Testing Speculative Work in a Lazy/Eager Parallel Functional Language

Alberto de la Encina, Ismael Rodríguez, and Fernando Rubio
Facultad de Informática
Universidad Complutense de Madrid
1. Introduction
2. The Language Eden
3. The Debugger Hood
4. Scheme for observing unnecessary speculation
5. Example: LinSolv
6. Conclusions
Motivation

- Sequentially lazy (Haskell)

- Eden:
  - Eager parallel computation
    (outputs of processes are always demanded)

- Eagerness $\Rightarrow$ Risk of unneeded work
  $\Rightarrow$ Need of profiling tools

- Functional language $\Rightarrow$ Fast development but difficult profiling
  $\Rightarrow$ no state!!
The Language Haskell

- **functional** (without side effects)

- **polymorphic**
  
  ```haskell```
  
  ```haskell```

- **lazy** (only demanded things are evaluated)

  ```haskell```

- **higher order** (functions are first class citizens)

  ```haskell```

  ```haskell```
Eden = Haskell + syntactical extensions for creating process topologies
+ eager evaluation of some expressions
+ eager process instantiations

- Process abstractions:
  \[ p :: \text{Process} \ (t_1, \ldots, t_m) \ (t'_1, \ldots, t'_n) \]
  \[ p = \text{process} \ (i_1, \ldots, i_m) -> (e_1, \ldots, e_n) \]
  \text{where} \ equations

- Process instantiations:
  \[ (#) :: \text{Process} \ a \ b -> a -> b \]
  \[ e_1 \ # \ e_2 \]
What is a Skeleton?

A skeleton is a parallel problem solving scheme
The aim is to reuse a parallel structure for many problems
It consists of:

1. A functional specification
2. One or more implementations. For each one:
   • A parallel algorithm
   • A cost model predicting the parallel execution time
Main idea: Processes are first class citizens in a higher-order language

Processes can receive/be parameters

Functional specifications: Written in Haskell

Parallel algorithms: Written in Eden itself ⇒ extensible
Parallel map

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]
\[
\text{map}\ f\ \text{xs} = [f\ x\mid x\leftarrow\text{xs}]
\]

A simple parallel version creates one process per list element:

\[
\text{map\_par} :: (a \to b) \to [a] \to [b]
\]
\[
\text{map\_par}\ f\ \text{xs} = [\text{pf}\ #\ x\mid x\leftarrow\text{xs}]\ 'using'\ \text{spine}
\]
\[
\text{where}\ \text{pf} = \text{process}\ x\to f\ x
\]
Parallel map — farm

A better approach creates a fix number of processes:

\[
\text{map}_\text{farm} :: \text{Int} \to (\text{Int} \to [a] \to [[a]]) \to ([[b]] \to [b]) \to \\
(a \to b) \to [a] \to [b]
\]

\[
\text{map}_\text{farm} \text{ np unshuffle shuffle f tasks} \\
= \text{shuffle} (\text{map}_\text{par} (\text{map} f) (\text{unshuffle np tasks}))
\]

Different strategies provided that: \( \text{shuffle} (\text{unshuffle np xs}) \equiv xs \)
Introduction to Hood

- In imperative programs debugging is simple. Intermediate values and the final result can be shown.

- **Hood** allows the programmer to observe the intermediate structures.

  ```plaintext
  natural = reverse . map (‘mod‘ 10)
      . takeWhile (/= 0) . iterate (‘div‘ 10)
  
  natural 3408
  > 3:4:0:8:[]

