Take-home Exam in Advanced Programming

Deadline: Friday, November 2, 23:55

Preamble

This is the exam set for the course Advanced Programming, B1-2012. This document consists of 15 pages; make sure you have them all. The set consists of three questions. Your solution will be graded as a whole.

In the event of errors or ambiguities in the exam text, you are expected to state your assumptions as to the intended meaning. You may ask for clarifications in the discussion forum, but do not expect an immediate reply. If there is no time to resolve a case, you should proceed according to your chosen (documented) interpretation.

What To Hand In

To pass this exam you must hand in both a report and your source code:

- *The report* should be around 5–10 pages, not counting appendices, presenting your solutions and reflections. The report should contain all your source code in appendices. The report must be a PDF.
- *The source code* should be in a zip-file containing one directory called `src` (which may contain further subdirectories).

The hand in is done via the course web page in Absalon.

Learning Objectives

To get a passing grade you must demonstrate that you are both able to program a solution using the techniques taught in the course *and* write up your reflections and assessments of your own work.

- For each question your report should give an overview of your solution, **including an assessment** of how good you think your solution is and on which grounds you base your assessment. Likewise, it is important to document all *relevant* design decisions you have made.
- In your programming solutions emphasis should be on correctness, on demonstrating that your have understood the principles taught in the course, and on clear separation of concerns.

- It is important that you followed the required API, as your programs might be subjected to automatised testing as part of the grading. Failure to implement the correct API may influence your grade.

Exam Fraud

You are allowed to discuss how a question should be understood with teachers and other students (preferably in the discussion forum), but you are *not* allowed to discuss concrete solutions nor to copy parts of other students' programs. Submitting answers you have not written yourself or *sharing your code with others* is considered exam fraud. Likewise, make sure that you use proper academic citation for the material you use (including what you may find on the internet, e.g., pieces of code).

Emergency Webpage

There is an emergency web page at

`http://www.diku.dk/~kflarsen/ap-e2012/`

in case Absalon becomes unstable. The page will describe what to do if Absalon becomes unreachable during the exam, especially what to at the hand-in deadline.

Question 1: IVars in Erlang

The task in this question is to implement functionality similar to the Haskell Par-monad, in particullar `IVars`, but in Erlang. However, no knowledge about the Haskell Par-monad should be needed. `IVars` are write-once (immutable) variables that can be shared between processes. They can be read, with `get`, an operation that will block until a value was written to the variable. They should only be updated, with `put`, once.

There are two flavours of `IVars` in this question: *vanilla* and *princess*. An `IVar` should be represented by one or more Erlang processes. An `IVar` can be *compromised* if it is updated more than once (see details in the following).

Implement an Erlang module, `pm`, with the following client API. In the following, V denotes an `IVar` (vanilla or princess).

There are two functions for creating `IVars`:

(a) A function `newVanilla()` for creating an empty `IVar` with vanilla flavour.

(b) A function `newPrincess(`P`)` that takes a predicate, P, as argument for creating an empty `IVar` with princess flavour.

Only a value that satisfy P can be put into the `IVar`; all other values will be (silently) rejected and dropped[1]. We call P a *princess predicate*.

All `IVars` support the same basic API for modifying them:

(c) A function `get(`V`)` for getting the content of V, this function must **block** until a value has been `put` into V. If a value, T, has been `put` into V then `get` should return T. Multiple `gets` are allowed on the same `IVar`. However, all `gets` on the same `IVar` must return the same value.

(d) A function `put(`V`,` T`)` for putting the value T (any Erlang term) into V.

The `put` function must **not block** and should return immediately (in particular, it should not wait for the execution of any princess predicates). However, the effects of `put` on V depends on the flavour of V:

- for a vanilla `IVar`: only the first call to `put` should update the content of V. All subsequent calls to `put` will mark as V *compromised* but will not have any other effects.
- for a princess `IVar`: only the first value, T, that satisfies the princess predicate should update the content of V. All subsequent calls to `put` (whether their values satisfy the princess predicate or not) should have no effect. Princess `IVars` are never compromised.

