical time is usually fixed to a constant physical duration, which here will need to be of the width of `slow` instead of `fast`.

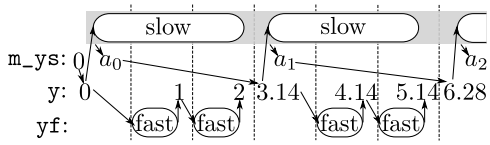


2.2 Decoupling With Futures

We introduce two additional constructs, `async` and `(!)`. `async` may be seen as a wrapper or a polymorphic higher-order operator of type: $\forall t, t'.(t \to t') \to t \to (\texttt{future}\ t')$, while `(!)` is simply $\forall t.(\texttt{future}\ t) \to t$. Moreover, a constant future holding a constant value i may be created with `async` i. Let us present our proposal on this example:

```
node a_slow_fast() = (y :float)
var big :bool; yf, v :float; ys :future float;
let
  big = period<<3>>();
  ys = (async 0.0) fby (async slow(y when big));
  yf = fast (v whenot big);
  y = merge big_step (!ys) (yf);
  v = 0.0 fby y;
tel
```

The programmer wants `slow` to compute asynchronously, so he adds `async` to the call to `slow`. The `fby` operator needs to be initialized with a value of the same type as the result of `async slow`, i.e., a `future float`. `async 0.0` is a construct returning a future holding the constant `0.0`. `ys` is consequently declared as a variable of type `future float`. The `!` operator is used to retrieve the actual value of `ys` when it is actually needed; here it is needed three ticks later to define `y`. In the diagram below, the gray box represents the wrapper in which `slow` executes concurrently. a_i are the futures returned instantly by the wrapper each time an input is given. `y` depends on these futures and on the fact that their corresponding computation is finished. This last dependence is not represented on diagrams for clarity:



Changes to the compilation result are very few: the variable `slow` is of type `Async_Slow`, `m_ys` of type `Future<Float>`, the equation `y = m_ys` is changed into `y = m_ys.get()` according to the added call to `(!)` and `m_ys` is initialized to `StaticFuture(0.f)`. The futures are provided by a library. The real changes in the compilation are found in the wrapper sketched below. It exhibits a `step` and a `reset` function as any node, but at the creation, it also spawns a worker thread. This thread is meant to execute concurrently the steps of an instance of the node `Slow`. Decoupling is achieved by using a queue `q` storing the inputs together with the corresponding future and instance of `Slow`. The decoupling is bounded by a static parameter `N` (default to 1), thanks to `q` being a blocking and bounded FIFO of size `N`. Note that the `reset` method creates a new instance of `Slow` instead of resetting it. This ease the code and presentation since otherwise caution is needed to prevent from resetting an instance still computing. It also follows the semantics presented in section 4. Static allocation of both futures and instances will be discussed in section 5.1.

```
class Async_Slow { //Sketch of the real code
  Slow instance; BoundedQueue q;
  Async_Slow(int N) { //Default N=1
    q = new BoundedQueue(N);
    new Thread(){ public void run() {
      while(true) { //Pseudocode with tuple
        (n, f, x) = q.pop();
        f.set(n.step(x));
    }}}.start(); //Spawn worker thread
  }
  Future<Float> step(float x) {
    Future<Float> f = new Future<Float>();
    q.push(instance, f, x);
    return f;
  }
  void reset() { instance = new Slow().reset(); }
}
```

3. PROGRAMMING PARALLELISM

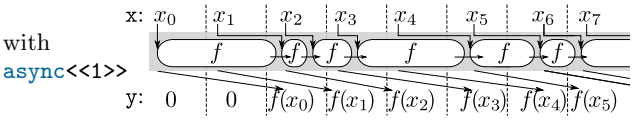
Beyond the desynchronization of the classical `slow_fast` example, futures capture a full range of concurrency patterns. This section explores this expressiveness on examples.

3.1 Jitter Smoothing With Delays

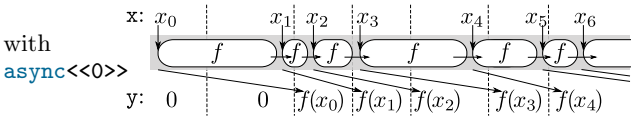
One of the elementary but crucial use of desynchronization is to smooth time-jittering computations. By adding a delay between the input and the output, one may achieve a more regular output. A decoupling of n ticks relies on two things, *data dependence:* the delay before the result is asked should at least be of n, *back pressure:* the input buffer needs to be at least of size $n-1$.

```
node smooth2<<node f(int)=(int)>> (x :int) = (y :int)
let
  y = !(async 0 fby<<2>> async<<1>> f(x));
tel
```

Here, `f` is a node given as static parameter, it is called with a decoupling of two ticks. The 1 given as static parameter to `async` is the size `N` of the input queue. The 2 given to `fby` gives a delay of 2. In the diagram below, ticks are of fixed duration. Buffering of the inputs is represented in the top part of the gray box. Notice the arrows between successive activations of f which depict the dependence created by its internal state.



With a buffer of size 0, decoupling is limited to one tick, and to keep the throughput, ticks have to be of variable duration:



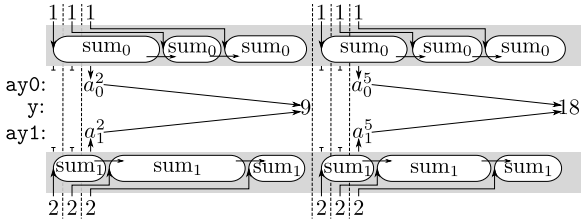
3.2 Partial Desynchronization

Synchronous programs often sample the result of a computation. This loosen partially the dependence on this computation. Running it asynchronously gives partial decoupling:

```
node partial_desync() = (y:int; c:bool)
var ay0, ay1 : future int
let
  ay0 = (async<<2>> sum(1)) when c;
  ay1 = (async<<2>> sum(2)) when c;
  c = false fby (false fby (true fby c));
  y = !ay0 + !ay1;
tel
```

sum(1)	1	2	3	4	5	6	7	8	9	...
sum(2)	2	4	6	8	10	12	14	16	18	...
c	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	...
y	.	.	9	.	.	18	.	.	27	...

