Take-home Exam in Advanced Programming

Deadline: Thursday, November 6, 16:00

Version 1.2

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2014. This document consists of 25 pages; make sure you have them all. Please read the entire preamble carefully.

The exam set consists of 3 questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner.

In the event of errors or ambiguities in the exam set, you are expected to state your assumptions as to the intended meaning in your report. 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 (at least) your solutions, reflections, and assumptions, if any. The report should contain all your source code in appendices. The report must be a PDF document.
- *The source code* should be in a .ZIP file, archiving one directory called `src` (which may contain further subdirectories).

Make sure that you follow the format specifications (PDF and .ZIP). If you don't, the hand in **will not be assessed** and treated as a blank hand in. The hand in is done via the course web page on 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 implement the required API, as your programs might be subjected to automated testing as part of the grading. Failure to implement the correct API may influence your grade.
- To get a passing grade, you *must* have some working code for all questions.

Exam Fraud

The exam is an individual exam, thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course.

You are only allowed to discuss how a question is to be interpreted with the teachers and their assitants on the discussion forum set up on the course-page on Absalon. If you are afraid that your question does not fall into this category, you can instead send an email to `kflarsen@diku.dk`.

Specifically, but not exclusively, you are **not** allowed to discuss any part of the exam with any other student nor to copy parts of other students' programs. Submitting answers you have not written yourself, or *sharing your answers with others*, is considered exam fraud.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. Make sure that you use proper academic citation for the material you draw considerable inspiration from (including what you may find on the Internet, e.g., pieces of code). Also note, that these rules mean that it is not allowed to copy any part of the exam set (or supplementary skeleton files) and publish it on forums other than the course discussion forum (e.g. IRC, exam banks, chatrooms, or suchlike).

During the exam period, students are not allowed to answer questions, *only teachers and their assitants are allowed to answer questions* on the discussion forum.

Breaches of the above policy will be handled in accordance with the Faculty of Science's disciplinary procedures.

Emergency Webpage

There is an emergency web page at

`http://www.diku.dk/~kflarsen/ap-e2014/`

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

Gotta go Fast

For the first two questions, you will be using Haskell to implement a language called Fast.

"Gotta Go Fast" – 2008 – artist unknown

Background

You are a language designer at a large fruity software company, with a large ecosystem of programmers writing software for your platform. For many years, the only supported language has been Subjective P, a pidgin language made by crudely bolting Debate, a high-level object-oriented language, onto P, an old low-level systems programming language.

For some reason, the programmers on your platform have proven unable to recognise the beauty of this design, and after years of complaints, you have decided to implement a new dynamically typed object-oriented enterprise language as a replacement.

The name of this language is Fast.

Fast is based on a traditional class model without inheritance, where every value is an object, and every object is an instance of a class. Synchronous (blocking, RPC-style) message passing is the only way objects can interact. An object maintains state in the form

of mutable fields, which can only be changed from within the object itself. A class has no fields, fields are added to an object at runtime (as in e.g. Python). The full details of the language will be given on the following pages, but the most important concepts are *classes* and *messages*.

A message can be any FAST value, and a class can define a *receive action*, which will be executed when a object of the given class receives a message which is not a method call. A "method call" is just a message consisting of a special kind of value, called a *term*, which consists of a symbol and a list of subvalues. For example, the value `foo(42, "bar")` is a term with the symbol `foo` and the subvalues `42` and `"bar"`. Sending the message `foo(42, "bar")` to the object `quux` is *exactly the same* as calling the method `foo` with arguments `(42, "bar")` on `quux`. The former is written `send(quux, foo(42, "bar"))`, and the latter `quux.foo(42, "bar")`. In this sense, the method call syntax is just a convenient shortcut. While methods could be implemented by pattern-matching on messages received in the receive action, FAST also supports a specialised syntax for defining methods that takes precedence - the receive action is only invoked if no method matches the message. This does not extend the power of the language, but merely provides syntactic sugar.