(e) A function `compromised(`V`)` that returns `true` if V is compromised; and `false` otherwise.

[1] No good, away! *Quote from the princess in H.C. Andersen's "Silly Hans (Jack the Dullard)"*

(f) Your library must be robust in the following sense: Princess `IVars` should be robust against erroneous princess predicates; princess predicates can return non-boolean results or can throw exceptions. All erroneous behaviour is considered equivalent to `false`, and the `IVar` should continue to work. You are allowed to (and should) assume that all princess predicates will terminate.

Demonstrate how to use the `pm` module by writing a module `pmuse` with the functions `pmmap` and `treeforall`:

(g) Write a `pmmap(`*F*`, `*L*`)` that maps the function *F* over all the elements in the list *L*, and the elements in the resulting list are computed in parallel. The result of `pmmap(`*F*`, `*L*`)` should be equivalent to the result of `lists:map(`*F*`, `*L*`)` for all lists *L*, and all functions *F* (up to effects only observable due to difference in the order of evaluation, which also would result in `lists:map` being undefined).

(h) Write a `treeforall(`*T*`, `*P*`)` that checks that all elements in the binary tree, *T*, satisfy the predicate *P*. Here a binary tree is either the atom `leaf` or a tuple `{node, `*E*`, `*L*`, `*R*`}` where *E* is an element, and *L* and *R* are binary trees. To build some example trees you may use the `tree_of` function from Appendix A.

The function should return a result as soon as it is known (even while there might be ongoing computations). You may assume that *P* will always terminate.

Your solutions to `pmmap` and `treeforall` should not comprise any `IVars` and you should give an argument to why this is the case.

The handed-in version of your code should not spew unnecessary debug information to the console.

Question 2: Parsing Soil

Humus is a pure actor-based (concurrent) programming language with a pure functional core. Thus, compared to Erlang, there are no side-effects and, most importantly, there are no blocking operations (`receive` in Erlang blocks the process until a matching message is received). The language here, Soil, is a much cut-down version of Humus, but some of the core ideas of Humus are retained.

In this question you should use one of the two monadic parser libraries, `SimpleParse.hs` or `ReadP`, from the course to write a parser for Soil. The interpreter must be implemented in a `SoilParser` module. You find Haskell skeletons for the parser and abstract syntax tree at Absalon.

Remember to write tests to show that your parser works. You can find some inspiration in Appendix B.

Grammar

Your parser must accept programs written in the following grammar:

Program ::= *DefOps*
DefOps ::= *FunDef DefOps*
| *ActOps*
FunDef ::= `'let'` *ident* `'('` *Params* `')'` `'from'` *name* `'='` *Expr* `'end'`
Params ::= *SomeParams name*
| ε
SomeParams ::= *SomeParams name* `','`
| ε
Expr ::= `'case'` *Prim* `'of'` *Cases* `'end'`
| `'if'` *Prim* `'=='` *Prim* `'then'` *Expr* `'else'` *Expr* `'end'`
| *ActOps*
Cases ::= `'('` *Params* `')'` `':'` *Expr Cases*
| `'_'` `':'` *Expr*
ActOps ::= *ActOps ActOp*
| ε
ActOp ::= `'send'` `'('` *Args* `')'` `'to'` *Prim*
| `'create'` *Prim* `'with'` *Fcall*
| `'become'` *Fcall*
Fcall ::= *Prim* `'('` *Args* `')'`
Args ::= *SomeArgs*
| ε
SomeArgs ::= *Prim* `','` *SomeArgs*
| *Prim*
Prim ::= *name*
| *ident*
| `'self'`
| *Prim* `'concat'` *Prim*

In this:

ε is the empty sequence.

name is a non-empty sequence of letters, digits and underscores (_), starting with a letter. These cannot equal any of the reserved keywords: `let`, `from`, `case`, `send`, etc.

ident is a `#` followed by a non-empty sequence of letters, digits and underscores (_).