Only one of every three results of each `sum` is used. For simplicity, `sum` is here a simple integrator, but it could exhibit a jittering behavior which would be smoothed out by this partial desynchronization:



3.3 Temporal Fork-Join

In stream programs, an array is often represented as the (scalar) stream of the array's elements. The following example shows the stream of arrays `x` and the associated scalar stream `lx` resulting from the flattening of the arrays in `x`.

x	$[x_0, x_1]$	.	$[x_2, x_3]$	.	$[x_4, x_5]$	.	...
lx	x_0	x_1	x_2	x_3	x_4	x_5	...

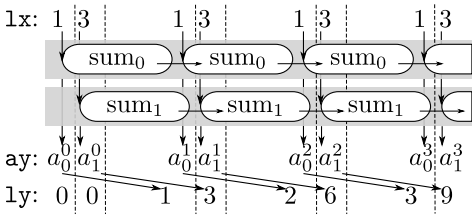
Note that `lx` is clocked at twice the rate than `x`. The temporal fork-join applies a node to a chunk of stream elements in parallel rather than to the elements of an array. Heptagon is not yet expressive enough to give a parametric version of the temporal fork-join, but here is the version of size 2:

```
node temporal_fj_2<<node f(int) = (int)>>
  (lx :int) = (ly :int)
var turn :bool; ay0, ay1, ay :future int;
let
  turn = true fby (false fby turn);
  ay0 = async f(lx when turn);
  ay1 = async f(lx whenot turn);
  ay = merge turn (ay0) (ay1);
  ly = !(async 0 fby<<2>> ay);
tel
```

`turn` alternates between *true* and *false*. Depending on `turn`, one of the two asynchronous instances of `f` is activated, `ay` is the joined output of theses instances. If we called (`!`) directly on `ay`, the instances of `f` would run sequentially, so a delay of 2 is added. Let us use this generic node with `sum` and the stream alternating the constants `1` and `3`:

```
lx = 1 fby (3 fby lx);
ly = temporal_fj_2<<sum>>(lx);
```

lx	1	3	1	3	1	3	1	3	1	3	...
turn	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	...
ly	0	0	1	3	2	6	3	9	4	12	...



For clarity, the (`!`) operator is not represented, but `ly` does wait for sum_0 to finish before getting its result; this delays the computation of the third, fifth and seventh values of `ly`.

Notice `ay` is the result of the `merge` operator applied to futures. Without futures, this stream manipulation would depend on the actual results of the computations, eliminating all parallelism. Alternatively, it is possible to retime the computations to such that the streams of actual values are merged, while preserving parallelism:

```
node retimed_tfj2<<node f(int) = (int)>>
  (lx :int) = (ly :int)
var turn :bool; ay0, ay1 :int;
let
  turn = true fby (false fby turn);
  ay0 = !(async 0 fby async f(lx when turn));
  ay1 = !(async 0 fby async f(lx whenot turn));
  ly = merge turn (ay0) (ay1);
tel
```

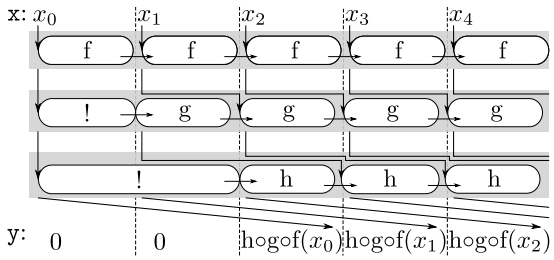
This transformation involves distributing the delay inside the `merge` branches, which is quite difficult because it depends on the understanding that `turn` defines a periodic alternation. If `turn` was more complex or statically unknown, no retiming would be possible. The ability to merge streams of futures is thus a clear progress.

3.4 Pipelining

It is possible to fully pipeline a three-node composition such as `h(g(f(x)))` assuming the result is requested with a minimal delay of 2. The easiest way to enforce this delay is to define `0 fby<<2>> h(g(f(x)))`. Since pipelined execution is a matter of chaining asynchronous computations, we define a generic combinator `pipeline_task<<f>>(ax)` which waits for the future `ax` to be ready before executing `f` on it. Here is the implementation of this combinator and its application to our three-node composition:

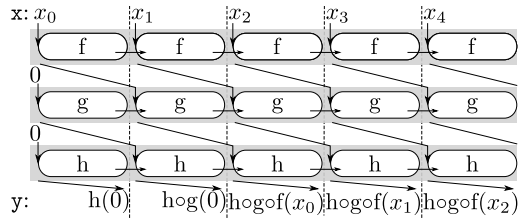
```
node pipeline_task<<node f(int) = (int)>>
  (i :future int) = (o :int)
let
  o = f (!i);
tel

node hgf(x :int) = (y :int)
var ax0, ax1, ax2 :future int;
let
  ax0 = async f (x);
  ax1 = async<<1>> pipeline_task<<g>> (ax0);
  ax2 = async<<2>> pipeline_task<<h>> (ax1);
  y = !(async 0 fby<<2>> ax2);
tel
```



Note that the input of a pipeline stage needs to be buffered for as long as its depth in the pipeline, that is why the async of `g` has a buffer of 1 and the one of `h` 2. Being able to give a future as input to an asynchronous task seems interesting, but it makes memory handling much harder (see 5.1). A simpler and more explicit version is to bufferize with a `fby` between each stage. The `fby` adds dummy values to the flow. It would be possible with clocks to prevent it, but for simplicity reasons we keep them here:

```
node retimed_hgf(x :int) = (y :int)
var ax0, ax1, ax2 :future int;
let
  ax0 = async f (x);
  ax1 = async g (!(async 0 fby ax0));
  ax2 = async h (!(async 0 fby ax1));
  y = !ax2;
tel
```

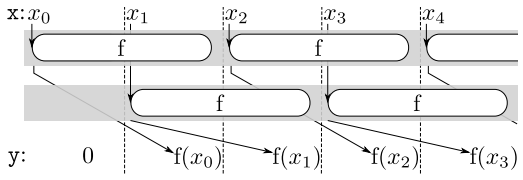


3.5 Stateless Data-Parallelism

In a dataflow language like ours, data-parallelism is the possibility to compute several successive values of a flow at the same time. For example with the code `y = f(x)`, computing $y_0 = f_0(x_0)$ and $y_1 = f_1(x_1)$ at the same time. When `f` is stateful, it is impossible since f_1 needs the state of `f` resulting from the computation of $f_0(x_0)$. But it is not a problem when `f` is a pure function.