It is very important to recognise that the expression `foo(42, x)` is not a function call, despite its likeness to function calls in other languages – because FAST has no functions. It is an expression that constructs a term `foo(42, `v`)`, where v is value of the variable `x`.

Examples of FAST programs can be found in Appendix A.

Handouts

Alongside the exam text, on Absalon you will find four Haskell source files - `Fast.hs`, `FastAST.hs`, `FastParser.hs`, and `FastInterpreter.hs`. You will be modifying the latter two of these files. They already contain an incomplete skeleton to help guide you in solving the problem, but, as long as you maintain the external interface, you are free to disregard the skeleton. Further details are given in the concrete exam questions.

Question 1: Parsing Fast

In this question you must use one of the three monadic parser libraries, `SimpleParse.hs`, `ReadP` or Parsec, from the course to write a parser for Fast. The parser must be implemented in a `FastParser` module. You find Haskell skeletons for the parser and abstract syntax tree on Absalon.

If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators`), in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

The grammar for Fast is given on the following page. Furthermore,

- ϵ is the empty sequence.
- *integer* is an arbitrary-precision integer constant, with leading zero permitted.
- *string* is a string constant matching the regular expression `"[^"]*"`. This is similar to most languages, except that backslash-escapes are not available.
- *Name* is a nonempty sequence of letters, digits, and underscores (_), starting with a letter, that is not a reserved word.
- The reserved words are `self`, `class`, `new`, `receive`, `send`, `match`, `return`, and `set`.
- The method call/field access operator (`.`) has highest priority, followed by `*` and `/`, which have higher precedence than `+` and `-`, which have higher precedence than `=`, which has higher precedence than `return`. All four arithmetic operators are left-associative.

Alpha-numeric tokens (*Name*s and reserved words) are separated by at least one whitespace (spaces, tabs, and newlines), symbolic tokens are separated by arbitrary whitespace.

If you make any kind of transformation of the grammar as part of your parser construction, make sure to detail them in your report.

Advice for your solution

If you have difficulties making your interpreter work for the full language, then try to make it work for a simpler subset of the language. For example, you can disallow chaining of method calls. This means that the expression

```
obj.f(x).g(y).h(z);
```

must then be written

```
set a = obj.f(x);
set b = a.g(y);
set c = c.h(z);
```

If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Grammar

Program	::=	*ClassDecls*
ClassDecls	::=	ϵ
	|	*ClassDecl ClassDecls*
ClassDecl	::=	`'class'` *Name* `'{'` *ConstructorDecl NamedMethodDecls RecvDecl* `'}'`
ConstructorDecl	::=	ϵ
	|	`'new' '('` *Params* `')' '{'` *Exprs* `'}'`
NamedMethodDecls	::=	ϵ
	|	*NamedMethodDecl NamedMethodDecls*
NamedMethodDecl	::=	*Name* `'('` *Params* `')' '{'` *Exprs* `'}'`
RecvDecl	::=	ϵ
	|	`'receive' '('` *Param* `')' '{'` *Exprs* `'}'`
Params	::=	ϵ
	|	*Params′*
Params′	::=	*Param*
	|	*Param* `','` *Params′*
Param	::=	*Name*
Args	::=	ϵ
	|	*Args′*
Args′	::=	*Expr*
	|	*Expr* `','` *Args′*
Exprs	::=	ϵ
	|	*Expr* `';'` *Exprs*
Expr	::=	*integer*
	|	*string*
	|	*Name*
	|	*Name* `'('` *Args* `')'`
	|	`'self'`
	|	`'return'` *Expr*
	|	`'set' 'self' '.'` *Name* `'='` *Expr*
	|	`'set'` *Name* `'='` *Expr*
	|	*Expr* `'+'` *Expr*
	|	*Expr* `'-'` *Expr*
	|	*Expr* `'*'` *Expr*
	|	*Expr* `'/'` *Expr*
	|	`'match'` *Expr* `'{'` *Cases* `'}'`
	|	`'send' '('` *Expr* `','` *Expr* `')'`
	|	`'self' '.'` *Name*
	|	*Expr* `'.'` *Name* `'('` *Args* `')'`
	|	`'new'` *Name* `'('` *Args* `')'`
	|	`'('` *Expr* `')'`
Cases	::=	ϵ
	|	*Case Cases*
Case	::=	*Pattern* `'->' '{'` *Exprs* `'}'`
Pattern	::=	*integer*
	|	*string*
	|	*Name* `'('` *Params* `')'`
	|	*Name*