`concat` is left associative.

Tokens are separated by arbitrary whitespaces (spaces, tabs, and newlines) or stand-alone punctuation `()=,:`.

Abstract syntax tree

Your parser must construct abstract syntax trees represented with the following data types. You can find this in `SoilAst.hs`.

```
module SoilAst where

type Ident    = String  -- w/o leading #
type Name     = String

type Program  = ([Func],[ActOp])

data Func     = Func { funcname  :: Ident
                     , params    :: [Name]   -- Zero or more
                     , receive   :: Name
                     , body      :: Expr}
              deriving (Eq, Show)

data Expr     = CaseOf Prim [([Name], Expr)] Expr
              | IfEq Prim Prim Expr Expr
              | Acts [ActOp]
              deriving (Eq, Show)

data ActOp    = SendTo [Prim] Prim
              | Create Prim Prim [Prim]
              | Become Prim [Prim]
              deriving (Eq, Show)

data Prim     = Id Ident
              | Par Name
              | Self
              | Concat Prim Prim
```

```
            deriving (Eq, Show)
```

Thus, you should implement a function `parseString` for parsing a Soil program given as a string:

```
parseString :: String -> Either Error Program
```

Where you decide and specify what the type `Error` should be.

For example, `parseString "send (#ok) to self"` should give `Right ([], [SendTo [Id "ok"] Self]`.

Likewise, you should implement function `parseFile(`*filename*`)` for parsing a Soil program in file specified by *filename*:

```
parseFile :: FilePath -> IO Either Error Program
```

Where `Error` is the same type as for `parseString`.

You should not change the types for the abstract syntax trees unless there is an update at Absalon telling you explicitly that you can do so.

Question 3: Interpreting Soil

In this part you shall write an interpreter for Soil, which is defined over the abstract syntax tree from Question 2. As an actor-based language, Soil is non-deterministic, which mainly comes from the possibility of concurrent execution of processes. However, in this question you are *not* asked to implement concurrency between processes. You must, however, implement two simple process scheduling algorithms. More details in the following.

The interpreter must be implemented in a `SoilInterp` module. You can find a Haskell skeleton for this with the required API (defined in the following) at Absalon.

You can find program examples in Appendix B and at Absalon.

The language

A program in Soil can be described in the following way:

- A program consists of some function definitions followed by a list of actor operations. The program is executed by evaluating the list of actor operations in order.
- A function definition is a (function) identifier, zero or more parameters, a parameter that holds the message received, and a function body. Bindings are not changes after initialization.
- A process consists of a (process) identifier, a function identifier, a list of function arguments, and a mailbox. Only the process identifier is static.

  A process runs by calling this process function with the arguments and the first (oldest) message in the mailbox. Unlike Erlang, a message in Soil will always be matched and bound to a single parameter. Afterwards, it is possible to use the `case-of` to decompose the elements of a tuple.

  Processes *never* terminate, but run their "process function" whenever there is a message in the mailbox. It is possible to change the function and the arguments with the `become` operation.

Soil operations have the following (informal) semantics:

`CaseOf switch [(match, do)] default`: tests sequentially if the pattern of `switch` matches a pattern of some `match`. If a pattern match is found, then the names in the match are bound to the corresponding values, and the associated `do` is performed; otherwise `default` is performed. A pattern matches if the tuples have the same number of elements. The case-of is used to decompose a tuple in a message.

`IfEq prim1 prim2 expr1 expr2`: tests if the evaluated values of `prim1` and `prim2` are equal. If so, evaluates `expr1`, otherwise evaluates `expr2`.

`SendTo msgs receiver`: sends the evaluated message in `msgs` to the `receiver`; i.e. inserts the evaluated values of `msg` into the mailbox of the `receiver`.

`Create pid funid [vals]`: creates a new process with the process id `pid` that should evaluate the function `funid` with evaluated arguments `vals`. The process is inserted into the process queue.