The example below asks for a stateless function `f` as parameter (using the keyword `fun`). At each tick i, `async f(x)` being stateless, a new task may be created to compute $f(x_i)$. To keep resources bounded, the number of concurrent tasks created by a given `async` is by default one and may be specified as second static parameter of `async`. Here we allocate two tasks, each with an input buffer of 0:

```
node data_parallel_2<<fun f(int) = (int)>>
  (x :int) = (y :int)
let
  y = !(async 0 fby async<<0,2>> f(x));
tel
```



3.6 Data-Parallelism From Resetting

Consider `lx` a scalar stream resulting from the flattening of a stream of arrays of size `n`. It is often the case that we would like to apply a node `f` on each element of the array and to reset it at every beginning of the flattened array. To do so we define a Boolean stream `r` which is *true* every `n` ticks, and we reset the application of `f` every `r`. Computing in parallel on each element of a chunk as was done in the temporal fork-join example is impossible since `f` is stateful, but computing `f` on `m` successive chunks at the same time is possible:

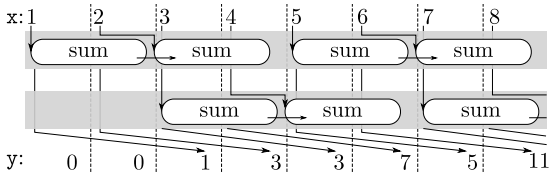
```
node array_dp<<node f(int) = (int); n, m :int>>
  (lx :int) = (ly :int)
var new :bool;
let
  r = period<<n>>();
  reset
    ay = async<<n-1, m>> f(lx);
  every r;
  ly = ! (async 0 fby<<n*(m-1)>> ay);
tel
```

Data-parallel execution requires that each set of computations on a chunk should not prevent the next one to begin. This is ensured by setting an input buffer of size `n-1` for each set of computations on a chunk. In the same way, the results of the first computations should not be requested before another chunk is fed in to the `m`-th task. This requires a delay of `n*(m-1)` on the output. To illustrate this generic node, we can apply it to `sum` with arrays of size 2 flattened into chunks, and with 2 tasks:

```
lx = 1 + (0 fby lx);
y = array_dp<<sum, 2, 2>>(lx);
```

Which gives the following chronogram and diagram:

x	1	2	3	4	5	6	7	8	9	10	...
r	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*	*false*	...
y	0	0	1	3	3	7	5	11	7	15	...



Stateless parallelism may be seen as a case of stateful parallelism reset at every tick. In the rest of the paper we only consider stateful parallelism. The wrapper with two static arguments `async<<N,T>>` has to spawn `T` threads and allocate an input queue for each. At each reset, the sequential dependence is cut and it may use another worker thread. We use a simple round-robin scheme to choose it:

```
class Async_F { //Sketch of the real code
  F instance; BoundedQueue[] q; int i; int T;
  Async_F(int N, int T) { //Default N=1 and T=1
    q = new BoundedQueue[T];
    this.T = T; i = 0;
    for (int j=0; j<T; j++) {
      q[j] = new BoundedQueue(N);
      new Thread(){ public void run() {
        while(true) { //Pseudocode with tuple
          (n,f,x) = q[j].pop();
          f.set(n.step(x));
      }}}.start(); //Spawn worker threads
    }
  }
  Future<Integer> step(int x) {
    Future<Integer> f = new Future<Integer>();
    q[i].push(instance, f, x);
    return f;
```

```
  }
  void reset() {
    instance = new F().reset();
    i = (i + 1) % T;
  }
}
```

3.7 Reordering Futures for Performances

Futures are strictly more expressive than FIFOs, in that they allow to retrieve the result of some computation at any time and in any order. In the case of a stream of futures coming from a stateful asynchronous computation without reset, trying to get the result of a future a_2, created after a_1, is useless since the result of a_1 needs to be computed for the computation of a_2 to begin. It may however be interesting, if the stream of futures is a combination of different sources, or if it comes from a function with `reset`. This is what we illustrate with our last example.

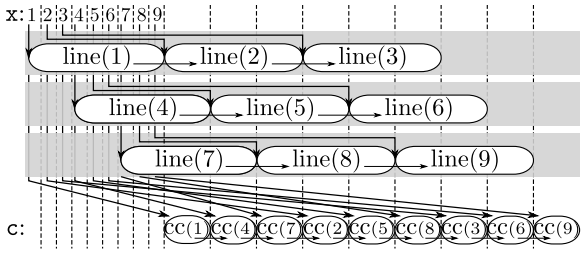
Consider a scalar stream representing a square matrix given line by line, element by element. Consider it is computed linewise with the node `line` reset every beginning of line, like with `array_dp`. In order to feed this matrix columnwise to a consumer node `cc`, the matrix needs to be transposed. Transposing the matrix of futures, then getting the values and feeding `cc` with them is much better than getting the values of the matrix, then transposing them and feeding them to `cc`. Indeed, thanks to the reset-induced data-parallelism, the first element of each line will be ready before the second element of each line, etc. This allows the column-width computation to beginning while the second element of the matrix is not even computed.

```
node transpose_adp<<n :int>>(x :int) = (c :bool)
var ay, ayt :future int; r :bool;
let
  r = period<<n>>();
  reset
    ay = async<<n-1, n>> line(x);
  every r;
  ayt = atranspose<<n>>(ay);
  c = cc(!ayt);
tel
```

We write `atranspose` for the matrix of futures transposition, instead of writing the code which is cluttering. It returns the transposition with a delay of the size of one matrix, `t = atranspose<<3>>(a)` returns:

a	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	s_{18}
t	0	0	0	0	0	0	0	0	0	a_1	a_4	a_7	a_2	a_5	a_8	a_3	a_6	a_9

The diagram below correspond to calling this node with `lx = 1 + (0 fby lx); y = transpose_adp<<3>>(lx);` The parenthesized values are the ones which would be computed if `line` and `cc` were identity functions: the result of `transpose_adp<<n>>(x)` would be equal to the transposition of `x` considered as a `n^n` scalar stream of flattened matrices. Transposing on the futures allows the computation of `cc` on 4 to be done before `line` finishes computing 2 and 3, etc.