Abstract syntax trees

Your parser must construct abstract syntax trees represented with the data types defined in `FastAST.hs`.

The mapping from grammar to constructors should be straightforward, perhaps with the exception of terms. The expression `foo(a,b)` is not a function call (Fast has no functions), but is instead a `TermLiteral`. In a *Pattern* context, it is a `TermPattern`.

Thus, you should implement module `FastParser` with the following interface: a function `parseString` for parsing a Fast program given as a string:

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

Where you decide and specify what the type `Error` should be. The type `Error` must also be exported from the module. The handed-out skeleton code already has the exports set up correctly.

Likewise, you should implement a function `parseFile` for parsing a Fast program given in a file located at a given path:

```
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 on Absalon telling you explicitly that you can do so.

Together with your parser you must also hand in a test-suite to show that your parser works (or where it does not work).

Question 2: Interpreting Fast

A Fast program consists of a set of class declarations. To start the program, the class with the name `Main` is instantiated, with no arguments passed to its constructor (see below). The result of the program is a string, which is constructed by sending `printLn()` messages to values.

A class declaration consists of three parts:

(a) An optional *constructor*, which implements object creation. When instances of the class are created using `new`, the number of arguments passed to `new` must match the number of parameters of the constructor. The return value of the constructor is not used. If no constructor is given, this is equivalent to a constructor having zero parameters and an empty body.

(b) A possibly empty sequence of *method declarations*.

(c) An optional *receive action*.

An object consists of a list of mutable *fields* (sometimes called "properties" in other languages), that act like variables stored in the object itself. These fields can be modified by the object itself, but not from outside the object. Every object is an instance of a class. When an object is first created, its constructor is executed, with the object initially having no fields. A field is created whenever it is first assigned a value - there is no need (or way) to declare which fields exist in advance.

Constructor, method and receive action bodies, as well as `match` bodies, all consist of semicolon-separated expressions. The value of the last expression is returned, unless `return` is used. If the sequence of expressions is empty, the term `nil()` is returned. Since FAST is an impure language, the order of evaluation matters, and is always left-to-right.

The semantics of most expressions should be intuitively understandable, but the following merit elaboration.

`new` *c*`(`*args*`)`**:** Instantiate an object of class *c*. The *args* are passed to the constructor, and when the constructor has finished executing, a reference to the new object is returned.

`self`: Yields a reference to the object that is currently executing (like `this` in inferior languages).

`self.`*p*: Read field *p* of the current object.

`self.`*p* **`=`** *e*: Change field *p* of the current object. If *p* does not already exist, it is created.

v: Read variable *v*, which can be a function parameter, `match`-bound name, or ordinary variable, but must already exist.

`set` *v* **`=`** *e*: Set variable *v* to a new value. If *v* does not already exist, it is created.

`return` *e*: Terminate execution of the current method (or constructor, or receive action) immediately, returning the result of evaluating *e*.

obj`.`*f*`(`*args*`)`: The same as `send(`*obj*`,` *f*`(`*args*`))`.

`send(`*obj*`,` *e*`)`:** Evaluate *e* and send the resulting value as a message to the object *obj*. Returns the result yielded by the receiving object. Message passing is thus *synchronous*. Further details on message receival are found below.