`Become funid [vals]`: changes the evaluation function of the running process to `funid` with evaluated arguments `vals`.

`Self`: evaluates to the `Ident` process id of the running process.

`Concat prim1 prim2`: evaluates `prim1` and `prim2` to `Idents` and returns a new `Ident` that is the string concatenation of the two other `Idents`.

There is no I/O in Soil, but (following an idea from Humus) we instead assume the existence of *predefined processes* that perform certain "impure" operations. In Soil, there are two such processes: `#println` that writes a line with whatever message it receives to standard output, and `#errorlog` that writes a line with whatever message it receives to a log file. A message (list of `Idents`) should be written in `#println` as a colon-separated (":") string (e.g. `["foo", "bar"]` becomes `"foo:bar"`). Your interpreter however, must neither write to standard output, nor any log file. Instead, the "would-be" printed lines, and the "would-be" written log file lines, should serve as the output of program evaluation. That is, the result of running a Soil program should be a tuple of two string lists. This is specified more formally in subquestion 6.

Errors

For a concurrent system we want to have a robust runtime system, i.e. when the program has started, processes must not halt with an error. We will simulate this behaviour by dividing program errors into two classes (static and runtime errors) that are treated differently. A static error can be checked at "compile-time", while a runtime error must perform a suitable recovery operation.

The following are the only static errors:

- Parameters in functions are bound more than once.
- More than one function is defined with the same name.
- A function body tries to use an undefined parameter. All parameter names are defined at the head of the function, so this can be checked statically.

Writing the static code pass is *not* part of the exam. You may assume that there are no static errors and may halt the interpreter with a Haskell `error` if a static error is detected, whatever makes your code most elegant.

Operations that lead to a runtime error must be discarded, and the program must continue as if nothing had happened. A runtime error must be logged by sending an appropriate message to the `#errorlog` process. The following operations cause runtime errors:

- A `SendTo` operation sends a message to an undefined process. Message passing is asynchronous and the message is lost.

- A `Become` or `Create` operation is called with an undefined function name or a wrong number of arguments.
- A `Create` operation creates a process with an existing process identifier. It is not a problem that processes have names that are identical to functions.

Your solution

When you are making your solution follow the structure outlined here. If you have trouble completing the implementation, you should try to make a partial implementation. Every task gives points.

Your code should be written in a monadic style without using the `IO` monad.

1. Define a `NameEnv` datatype that maps `Names` to `Idents` and functions for insert and lookup in the environment. If a name already exists when inserting, or a parameter name does not exist when looking it up, perform the desired handling strategy of static errors, as mentioned earlier.
2. Similar to the previous, define a `FuncEnv` datatype that maps `Idents` to `Funcs`, and functions for insertion and lookup of functions in the environment. If an identifier already exists when inserting, perform the desired handling strategy of static errors, as mentioned earlier. An identifier that is not associated with a function when trying to look it up, is a runtime error; lookup of functions only occur in `Create` and `Become` operations.
3. Define a `Process` datatype, which contains the function that the process is executing, the values to its arguments, and the current mailbox of the process.

   ```
   type Message = [Ident]

   data Process = Process { function  :: Ident
                          , arguments :: [Ident]
                          , mailbox   :: [Message]}
   ```

   Based on a process, define a `ProcessQueue` datatype that holds all currently executing processes. On this implement functions that performs the following operations:

   - adding new message to a process,
   - fetching the oldest message from a process,
   - updating evaluation function and its arguments for a process
   - fetching evaluation function information from a process, and
   - adding a new process to the end of the queue.
4. Define a function `processStep` that performs one process step. That is, given a process id, lookup the function and arguments, fetch a message from the mailbox and execute the process function with the given values. This also includes functions that evaluate over the abstract syntax tree. If the mailbox is empty when fetching a message, the function should return without executing the process function.