4. SEMANTICS

Our language Heptagon is compiled source to source into a data-flow core language [5], allowing the formal semantics to be provided on a smaller language. The semantics builds on the construction and presentation of Delaval et al. [7].

A program P declares some definitions d and a main set of equations D. Definitions are either stateless (`fun`) or stateful (`node`) function declarations. A function inputs a variable x and defines an expression e with the help of some local equations. An expression e may be the usual immediate value i, variable x, pair (e, e), first (resp. second) element of a variable holding a pair `fst(x)` (resp. `snd(x)`, initialized synchronous register i `fby` x, sampling x `when` x, combination `merge` x x x, and the conditionally reset application of a function $f(x)$ `every` x.

We added to this classical core the conditionally reset asynchronous function application `async` $f(x)$ `every` x, the get operator `!` x, and the immediate future `async` i.

$$
\begin{aligned}
P &::= d; D \qquad\qquad d ::= \; d; d \mid f(x) \texttt{ = } e \texttt{ with } D \\
D &::= D \texttt{ and } D \mid x \texttt{ = } e \\
i &::= \texttt{async } i \mid \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \ldots \\
e &::= i \mid x \mid (x, x) \mid \texttt{fst(x)} \mid \texttt{snd(x)} \mid i \texttt{ fby } x \\
&\quad \mid x \texttt{ when } x \mid \texttt{merge } x\ x\ x \mid f(x) \texttt{ every } x \\
&\quad \mid \texttt{async } f(x) \texttt{ every } x \mid \texttt{! } x
\end{aligned}
$$

4.1 Standard Synchronous Semantics

Before dealing with futures and the asynchronous part of the semantics, let us present the baseline synchronous semantics of the core language.

A synchronous program *reacts* to some input performing so-called synchronous reactions. Formally, a synchronous program reacts by rewriting itself into another program, while emitting a reaction environment R_o containing all the values of the streams it defines. The semantics does not express the schedule of equations inside a reaction: all variables are seen as being defined simultaneously. To this matter, the predicate for some equations D at the same time emits R_o and reacts in the environment $R = R_i, R_o$, which already contains R_o and is augmented with the inputs in R_i. A is the asynchronous environment. The reaction environment R is a function from variables to extended values w which are either values v or absence of value abs. The second predicate defines that an expression rewrites itself while emitting an extended value.

$$
\begin{aligned}
R &::= x \mapsto w & w &::= v \mid abs \mid (w, w) \\
v &::= i \mid (v, v) & abs &::= \bot \mid (abs, abs) \\
A; R &\vdash D \xrightarrow{R_o} D' & A; R &\vdash e \xrightarrow{w} e'
\end{aligned}
$$

The rules are in Figure 1, they follow the classical [15] with a usual presentation [7]. By looking together at the Tautology and Def rules, it is clear that the environment emitted matches R. The And rule illustrate that equations are recursive, each one emitting part of the result but reading R in full. The synchronous behavior is given by the fact that is an abs value is emitted, the expression doesn't change and requires everything to be absent. The rule Instantiate uses the "do until then " construct which is not in the syntax. The idea is that a node f, when called, is the first time instantiated by inlining its code *until* the node is to be reset, in which case the program is again set to be a call to

$$\textsc{Immediate}\ \frac{w = i \mid abs}{A; R \vdash i \xrightarrow{w} i} \qquad \textsc{Tautology}\ \frac{R(x) = w}{A; R \vdash x \xrightarrow{w} x} \qquad \textsc{Pair}\ \frac{R(x_1) = w_1 \qquad R(x_2) = w_2}{A; R \vdash (x_1, x_2) \xrightarrow{(w_1,w_2)} (x_1, x_2)}$$

$$\textsc{First}\ \frac{R(x) = (w_1, w_2)}{A; R \vdash \texttt{fst(x)} \xrightarrow{w_1} \texttt{fst(x)}} \qquad \textsc{Second}\ \frac{R(x) = (w_1, w_2)}{A; R \vdash \texttt{snd(x)} \xrightarrow{w_2} \texttt{snd(x)}}$$

$$\textsc{With}\ \frac{A; R, R_1 \vdash D \xrightarrow{R_1} D' \qquad A; R, R_1 \vdash e \xrightarrow{w} e'}{A; R \vdash e\ \texttt{with}\ D \xrightarrow{w} e'\ \texttt{with}\ D'} \qquad \textsc{Def}\ \frac{A; R \vdash e \xrightarrow{w} e'}{A; R \vdash x\ \texttt{=}\ e \xrightarrow{x \mapsto w} x\ \texttt{=}\ e'}$$

$$\textsc{And}\ \frac{A; R \vdash D_1 \xrightarrow{R_1} D_1' \qquad A; R \vdash D_2 \xrightarrow{R_2} D_2'}{A; R \vdash D_1\ \texttt{and}\ D_2 \xrightarrow{R_1, R_2} D_1'\ \texttt{and}\ D_2'}$$

