轻闪PDF

Question 1: Agner: A concurrency tracer

程序代写代做 CS编程辅导

Concurrency bugs, including especially deadlocks and race conditions, are notoriously difficult to replicate and diagnose, even in otherwise well-behaved, message-oriented languages such as E[illegible]m is that, when multiple concurrent threads are executing, the relativ[illegible]ents may depend unpredictably on factors such as the compiler versio[illegible] of cores available, and other factors that are not easy to vary, and tha[illegible] in the future. For example, consider the Erlang top-level expression[illegible]

```
P = self(), spawn(fun () -> P ! a end), P ! b, receive X -> X end.
```

WeChat: cstutorcs

In most current implementations, this would probably evaluate to b every time it is run. However, if there had been more code between the spawn and the sending of b, such as some nontrivial computations, or just `timer:sleep(1)`, the result would be likely to change to a. In even slightly more complicated examples it quickly becomes infeasible for humans to consider all the possible execution scenarios and verify that they all lead to an acceptable outcome.

Assignment Project Exam Help

However, because of the inherent simplicity and cleanliness of the Erlang concurrency model, and exploiting the expressive power of Haskell with monads, it is quite feasible to construct a usable *concurrency tracer* that systematically explores the consequences of various scheduling choices and enumerates the ultimate possible outcomes, including step-by-step scenarios of how each outcome could arise. This task is about implementing a simple such tracer named Agner.

Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com
轻闪PDF

An overview of the Agner language

The syntax and semantics of Agner are closely based on Erlang, though quite a bit simpler. Mostly, Agner programs will have the same meanings as in Erlang, but there are some important differences. This overview presupposes a working knowledge of Erlang.

The sequential core

Like Erlang, Agner is an evaluation-oriented language, with a single-assignment store. The following expression forms are available (the exact syntactic and lexical rules will be specified later):

Atoms An atom is either a simple identifier starting with a lowercase letter, such as `hello`, or an (almost) arbitrary sequence of characters enclosed in single quotes, such as `'Hello, world!'`. The quotes are not part of the atom name, so that `'hello'` and `hello` denote exactly the same atom. Atoms evaluate to themselves.

Variables Variables are simple identifiers starting with an uppercase letter or underscore. A variable can have a current value in the store, in which case it will evaluate to that value; or it can be unbound, in which case evaluating it signals an exception `{badvar, 'X'}`, where `X` is the name of the variable. (In AGNER, unlike in ERLANG, unbound variables are detected only at runtime, rather than before execution.) Variables get values by pattern-matching (see below); once a variable has a value, that value cannot change.

Tuples A *tuple expr* parated, possibly empty, sequence of expressions, enclosed in cu `foo,X,{}}`. Like in ERLANG, such an expression simply evaluat lues that its elements evaluate to.

Lists A *list expressi* the form `[Exp1 | Exp2]`. Like in ERLANG, the generalized fo `pn]` is syntactic sugar for

```
[Exp1 | [Exp2 | ... [Expn | []]]]
```

and `[Exp1, ..., Expn | ExpT]` for

```
[Exp1 | [Exp2 | ... [Expn | ExpT]]]
```

(As usual there is no requirement that `ExpT` must evaluate to a list.) In AGNER, unlike in ERLANG, lists are actually not a separate data type, but just an abbreviation, with `[]` representing the *atom* `'[]'`, and `[Exp1 | Exp2]` representing the *tuple* `{'[|]', Exp1, Exp2}`. For example `[a]` is just a more convenient notation for `{'[|]', a, '[]'}`.

Patterns and matches A pattern, like in ERLANG, is a restricted expression built out of atoms, variables, and tuples (with list-notation desugared like for expressions). Additionally, the wildcard pattern `_` acts like an anonymous variable (with each occurrence distinct). For example, `{a, X, [H|_]}` is a pattern. Also like in ERLANG, and unlike in HASKELL, a variable may occur multiple times in a pattern, as in `{X,X}`.

A *match expression* is of the form `Pat = Exp`. When evaluated, it evaluates `Exp` to a value and checks that that this value is of the shape described by the pattern, possibly binding variables occurring in the pattern. It also returns the value. For example, `{a, X, [H|_]} = {a, b, [foo, bar, baz]}` will succeed and bind `X` to `b` and `H` to `foo`. If the pattern does not match the value, the exception `{badmatch,V}` is thrown, where `V` is the value that the expression evaluated to

If variable in the pattern is already bound, the value it would be (re)bound to must be the same as its current value. For example, after `X = a`, the pattern `{X,b}` matches `{a,b}`, but not `{c,b}`. In particular, `{X,X} = {Exp1,Exp2}` (where `X` is previously unbound), succeeds if and only if `Exp1` and `Exp2` evaluate to the same value.

Sequences A *sequence expression*, often referred to as a *body*, is of the form

```
Exp1, ..., Expn
```

(where $n \geq 1$). It evaluates all of the expressions in turn (possibly binding variables in matches), and returns the value of the last one. (If any of the expressions signal an exception, any bindings are undone.). An expression sequence can appear anywhere in an expression by enclosing it as `begin Exp1, ..., Expn end`, but many places in the grammar also allow sequencing without an explicit `begin`.