5. Implement the simple process scheduling based on a round-robin algorithm. That is, implement a function `nextProcessRR` that treats the process queue as a FIFO queue and when called returns the `Ident` of the next process to be executed. The returned process is then moved to the back of the queue. `nextProcessRR` should return the next process no matter the amount of messages in the mailbox. Newly created processes must always be placed at the back of the queue.

   The single process execution combined with this scheduling will make our distributed language deterministic, but thus also easier for you to test.

6. Define a function `runProgRR` as

   ```
   runProgRR :: Int -> Program -> ([String], [String])
   ```

   that runs a Soil program for a given number (the `Int`) of process steps (calls to `nextProcessRR`) and produces a list of strings with the results printed by `#println` and a list of strings with the runtime error messages.

   You should test your solution with the programs listed in Appendix B running for 20 steps.

7. Finally, extend our interpreter to also simulate that processes can execute in any order. This will be done by performing all possible process execution orders. So write a process scheduler function, called `nextProcAll`, that returns a *list* of process that can process a message. Processes with no messages in their mailbox should, thus, not be included in the list.

   Similarly, implement a run-function

   ```
   runProgAll :: Int -> Program -> [([String], [String])]
   ```

   that returns a list of all possible results when running the list of processes from `nextProcAll` for a given number of process steps.

   Test this implementation with the same tests as earlier.

Appendix A: Binary Trees

```
tree_of(Xs) -> build([{leaf,X} || X <- Xs ]).

build([]) -> leaf;
build([{leaf,X}]) -> {node, X, leaf, leaf};
build([{{node,Y,T1,T2}, X}]) -> {node, Y, T1, build([{T2, X}])};
build(List) -> build(sweep(List)).

sweep([])   -> [];
sweep([Ts]) -> [Ts];
sweep([{T1,X1},{T2,X2}|Ts]) -> [{{node, X1, T1, T2},X2}|sweep(Ts)].
```

Appendix B: Soil Example Programs

In this appendix you find three examples Soil programs. Text-files with these Soil programs and their abstract syntax trees can be found at Absalon.

Hello World

In the following are a few examples of the language. The following are a simple hello-world program

```
let #print() from message =
  send (message) to #println
end

create #hw1 with #print()
create #hw2 with #print()
send (#Hello) to #hw1
send (#World) to #hw2
```

I'll clean up your mess

One process duplicates the message, the other half the messages. This should never terminate.

```
let #dub() from message  =
  case message of
    (sender, msg) : send (self, msg) to sender
                    send (self, msg) to sender
    _             : send (#FaultyMessage) to #println
  end
end

let #half(state) from message =
  if state == #skip then
    become #half(#return)
    send (#SkippingMessage) to #println
  else
    case message of
      (sender, msg) : become #half(#skip)
                      send (self, msg) to sender
      _             : send (#FaultyMessage) to #println
    end
  end
end

create #dubproc with #dub()
create #halfproc with #half(#return)
send (#halfproc, #foo) to #dubproc
```

Hello World behind gates

The #foobar process forwards everything to #println. When the #gatekeeper process has received two messages it concatenates the two receiver-ids (e.g. #foo and #bar becomes #foobar) and send the collected message to the concatenated process-id. This should in the deterministic (round-robin) version only print `Hello_World` but create other combinations with run-time send errors. In the non-deterministic version it can also print `Bye_World`.

```
let #printer () from message =
  send (message) to #println
end

let #gate (fst, fstmsg) from message =
  case message of
    (snd, sndmsg) :
      if fst == #none then
        become #gate(snd, sndmsg)
      else
        send (fstmsg, sndmsg) to fst concat snd
        become #gate(#none, #none)
      end
    _ :
  end
end

let #repeat (other) from message =
  send (message) to #gatekeeper
  send (message) to other
end

create #foobar with #printer()
create #gatekeeper with #gate(#none, #none)
create #repeater1 with #repeat(#repeater2)
create #repeater2 with #repeat(#repeater1)
send (#foo, #Hello) to #repeater1()
send (#bar, #World) to #repeater2()
send (#foo, #Bye) to #repeater1()
```