$$\textsc{Fby-abs}\ \frac{R(x) = abs}{A; R \vdash v\ \texttt{fby}\ x \xrightarrow{abs} v\ \texttt{fby}\ x} \qquad \textsc{Fby}\ \frac{R(x) = v'}{A; R \vdash v\ \texttt{fby}\ x \xrightarrow{v} v'\ \texttt{fby}\ x} \qquad \textsc{When-abs}\ \frac{R(x_1) = R(x_2) = abs}{A; R \vdash x_1\ \texttt{when}\ x_2 \xrightarrow{abs} x_1\ \texttt{when}\ x_2}$$

$$\textsc{When-t}\ \frac{R(x_1) = v_1 \qquad R(x_2) = true}{A; R \vdash x_1\ \texttt{when}\ x_2 \xrightarrow{v_1} x_1\ \texttt{when}\ x_2} \qquad \textsc{When-f}\ \frac{R(x_1) = v_1 \qquad R(x_2) = false}{A; R \vdash x_1\ \texttt{when}\ x_2 \xrightarrow{abs} x_1\ \texttt{when}\ x_2}$$

$$\textsc{Merge-abs}\ \frac{R(x_1) = R(x_2) = R(x_3) = abs}{A; R \vdash \texttt{merge}\ x_1\ x_2\ x_3 \xrightarrow{abs} \texttt{merge}\ x_1\ x_2\ x_3}$$

$$\textsc{Merge-t}\ \frac{R(x_1) = true \qquad R(x_2) = v_2 \qquad R(x_3) = abs}{A; R \vdash \texttt{merge}\ x_1\ x_2\ x_3 \xrightarrow{v_2} \texttt{merge}\ x_1\ x_2\ x_3} \qquad \textsc{Merge-f}\ \frac{R(x_1) = false \qquad R(x_2) = abs \qquad R(x_3) = v_3}{A; R \vdash \texttt{merge}\ x_1\ x_2\ x_3 \xrightarrow{v_3} \texttt{merge}\ x_1\ x_2\ x_3}$$

$$\textsc{Instantiate-abs}\ \frac{R(x_1) = R(x_2) = abs}{A; R \vdash f(x_1)\ \texttt{every}\ x_2 \xrightarrow{abs} f(x_1)\ \texttt{every}\ x_2} \qquad \textsc{Instantiate}\ \frac{\text{code}(f) = f(x)\ \texttt{=}\ e\ \texttt{with}\ D \qquad A; R \vdash e\ \texttt{with}\ (x\ \texttt{=}\ x_1\ \texttt{and}\ D) \xrightarrow{w} e'}{A; R \vdash f(x_1)\ \texttt{every}\ x_2 \xrightarrow{w} \text{do } e' \text{ until } x_2 \text{ then } f(x_1)\ \texttt{every}\ x_2}$$

$$\textsc{DoUntil-f/abs}\ \frac{R(x) = false \mid abs \qquad A; R \vdash D_1 \xrightarrow{R_1} D_1'}{A; R \vdash \text{do } D_1 \text{ until } x \text{ then } D_2 \xrightarrow{R_1} \text{do } D_1' \text{ until } x \text{ then } D_2} \qquad \textsc{DoUntil-t}\ \frac{R(x) = true \qquad A; R \vdash D_2 \xrightarrow{R_2} D_2'}{A; R \vdash \text{do } D_1 \text{ until } x \text{ then } D_2 \xrightarrow{R_2} D_2'}$$

Figure 1: Synchronous semantics

f. code(f) is a simple lookup in the immutable definitions d of the program. The tricky part is that while inlining, the first reaction needs to take place, and this is why e' is set instead of the inlined code (e `with` $x = x_1$ `and` D). The same remark applies to the DoUntil rules, which model strong preemption; it replaces itself by D_2' which is the result of the reaction of D_2.

4.2 Asynchronous Tasks

We give to each asynchronous, concurrent task an unique identifier a. It basically represent the result of the `new` done in the `reset` method of `async` wrappers. As we have seen in the examples, a concurrent task buffers an input and returns a future holding the corresponding output within the same tick. The asynchronous part is the reaction of a which is done in parallel. To allow for this, the asynchronous environment A stores for each a the streams of its inputs (in), outputs (out) and task state ($state$). To keep track of what has been computed by a, it also stores a counter (cnt), which indexes the current position of a in its streams:

$$A ::= a \mapsto \{in; out; state; cnt\} \qquad out ::= n \mapsto w$$
$$in ::= n \mapsto w \quad cnt ::= n \quad state ::= n \mapsto f(x)\ \texttt{=}\ e\ \texttt{with}\ D$$

The reaction of a task a is similar to the reaction of a synchronous program, the difference being that inputs are read in the input stream in, and the result is stored in the output stream out. This reaction is written $A \overset{a}{\rightsquigarrow} A'$ and is a correct evolution of A, if the cnt counter of a is incremented, and if the input, output and state streams agree to define the cnt reaction of a:

$$\frac{\begin{array}{c} A(a) = \{in; out; state; n\} \quad state(n) = f(x)\ \texttt{=}\ e\ \texttt{with}\ D \\ R = R_i, R_o \qquad R_i = x \mapsto in(n) \\ A; R \vdash D \xrightarrow{R_o} D' \quad A; R \vdash e \xrightarrow{w} e' \quad out(n) = w \\ A'(a) = \{in; out; state; n+1\} \\ state(n+1) = f(x) = e'\ \texttt{with}\ D' \end{array}}{A \overset{a}{\rightsquigarrow} A'}$$

As we saw in the examples, the reaction of a may need the reaction of other tasks to happen. It may even require multiple reactions of a given task. Let $\overset{a}{\rightsquigarrow}^*$denote the reflexive transitive closure of $\overset{a}{\rightsquigarrow}$. We are finally able to define the transition $\rightsquigarrow^*$which provides an abstraction of the interleaving of the asynchronous tasks, while ensuring proper evolution of A:

$$A \rightsquigarrow^* A' \overset{\text{def}}{=} \forall a,\ A \overset{a}{\rightsquigarrow}^* A'$$

4.3 Execution of a Program

A program reacts to a sequence of inputs $S_i = R_i^0 R_i^1 \ldots$, generating a sequence of outputs $S_o = R_o^0 R_o^1 \ldots$, in a sequence of asynchronous environments $B = A^0 A^1 \ldots$.