`match` *e* **`{`** pat_1 **`->`** **`{`** es_1 **`}`** **`...`** pat_n **`->`** **`{`** es_n **`}`** **`}`:** Evaluate *e* and match the resulting value against each of the patterns $pat_1 ... pat_n$. For the first pattern pat_i that matches, evaluate the corresponding es_i, with new variables in scope corresponding to the names (if any) bound in pat_i. For more on pattern matching, see below. If no pattern matches, return `nil()`.

Message Receival

When an object is sent a message, the list of method declarations in its class is inspected to see whether a matching method exists. For a term message $f(v_1, ..., v_n)$, a matching method must have the name f and take n parameters. If a matching method is found, its body is executed with the parameters bound to the subvalues of the message term, and

the result of the body returned to the sender of the message. If and only if no matching method can be found, the receive action is executed, with its single parameter bound to the message value.

If the message is not a term value, there can be no matching method, and the receive action is always chosen.

If there is no matching method and the class has not defined a receive action, execution must stop with an error.

Pattern Matching

The possible patterns, and what they match, are defined as follows.

`ConstInt` *i*: Matches the integer value i, with no new bindings.

`ConstString` *s*: Matches the string value s, with no new bindings.

`AnyValue` *k*: Matches any value v, yielding the new binding $k \mapsto v$.

`TermPattern` *sym* `[`k_1`, ..., `k_n`]`: Matches the term value $sym(v_1, ..., v_n)$, where each v_i can be any value. Yields the bindings $k_1 \mapsto v_1, ..., k_n \mapsto v_n$.

Built-in Methods

All non-reference values support a `printLn()` message (equivalent to a `printLn` method taking no arguments) which, when received, causes the value to append its textual representation (followed by a newline character) to the *output string* which is returned as the result of a FAST program. The textual representation is the result of the function `printed` in the handed-out `FastInterpreter.hs`. Sending a `printLn()` message to an instance of a user-defined class has no special significance (i.e. fails with an error, unless the class itself handles the `printLn()` message).

Errors

The interpreter must terminate with an error in *at least* the following circumstances:

- There is no `Main` class.
- A constructor is called with an invalid number of arguments.
- There are duplicate class, parameter, or method names, or duplicate names in a `TermPattern`.
- The arithmetic operators are passed non-integers - this isn't PHP, after all.
- A message is sent for which there is no applicable method and no receive action.

The interpreter must not terminate with an error on programs that are not erroneous.

Your Task

The main objective of this question is that you should demonstrate that you know how to write an interpreter using monads for structuring your code. Thus you should structure your solution along the following lines:

(a) Define a module `FastInterpreter` that exports a function `runProg` and a type `Error`.

(b) Define a function `runProg`

```
runProg :: Prog -> Either Error String
```

`runProg` *p* *p* runs program *p*, yielding either a runtime error, or the output of the program (as per the `printLn` method described above). You will need to define the type `Error` as well.

(c) See the handed-out `FastInterpreter.hs` for a *strongly recommended* skeleton for your solution. This file consists of incomplete and commented-out definitions – the latter are not part of the core skeleton, but are provided as guidance and inspiration for your own helper functions.

Your solution should implemented as module `FastInterpreter` that exports at least `Error` and `runProg`. The handed-out skeleton code already has the exports set up correctly.

Once you have implemented the parser and interpreter, the file `Fast.hs` can be used to run FAST programs, as follows.