  -- after iterate 3408:340:34:3:0:_
  -- after takeWhile 3408:340:34:3:[]
  -- after map 8:0:4:3:[]
  ```
Introduction to Hood

observe: String → a → a

- observe s a = a
- as a side effect, the value of a associated to s is saved in a file.

natural = reverse
  . observe "after map" . map (‘mod‘ 10)
  . observe "after takeWhile" . takeWhile ( /= 0)
  . observe "after iterate" . iterate (‘div‘ 10)

observe "sum" sum (4:2:5:[])
-- sum { \ (4:2:5:[]) -> 11 }

observe "length" length (4:2:5:[])
-- length { \ (_,_:_:_:[]):} -> 3 }
Observing communication of processes — an example

• Process for generating infinite primes \( \geq n \):

\[
\text{pprimes} = \text{process } n \rightarrow \text{outputs}
\]
\[
\quad \text{where outputs} = \text{generatePrimes } n
\]
\[
\text{generatePrimes } x = \text{if (isPrime } x\text{) then } x \text{ : restOfPrimes}
\]
\[
\quad \text{else restOfPrimes}
\]
\[
\quad \text{where restOfPrimes} = \text{generatePrimes } (x+1)
\]

• Process for computing the shortest list of consecutive primes from \text{initialNumber} such that its multiplication is \( \geq \text{threshold} \):

\[
\text{myComputation} \text{ initialNumber} \text{ threshold} = \text{take neededNumber} \text{ primes}
\]
\[
\quad \text{where primes} = \text{pprimes } # \text{ initialNumber}
\]
\[
\text{products} = \text{scanl} (\ast) 1 \text{ primes}
\]
\[
\text{neededNumber} = \text{length} \left( \text{takeWhile} \left( \lt \text{threshold} \right) \text{ products} \right)
Observing communication of processes — an example

- Process for generating infinite primes ≥ n:
  
  ```haskell
  pprimes = process n -> (observe "outsFromProcess" outputs)
  where outputs = generatePrimes n
  generatePrimes x = if (isPrime x) then x : restOfPrimes
                   else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
  ```

- Process for computing the shortest list of consecutive primes from initialNumber such that its multiplication is ≥ threshold:
  
  ```haskell
  myComputation initialNumber threshold = take neededNumber primes
  where primes = pprimes # initialNumber
     products = scanl (*) 1 primes
  neededNumber = length (takeWhile (< threshold) products)
  ```
Observing communication of processes — an example

- Process for generating infinite primes ≥ n:

```plaintext
pprimes = process n -> (observe "outsFromProcess" outputs)
  where outputs = generatePrimes n
generatePrimes x = if (isPrime x) then x : restOfPrimes
  else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
```

- Process for computing the shortest list of consecutive primes from initialNumber such that its multiplication is ≥ threshold:

```plaintext
myComputation initialNumber threshold = take neededNumber primes
  where primes = observe "insFromProcess" (pprimes # initialNumber)
  products = scanl (*) 1 primes
  neededNumber = length (takeWhile (< threshold) products)
```
The general case
The general case

- **processObs**: function to run a given function as a new process

  ```
  processObs f = process ins -> outs
  where outs = f ins'
  ins' = ins
  ```

- **###**: (dummy) function to instantiate a process

  ```
  p ### actualParameters =
  p # actualParameters
  ```
The general case

- **processObs**: construction to observe the instantiated process

  ```haskell
  processObs f = process ins -> (observe "outsFromProcess" outs)
  where outs = f ins'
  ins' = observe "insToProcess" ins
  
  #": (dummy) function to instantiate a process

  ```haskell
  p #" actualParameters =
  p # actualParameters```
The general case

- processObs: *construction* to observe the instantiated process

```
processObs f = process ins \rightarrow (observe "outsFromProcess" outs)
   where outs = f ins'
   ins' = observe "insToProcess" ins
```

- `##`: *construction* to observe the invoker process

```
p ## actualParameters =
   observe "insFromProcess"
   (p # (observe "outsToProcess" actualParameters))
```
LinSolv Scheme

Exact solution arbitrary precision integers

\[ Ax = b \] where \( A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^n, n \in \mathbb{N} \)

Multiple homomorphic images approach:

- map the input data into several homomorphic images
- compute the solution in each image
- combine the results of all images to a result in the original domain.
LinSolv: To Speculate or not To Speculate

How much speculation should be included in LinSolv?

- **mapfarm:** hundreds of useless messages   Too much!!!!!!!
- **on demand:** No speculation at all   Bottleneck!!!
- **maprw:** 15 useless messages   Controlled
LinSolv: Results on a Beowulf

LinSolv: Speedup

![Graph showing speedup vs processors](image-url)
Conclusions and Future Work

- Profiling tool reporting speculative work
  - Rewriting Hood to parallelize it
  - Rewriting basic Eden constructions
- Rewriting the skeletons library
- Application to a real example

- Graphical interface
- Application to more examples
- Formal semantics