$$S_i \vdash_\infty P : B; S_o$$

The execution of $P = d; D^0$ is defined, synchronous step by synchronous step, with for all $k \geq 0$:

$$\frac{R_i^k \vdash_k d; D^k : A^k; R_o^k \quad A^k; R^k \vdash D^k \xrightarrow{R_o^k} D^{k+1} \quad A^k \rightsquigarrow^* A^{k+1}}{R_i^{k+1} \vdash_k d; D^{k+1} : A^{k+1}; R_o^{k+1}}$$

We saw that in order to abstract the schedule of equations, the semantic rules construct and check the validity of R in one go. R is temporary and can be thrown away after the

reaction. On the contrary, A is a continuously evolving environment, and a snapshot A^k is taken at each reaction. Similarly to R, a snapshot is constructed *and* checked during the reaction: inputs are queued up, asynchronous tasks advance and tasks with fresh code are attached, while all of this is already in the premise of the rules. This is fundamental to hide the interleaving between the main program and the asynchronous task. Note that, by the definition of $\rightsquigarrow^*$:

$$A^k.cnt \leq A^{k+1}.cnt \qquad A^k.state \subseteq A^{k+1}.state$$
$$A^k.in \subseteq A^{k+1}.in \qquad A^k.out \subseteq A^{k+1}.out$$

This permits to define the asynchronous environment A^∞ as the limit of the sequence. It is very important to note that A^∞ is thus scheduling-independent.

4.4 Extension of the Synchronous Semantics

A future is a *synchronous value* used through an indirection. It is basically is a reference holding a value guarded by a readiness condition. In the semantics, instead of using a generic reference representation, since any future is the result of a computation performed by an asynchronous task, we define a future as a special couple $\langle a, n\rangle$, with n the index of the value in the output stream of a. We add futures in the values of the semantics:

$$v ::= i \mid (v, v) \mid \langle a, n\rangle$$

To get the value of a future $\langle a, n\rangle$, we ensure that it is ready by asking a to have a current counter not less than n:

GET
$$\frac{R(x) = \langle a, n\rangle \quad A(a) = \{\_; out; \_; n'\} \quad out(n) = w \quad n' \geq n}{A; R \vdash \texttt{!}x \xrightarrow{w} \texttt{!}x}$$

The first time a node is called in an `async`, an asynchronous task a is dedicated to this computation. One has to make sure that a holds the right initial code in $state(0)$ and that the input queue at instant 0 is set with the correct input. The program is then rewritten into the simpler inputs(, ,) construct, which increments the input counter by one, ensuring that the input queue is filled in order. A task a operates a synchronous program and for each input, one output is set. Thus, the future emitted holds the input counter which also indicates the corresponding index in the output stream:

INSTANTIATE-ASYNC
$$\frac{\begin{array}{c}\text{code}(f) = f(x) \texttt{ = } e \texttt{ with } D \qquad A(a) = \{in; \_; state; \_\} \\ in(0) = R(x_1) \qquad state(0) = e \texttt{ with } (x \texttt{ = } x_1 \texttt{ and } D)\end{array}}{\begin{array}{c}A; R \vdash \texttt{async } f(x_1) \texttt{ every } x_2 \xrightarrow{\langle a,0\rangle} \\ \text{do inputs}(a, 1, x_1) \text{ until } x_2 \text{ then } \texttt{async } f(x_1) \texttt{ every } x_2\end{array}}$$

INPUTS
$$\frac{A(a) = \{in; \_; \_; \_\} \qquad in(n) = R(x)}{A; R \vdash \text{inputs}(a, n, x) \xrightarrow{\langle a,n\rangle} \text{inputs}(a, n + 1, x)}$$

Instantiation is not done until the input is present:

INSTANTIATE-ASYNC-ABS
$$\frac{R(x_1) = R(x_2) = abs}{A; R \vdash \texttt{async } f(x_1) \texttt{ every } x_2 \xrightarrow{abs} \texttt{async } f(x_1) \texttt{ every } x_2}$$

An immediate future is not the result of a computation, but for homogeneity, we simulate it with a stalled task:

IMMEDIATE-ASYNC
$$\frac{A(a) = \{\_; out; \_; 0\} \qquad out(0) = i}{A; R \vdash \texttt{async } i \xrightarrow{\langle a,0\rangle} \texttt{async } i}$$

4.5 Semantics Preservation

We need to compare our code with a fully synchronous code. To this mean, we define the ra function, which removes the asynchronous features. We will call $\text{ra}(P)$ the synchronized version of P, it is the identity except for:

$$\text{ra}(\texttt{async } f(x_1) \texttt{ every } x_2) = f(x_1) \texttt{ every } x_2$$
$$\text{ra}(\texttt{!}x) = x \qquad \text{ra}(\texttt{async } i) = i$$

The preservation theorem states that any value computed by the synchronized program is either the same in the original program, or replaced by a future holding that value. We express this property with the sync_A function, mapping extended values to extended values without futures:

$$\text{sync}_A(\langle a, n\rangle) = A\langle a, n\rangle$$
$$\text{sync}_A(w, w) = (\text{sync}_A(w), \text{sync}_A(w)) \qquad \text{sync}_A(abs) = abs$$
$$\text{sync}_A(v, v) = (\text{sync}_A(v), \text{sync}_A(v)) \qquad \text{sync}_A(i) = i$$

$A\langle a, n\rangle$ is a notation for the value associated to $\langle a, n\rangle$ in A, that is $A(a).out(n)$.

THEOREM 1 (PRESERVATION). *Under a stream of inputs without future, a program produces the same, or a future holding the same extended value as its synchronized version:*

$$\begin{array}{ll} \textit{if} & \forall R_i \in S_i, \forall x, R_i(x) \neq \langle a, n\rangle \\ \textit{then} & S_i \vdash P : A^\infty; S_o \iff S_i \vdash \text{ra}(P) : \emptyset; S'_o \\ \textit{with property } \mathsf{p}\textit{:} & \forall k, \forall x, R'^k_o(x) = \text{sync}_{A^\infty}(R^k_o(x)) \end{array}$$

First, note that we use A^∞ as an asynchronous environment, thanks to the remark in section 4.3. Second, restricting the inputs as not being futures is a sound simplification as the main program inputs actual values.