```
% runhaskell Fast.hs program.fast
```

You should *not* need to modify `Fast.hs`.

Advice for your solution

If you have difficulties making your interpreter work for the full language, then try to make it work for a simpler subset of the language, for example by one (or more) of the following simplifications:

- Disallowing `return`.
- Supporting only integer-constant patterns in `match`.
- Disallowing `send` and receive actions - that is, permit only straight method definitions and method calls.

If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Question 3: Spreadsheet

This question is about implementing a spreadsheet engine in Erlang, that allows cells to be evaluated concurrently. In this question a spreadsheet consists of a single *sheet* with a number of *cells* in it (we ignore the typical grid aspect of spreadsheets, and consider that a frontend issue).

We use the following terminology:

- A *sheet server* contains a mapping from names (atoms) to cell servers.
- A *cell server* is (at least) one Erlang process used for modelling the state of a single cell in a spreadsheet.
- Other cells' values can depend on the value of a given cell. And we can also have other processes that depends on the value of a cell, to provide an user interface, for instance. We call all processes (cell servers and others) that depends on the value of a given cell the *viewers* of a cell.

  Thus, a cell must keeps track of a set of *viewers* that must be notified when the cell is updated.
- The value of a cell depends on a (perhaps empty) collection of other cells, we call these the *dependencies* of the cell.
- A *viewer* is a process that is ready to receive *update messages*.
- An *update value* is either: a pair `{def, Any}` where the first element is the atom `def` and the second element can be any Erlang value; or one of the atoms `undefined`, `updating`, or `error`.
- An *update message* has the form `{updated, Name, UVal, Extra}`. That is, a record where the first element is the atom `updated`, the second element is an atom denoting the cell updated, the third element is an update value, and the forth element is extra information you can decide and use to detect various error or robustness situations (see the following).
- The value of a cell is *defined* when all of its dependencies are defined. If one of the dependencies of a cell, C, becomes undefined and thus sends an update message to C with the update value `undefined`, then the value of C is also undefined and C should send an update message to its viewers with the atom `undefined` as its update value.
- A *formula* is a triple $\{\texttt{formula}, F, Deps\}$, where the first element is the atom `formula`, the second element is a function, and the third element is a list of dependencies where each element is a an atom denoting a cell (note that *Deps* may contain duplicate elements).
- A cell, C, that contains the formula $\{\texttt{formula}, F, [A_1, \ldots, A_n]\}$, is evaluated to a value V by computing $F([V_1, \ldots, V_n])$, where V_i is the value of the cell denoted by A_i. The cell, C, must keep track of the values of its dependencies, by being a viewer of them,

and must re-evaluate *F* when one of the dependencies is updated. A formula is only evaluated when all dependencies are defined.

If it takes more than 500 milliseconds to evaluate *F*, then *C* should send an update message to its viewers with the atom `updating` as its update value. If a cell receive update message with the atom `updating` as the new value from one of its dependencies then it should send a similar message to all of its viewers (unless the *C* is already in an updating state).

If the function *F* throws an exception of kind `throw` or `exit`, then *C* should send an update message to its viewers with the atom `error` as its update value. If a cell receive update message with the atom `error` as the new value from one of its dependencies then it should send a similar message to all of its viewers (unless the *C* is already in an error state).

Implement an Erlang module, `sheet`, with the following client API, where *S* is a process ID for a sheet server and *C* is process ID for a cell server:

(a) A function `sheet/0` for creating a new sheet server. The function should start a new sheet server and return `{ok,`*S*`}` if it succeeds.

(b) A function `cell(`*S*`, `*A*`)` that returns `{ok, `*C*`}`, where *C* is the pid for the cell associated to the atom *A*. The function should start a new cell server the first time the function is called for each *A*.

(c) A function `add_viewer(`*C*`, `*P*`)` for adding a viewer *P* (a pid) to the cell *C*.

New viewers should get an update message about the current value of the cell, if the value is defined.

(d) A function `remove_viewer(`*C*`, `*P*`)` for removing the viewer *P* (a pid) from the viewers of the cell *C*.

(e) A blocking function `get_viewers(`*C*`)` that returns the list of viewers associated with the cell *C*.

(f) A non-blocking function `set_value(`*C*`, `*V*`)` for setting the value, *V*, of the cell *C*. The value can be either a formula or any other Erlang value.

If *V* is a formula, `{formula, `*F*`, `*Deps*`}`, then *C* should be added as a viewer of the cells in *Deps* (the atoms in *Deps* are consider to be cell names in the same sheet as *C*).

If *V* is not a formula, then that value should be sent to all viewers of *C*.

Remember that when a cell is updated it might need to remove itself as viewer of other cells.

In Appendix B you can find an example programs that shows how to use (parts of) the API.

You must document how your implementation handle (or mention that it does not handle) the following error or robustness situations:

- *Circular dependencies* is in the simplest case when two cell depend on each other. Your implementation can detect this by sending extra information in the forth element of an update message, for instance the set of all the cells used to compute the updated value. If a circular dependency is detected then the affected cells should go into an error state.
- *Long running formulas*, it must be possible to update a cell even while it is evaluating a formula. In general, evaluation of formulas should not be able to block the cell from interacting with clients.
- *Deadlocks*, your implementation should not have any, and you should have some argument why that is the case.
- A sheet server should only work when all of its cells are working.

Suggested approach

You might want to start by implementing a (rather boring) version of the `sheet` module where you do not handle formulas. You might want to leave out the timeouts in connection with evaluation of formulas. Also, you might want to leave out handling some of the robustness situations.

If you make such omissions make sure to clearly documenting them in your assessment, and explain why the omissions cause you problems and how you might handle them if you had more time.

Appendix A: Example Fast programs

Appendix A.1: `fact.fast`

```
class Fact {
  fact (n) {
    match n {
      0 -> { 1; }
      x -> { n * self.fact(n-1); }
    };
  }
}

class Main {
  new () {
    set f = new Fact();
    f.fact(0).printLn();
    f.fact(10).printLn();
  }
}
```

Appendix A.2: `fact.ast`

```
[ClassDecl {
    className = "Fact",
    classConstructor = Nothing,
    classMethods = [
      NamedMethodDecl "fact"
      (MethodDecl {
          methodParameters = ["n"],
          methodBody = [Match (ReadVar "n")
                        [(ConstInt 0,
                          [IntConst 1]),
                         (AnyValue "x",
                          [Times (ReadVar "n")
                           (CallMethod Self "fact"
                            [Minus (ReadVar "n") (IntConst 1)])])]]
          })],
    classReceive = Nothing},
 ClassDecl {
   className = "Main",
   classConstructor =
     Just (MethodDecl {
               methodParameters = [],
               methodBody = [
                 SetVar "f" (New "Fact" []),
                 CallMethod
                   (CallMethod (ReadVar "f") "fact" [IntConst 0])
                   "printLn" [],
                 CallMethod
                   (CallMethod (ReadVar "f") "fact" [IntConst 10])
                   "printLn" []]}),
   classMethods = [], classReceive = Nothing}]
```

Appendix A.3: `return.fast`

```
class Main {
  new () {
    self.fact(0).printLn();
    self.fact(10).printLn();
  }

  fact (n) {
    match n {
      0 -> { return 1; }
    };
    "This line should not be reached if n==0, because return ends the entire method.";
    n * self.fact(n-1);
  }
}
```

Appendix A.4: `return.ast`

```
[ClassDecl {
    className = "Main",
    classConstructor =
      Just (MethodDecl {
               methodParameters = [],
               methodBody = [
                 CallMethod (CallMethod Self "fact" [IntConst 0])
                   "printLn" [],
                 CallMethod (CallMethod Self "fact" [IntConst 10])
                   "printLn" []]}),
    classMethods = [
      NamedMethodDecl "fact"
      (MethodDecl {
          methodParameters = ["n"],
          methodBody = [
            Match (ReadVar "n") [(ConstInt 0,[Return (IntConst 1)])],
            StringConst
            "This line should...",
            Times (ReadVar "n")
            (CallMethod Self "fact"
             [Minus (ReadVar "n") (IntConst 1)])
            ]})
      ],
    classReceive = Nothing}]
```

Appendix A.5: `proxy.fast`