The proof relies on the stronger property P:

$$A^k; R^k_i \vdash D^k \xrightarrow{R^k_o} D^{k+1} \quad \wedge \quad \emptyset; R^k_i \vdash \text{ra}_{A^k}(D^k) \xrightarrow{R'^k_o} D'$$
$$\implies \begin{cases} (1) & \forall x, R'^k_o(x) = \text{sync}_{A^k}(R^k_o(x)) \\ (2) & \wedge\ D' = \text{ra}_{A^{k+1}}(D^{k+1}) \end{cases}$$

with

$$\text{ra}_A(w) = \text{sync}_A(w)$$
$$\text{ra}_A(\text{inputs}(a, n, x)) = \text{ra}_A((A(a).state(n))(x))$$

It states that synchronizing a program at any execution step k and performing a reaction leads to the same as performing a reaction and synchronizing the resulting program: synchronization commutes with reaction. Synchronizing the program at any step during its execution requires to extend ra on elements outside the syntax and makes ra dependent on the asynchronous environment. ra synchronizes futures by looking up their value, and instead of putting a value at the nth place in the input queue of a, it directly applies the code $state(n)$ of a to this input. Synchronizing a program is not possible if a future is not ready in A, or if a task a is not up to date with its input. So the property P does not hold with every schedule of the asynchronous tasks, but holds with an *eager scheduling*. The proof is done by induction over the constructs and is a bit verbose, but we look here at two symptomatic examples:

The simplest case is `x = async i; y = !x;` in which `x` holds a future $\langle a, 0\rangle$, and the asynchronous environment A is so that $A\langle a, 0\rangle = i$. `y` is equal to $A\langle a, 0\rangle$ which is i.

Its synchronized version is `x = i; y = x;`. Both codes are rewritten without change, and so holds property P.

The case of `x = async f(z); y = !x;` rewrites itself into `x = do` inputs$(a, 1,$`z`$)$ `until false then ...; y = !x;` in which `x` holds a future $\langle a, 0\rangle$, with $A^0(a).state(0)$ equal to the code of `f`, `y` asks for the result of this code applied on the input `z`. The synchronized version `x = f(z); y = x;` does immediately apply the code of `f` to `z` and writes the result in `x` and `y` (we have property P.(1)). It rewrites itself into `x = do` e `with` D `until false then ...; y = x;`. The next step, $A^1(a).state(1)$ is computed and is equal to e `with` D (we have property P.(2)). Next steps are similar.

An eager scheduling ensures that $A^1(a).state(1)$ is computed in time. Since S_o and A^∞ are independent of the interleaving, we can choose an eager scheduling, which gives the property P and the theorem follows.

4.6 Typing, Clocking and Causality

The preservation theorem 1 proves that the usual validity checks of data-flow synchronous programs—clocking and causality—can cope with the asynchronous constructs without modification. One simply ought to apply ra, which discard our extensions before running the checks.

Typing needs to deal with the paramteric type `future` t and with the introduced operators of section 2.2.

5. DISCUSSION

So far, real-time constraints and scheduling of tasks have not been dealt with. One would have to look for example at SynDEx [14] and the AAA methodology, to import these techniques into our context. Nonetheless, being able to program in a clean synchronous semantics the beginning and the end of concurrent tasks should be beneficial to the designer. Task priority, periodic schedule, and anything which is statically decided by the designer should be provided as static parameters of the `async` constructs. In our implementation, we experimented with the priority as a third argument and a processor identifier as a fourth one.

Scheduling of asynchronous tasks has to ensure any static constraints from the source: input queue size, thread number, etc. Note that there exists always an eager, dynamic schedule mimicking the sequential compilation of the synchronous semantics. The schedule may also be fixed and offline for static Kahn networks [17]. In a real-time system, the schedule may also be time-triggered, and the get operator can be handled as any inter-task dependence [14].

5.1 Static Resource Usage

In the presented semantics, A^∞ is unbounded in two ways: at each application of the rule INSTANTIATE-ASYNC, a fresh task is used, and the input and output streams, for each task, are entirely stored. In Section 3, we discussed the fact that in our implementation, the input queue size (q) and the number of tasks (p) are finite and set by the programmer as static arguments to the `async` construct. We saw that these resource constraints have an influence on the possible schedules, but do not change the semantics, indeed, the asynchronous version even with these bounds is still looser than the synchronous version.

The bound on the input queue prevents an asynchronous task to lag more than q ticks behind the task providing its input. So all inputs older than the last q ones may be dropped.

The number of tasks specified by `async` allows to statically allocate them at the beginning of the program. Indeed, an `async` does have only one current task and a task has finished if it is not the current one and has emptied its input queue. So, when a task is needed to become the new current one (rule INSTANTIATE-ASYNC), either a fresh one exists, or it waits for one to finish emptying its input queue.

Without further restriction, futures may escape their scope and require concurrent garbage-collection. Indeed, contrary to the input queue, it is not because a task is defining the nth output value that we are sure some older output value is no more needed. An output value is needed as long as a future holding it is in the program. To avoid garbage collection, we propose that a node may not return or take a future as input. Among our examples, the only violation of this rule is the version of the pipelining pattern, which was better rewritten with explicit delays. With this restriction, futures do not escape a node. At the end of the node's step, every futures it has used are dead, except for those stored in the registers of the node. Dealing with the futures is then a matter of a simple, constant time scan of a the node's bounded memory at the end of each step.

5.2 Safety of Asynchronous Constructs

If shared memory is used in the actual implementation, it is not directly accessible to the programmer, and the correctness of the futures library is sufficient to prevent races.