```
class Fact {
  fact (n) {
    match n {
      0 -> { 1; }
      x -> { n * self.fact(n-1); }
    };
  }
}

class Proxy {
  new (c, log) {
    set self.receiver = c;
    set self.log = log;
  }

  receive (msg) {
    match self.log {
      true() -> {
        "Method call:".printLn();
        msg.printLn();
      }
    };
    send(self.receiver, msg);
  }
}

class Main {
  new () {
    set f = new Fact();
    f.fact(10).printLn();
    set p1 = new Proxy(f, false());
    p1.fact(10).printLn();
    set p2 = new Proxy(f, true());
    p2.fact(10).printLn();
  }
}
```

Appendix A.6: `proxy.ast`

```
[ClassDecl {
    className = "Fact",
    classConstructor = Nothing,
    classMethods = [
      NamedMethodDecl "fact"
      (MethodDecl {
          methodParameters = ["n"],
          methodBody = [
            Match (ReadVar "n")
            [(ConstInt 0,[IntConst 1]),
             (AnyValue "x",
              [Times (ReadVar "n")
               (CallMethod Self "fact" [
                   Minus (ReadVar "n")
                   (IntConst 1)])])]]})
      ],
    classReceive = Nothing
    },
 ClassDecl {
   className = "Proxy",
   classConstructor = Just (
     MethodDecl {
        methodParameters = ["c","log"],
        methodBody =
          [SetField "receiver" (ReadVar "c"),
           SetField "log" (ReadVar "log")]}),
   classMethods = [],
   classReceive = Just (
     ReceiveDecl {
        receiveParam = "msg",
        receiveBody = [
          Match (ReadField "log")
          [(TermPattern "true" [],
            [CallMethod
               (StringConst "Method call:")
               "printLn" [],
             CallMethod (ReadVar "msg") "printLn" []])],
          SendMessage (ReadField "receiver") (ReadVar "msg")
          ]
        })
   },
 ClassDecl {className = "Main",
            classConstructor =
              Just (MethodDecl {
                       methodParameters = [],
```

```
                methodBody = [
                  SetVar "f" (New "Fact" []),
                  CallMethod (CallMethod (ReadVar "f") "fact" [IntConst 10])
                    "printLn" [],
                  SetVar "p1"
                    (New "Proxy" [
                        ReadVar "f",
                        TermLiteral "false" []]),
                  CallMethod
                    (CallMethod (ReadVar "p1") "fact" [IntConst 10])
                    "printLn" [],
                  SetVar "p2"
                    (New "Proxy" [
                        ReadVar "f",
                        TermLiteral "true" []]),
                  CallMethod
                    (CallMethod (ReadVar "p2") "fact" [IntConst 10])
                      "printLn" []]}),
classMethods = [],
classReceive = Nothing}]
```

Appendix A.7: `observable.fast`

```
class Observable {
  new (value) {
    set self.value = value;
    set self.observers = nil();
  }

  addObserver (obj, cookie) {
    set self.observers = cons(observer(obj, cookie), self.observers);
  }

  setValue (value) {
    set self.value = value;
    self.notifyObservers(self.observers);
  }

  notifyObservers (xs) {
    match xs {
      cons(x, xs) -> {
        "Note - new xs shadows the old one";
        match x {
          observer(obj, cookie) -> {
            obj.notify(cookie, self.value);
          }
        };
        self.notifyObservers(xs);
      }
    };
  }
}

class Observer {
  notify(cookie, newval) {
    "Changed:".printLn();
    cookie.printLn();
    "New value".printLn();
    newval.printLn();
  }
}

class Main {
  new () {
    set box = new Observable(0);
    set obs1 = new Observer();
    set obs2 = new Observer();
    box.setValue(1);
```

```
    box.addObserver(obs1, obs1());
    box.setValue(2);
    "".printLn();
    box.addObserver(obs2, obs2());
    box.setValue(3);
  }
}
```

Appendix A.8: `observable.ast`

```
[ClassDecl {
    className = "Observable",
    classConstructor = Just (
      MethodDecl {
         methodParameters = ["value"],
         methodBody = [
           SetField "value" (ReadVar "value"),
           SetField "observers" (TermLiteral "nil" [])]}),
    classMethods = [
      NamedMethodDecl "addObserver"
      (MethodDecl {
          methodParameters = ["obj","cookie"],
          methodBody = [
            SetField "observers"
            (TermLiteral "cons"
             [TermLiteral "observer"
              [ReadVar "obj",
               ReadVar "cookie"],
              ReadField "observers"])]}),
      NamedMethodDecl "setValue"
      (MethodDecl {
          methodParameters = ["value"],
          methodBody = [
            SetField "value" (ReadVar "value"),
            CallMethod Self "notifyObservers" [ReadField "observers"]]}),
      NamedMethodDecl "notifyObservers"
      (MethodDecl {
          methodParameters = ["xs"],
          methodBody = [
            Match (ReadVar "xs")
            [(TermPattern "cons" ["x","xs"],
              [StringConst "Note - new xs shadows the old one",
               Match (ReadVar "x")
               [(TermPattern "observer" ["obj","cookie"],
                 [CallMethod (ReadVar "obj") "notify"
                    [ReadVar "cookie",
                     ReadField "value"]])],
               CallMethod Self "notifyObservers" [ReadVar "xs"]])]})],
    classReceive = Nothing},
 ClassDecl {
   className = "Observer",
   classConstructor = Nothing,
   classMethods = [
     NamedMethodDecl "notify"
     (MethodDecl {
```

```
        methodParameters = ["cookie","newval"],
        methodBody = [
          CallMethod (StringConst "Changed:")
            "printLn" [],
          CallMethod (ReadVar "cookie")
            "printLn" [],
          CallMethod (StringConst "New value")
            "printLn" [],
          CallMethod (ReadVar "newval")
            "printLn" []]})],
  classReceive = Nothing
  },
ClassDecl {
  className = "Main",
  classConstructor = Just (
    MethodDecl {
       methodParameters = [],
       methodBody = [
         SetVar "box" (New "Observable" [IntConst 0]),
         SetVar "obs1" (New "Observer" []),
         SetVar "obs2" (New "Observer" []),
         CallMethod (ReadVar "box")
           "setValue" [IntConst 1],
         CallMethod (ReadVar "box")
           "addObserver" [
             ReadVar "obs1",
             TermLiteral "obs1" []],
         CallMethod (ReadVar "box")
           "setValue" [IntConst 2],
         CallMethod (StringConst "")
           "printLn" [],
         CallMethod (ReadVar "box")
           "addObserver" [
             ReadVar "obs2",
            TermLiteral "obs2" []],
         CallMethod (ReadVar "box")
           "setValue" [IntConst 3]]}),
  classMethods = [],
  classReceive = Nothing
        }]
```

Appendix B: Program using the sheet module

```
print_viewer() ->
    receive
        {updated, Name, {def, Val}, _} ->
            io:format("~p has value ~p~n", [Name, Val]),
            print_viewer();
        _ -> print_viewer()
    end.

sum2() ->
    {ok, Sheet} = sheet:sheet(),
    UI = spawn(fun print_viewer/0),
    {ok, Sum} = sheet:cell(Sheet, sum),
    sheet:add_viewer(Sum, UI),
    sheet:set_value(Sum, {formula, fun lists:sum/1, [a, b]}),
    {ok, A} = sheet:cell(Sheet, a),
    {ok, B} = sheet:cell(Sheet, b),
    sheet:set_value(A, 23),
    sheet:set_value(B, 19),
    {Sheet, Sum, A, B, UI}.
```