Under the assumption of a correct scheduling, the preservation theorem 1 ensures that if the synchronized version of a program is causal, the original program is deadlock-free. Finally, livelocks are impossible thanks to the fact that decoupling is bounded by the input queue sizes. Indeed, consider that a task is never given a chance to compute, since the decoupling with its caller is bounded, the caller also is stalled. Inductively it would imply the main program to be stalled. The bound on the decoupling enforce then that all tasks are stalled, which would be an incorrect schedule.

6. RELATED WORK

The distribution and parallelization of synchronous programs has been an active topic. The majority of these works are on Globally Asynchronous Locally Synchronous (GALS) approaches, targeting ESTEREL [4] or SIGNAL [21], and are discussed in a survey by Girault [10]. Model-based design has been applied to the distribution of a LUSTRE program into a real-time task graph [2]. The language PRELUDE [20], inspired by LUSTRE, is a real-time Architecture Description Language (ADL) to handle multi-rate but periodic systems. The generation of multi-threaded code from ESTEREL has been studied by Yuan et al. [25] but does not decouple computations across synchronous instants. CRP in ESTEREL [3, 4] introduces an `exec` construct to launch an external function with the result returned to the caller as a signal. The caller needs to handle the result whenever it arrives. The main purpose of `exec` is to trigger external tasks like robot actuation. At the opposite, OCREP [9] for LUSTRE considers the automatic distribution of the intermediate imperative language OC used by the LUSTRE compiler. The programmer does not control the desynchronization, she controls the distribution by indicating where variables should be located. Then OCREP automatically generate a distributed code with FIFOs trying, by using bi-simulation techniques, to link directly production to consumption. It is able [11] in the classi-

cal `slow_fast` example to determine the actual consumption and to remove the register `m_ys`. In the general case, it is impossible to do so, as it would require to predict whether and when a value of a stream is used. Moreover, the decoupling achieved by OCREP is, by design, bounded by one tick [11], and depends on choices and optimizations performed when generating OC.

Since futures have been used in a wide range of settings, they are themselves very diverse. The principal distinction opposes explicit vs. implicit futures. Implicit futures are futures without explicit get operator: they require language support but allow for elegant, implicit pipelining. It often comes with laziness support or continuation passing style. Recent works include F# concurrency [24] and many results from the Haskell community [22], the central library being `Control.Concurrent`. Alice ML implements an elegant concurrent lambda calculus with implicit futures [19]. Explicit futures are also coming in C11, C++11, etc.

The cyclo-static data flow model of computation [18, 6, 23] computations over a variety of shared and distributed memory targets. Our approach is more general since languages like LUSTRE do not have any periodic restrictions. On the other hand, cyclo-static approaches leverage the periodicity restrictions to allocate resources and partition computations automatically [13]. We see as an advantage for embedded system design that the programmer remains in control of parallelization and resource allocation. In addition, these decisions may still be automated on periodic subsets of a LUSTRE program.

7. CONCLUSION

We presented a novel approach to the distribution of synchronous programs, using futures in a semantics-preserving desynchronization. We showed that resources for parallel execution can be bounded statically, and that desynchronization does not lead to deadlock, livelock or races. We illustrated the expressiveness of our proposal on numerous examples, comparing with automatic techniques. We presented natural ways to exploit task and data parallelism, hiding the computation and communication latencies through explicit delays set by the designer. We also highlighted the importance of the reset operator to extract data parallelism from stateful tasks. We formally defined the semantics of our language and we validated the examples using a prototype compiler and runtime library.

Acknowledgments.

This work was partly supported by the INRIA large scale initiative Synchronics and by the European FP7 projects PHARAON id. 288307 and TERAFLUX id. 249013. We are thankful to Stephen Edwards, Louis Mandel, Adrien Guatto and Guillaume Baudart for their valuable recommendations on the paper.

8. REFERENCES

[1] http://aadl.info.

[2] M. Alras, P. Caspi, A. Girault, and P. Raymond. Model-based design of embedded control systems by means of a synchronous intermediate model. In *Design and Test in Europe (DATE)*, May 2009.

[3] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Principles Of Programming Languages*, pages 85–98. ACM, 1993.

[4] G. Berry and E. Sentovich. An implementation of constructive synchronous programs in polis. *Formal Methods In System Design*, 17(2):135–161, Oct. 2000.

[5] D. Biernacki, J.-L. Colaco, G. Hamon, and M. Pouzet. Clock-directed modular code generation of synchronous data-flow languages. In *International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[6] G. Bilsen, M. Engels, L. R., and J. A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing (ICASSP'95)*, pages 3255–3258, Detroit, Michigan, May 1995.

[7] G. Delaval, A. Girault, and M. Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[8] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12)*, Beijing, 12-13 June 2012. Best paper award.

[9] A. Girault. *Sur la Répartition de Programmes Synchrones*. Phd thesis, INPG, Grenoble, France, January 1994.

[10] A. Girault. A survey of automatic distribution method for synchronous programs. In *International workshop on synchronous languages, applications and programs, SLAP*, volume 5, 2005.

[11] A. Girault, X. Nicollin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *Transactions on Embedded Computing Systems (TECS)*, 5(3):687–717, 2006.

[12] M. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, MIT, 2010.

[13] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct 2006.

[14] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *International Workshop on Hardware Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[15] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 207–218, 1991.

[16] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, 1985.

[17] R. L. Jeronimo Castrillon and G. Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Trans. on Industrial Informatics*, page 19, nov 2011.

[18] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[19] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.

[20] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3), 2011.

[21] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111, 2006.

[22] D. Sabel and M. Schmidt-Schauß. A contextual semantics for concurrent haskell with futures. In *Principles and Practices of Declarative Programming*, pages 101–112. ACM, 2011.

[23] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference*, pages 777–782. IEEE, 2007.

[24] D. Syme, T. Petricek, and D. Lomov. *The F# Asynchronous Programming Model*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer Verlag, 2011.

[25] S. Yuan, L. H. Yoong, and P. S. Roop. Compiling esterel for multi-core execution. In *Euromicro Conference on Digital System Design*, pages 727–735, 2011.