轻闪PDF
程序代写代做CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

轻闪PDF
程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

Note that in Agner, unlike in Erlang, it is explicitly specified that all subexpressions within a single expression are also evaluated from left to right. This means that, e.g., `{X=a,X}` evaluates to `{a,a}`, while `{X,X=a}` signals that `X` is unbound. (In Erlang, both expressions are statically illegal.)

Case analysis A *ca*[illegible] e form

```
case Exp [illegible] ...; Patn -> Bodyn end
```

(Erlang's case [illegible] orted.) Like in Erlang, first `Exp` is evaluated to a value `V`, which [illegible] ainst all the patterns in turn. If `V` matches some `Pati` (possibly [illegible] bles), the corresponding `Bodyi` is evaluated and its result becomes the result of the case-expression. (Any variables bound by the expression, pattern-match, or body also remain bound after the case-expression.) If none of the patterns match, the case-expression signals the exception `{badcase,V}`.

In Erlang, variables bound by a `Bodyi` are in general *not* available after the case-expression, unless they are bound by *all* the cases. (Variables that fail this check are called *unsafe*, and cause compilation to fail.) Agner, which only detects unbound variables at runtime, imposes no such restrictions.

Functions A *function call* is an expression of the form `Exp0(Exp1,...,Expn)`, where $n \geq 0$. Agner, like Erlang (and most other languages, except Haskell), has an eager semantics, so first all of the expressions are evaluated (from left to right), and then the function that `Exp0` evaluates to is applied to the list of argument values.

Like in Erlang, there are two kinds of functions in Agner, *named* and *anonymous*. For the former, if `Exp0` evaluates to (usually, is already) an atom `f`, the program must contain a *declaration* of a function with that name, of the form,

```
f(X1, ..., Xn) -> Body
```

Then the formal parameters (the `X`'s) are bound to the values of the corresponding actual parameters, the function body is evaluated, and its result becomes the result of the function call. Function declarations may be (mutually) recursive, as usual, and the functions may be declared in any order.

Note that, unlike in Erlang, there can only be a single clause in a function declaration, rather than a collection of `;`-separated ones. Consequently, the function parameters must also be simple variable names, not more general patterns. Likewise, all the parameter names must be distinct in Agner; this is considered a *syntactic* restriction. Erlang's multiple-clause functions can expressed, slightly more verbosely, by explicit case-expressions; for example, the Erlang function declaration

```
equal(X,X) -> true; equal(_,_) -> false.
```

could be written in Agner as

```
equal(X,Y) -> case X of Y -> true; _-> false end.
```

Agner, like Erlang (and Haskell) is *statically scoped*. This means that any variables bound at the place where the function is called are *not* available in the function body. For example, with the declaration

轻闪PDF

```
foo(X) -> Z = {X,X}, Z.
```

the expression (sequence) Z = a, X = b, foo(c) will still evaluate to {c,c}, despite the seemingly conflicting binding for Z. Conversely, in foo(c), {X,Z}, the caller's X and Z will remain unbound after the call, unless they already had values.

Unlike in ERLA[illegible] can only have a single arity; that is, there cannot be declaration[illegible] ... and foo(X,Y) -> ... in the same program. Also, there ar[illegible] declared functions live in the same name space, shared with th[illegible] (BIFs). (However, there is a lexical hack for : that allows for bett[illegible]patibility with ERLANG; see later.)

If the number [illegible] in a function call does not agree with the number of formal parameters in the declaration, the exception {badarity, F} is signaled, where F is the function name. If the atom F is not declared as a function all, the exception will instead be {undef, F}. If F is neither an atom nor an anonymous function value, the exception is {badfun, F}.

The second kind of functions are the anonymous *function values*, produced by *function expressions*, fun (X1, ..., Xn) -> Body end. Calling an anonymous function is very similar to calling a named one; for example,

```
F = fun(X) -> {X,X} end, F(a).
```

would evaluate to {a,a}. There is, however, a subtlety in that – again in accordance with static scoping – variables that are *already bound* at the place where the function-expression is evaluated (*not* where the resulting functional value is later called) are also available in the function body. For example,

```
F = fun(X) -> fun (Y) -> {X,Y} end end, G = F(a), G(b).
```

would evaluate to {a,b} in both ERLANG and AGNER. Note that formal parameters always shadow any existing variables of the same name, so that

```
X = a, F = fun(X) -> X end, {F(b), X}.
```

evaluates to {b,a} as expected. However, other variables, even if they are seemingly "defined" in the function body by the pattern-matching syntax, may clash with already existing bindings, so that (like in ERLANG),

```
Y = a, F = fun(X) -> Y = X, {X,Y} end, F(b).
```

would actually signal a match exception. (But had the final call been F(a), the result would still be {a,a}).

Functional values always compare as equal to themselves. For example,the match in F = fun(X) -> X end, F = F. would always succeed. It is *unspecified* whether two function values produced by *different* evaluations of identical or similar function expressions compare as equal. For example all of these may succeed or fail:

```
F = fun(X) -> X end, G = fun(X) -> X end, F = G.
F = fun(X) -> X end, G = fun(Y) -> Y end, F = G.
F = fun(X) -> X end, D = a, G = fun(X) -> X end, F = G.
```

程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

Exceptions Unlike Erlang, Agner does not distinguish formally between various exception classes, such as errors or thrown exceptions. Any Agner value (though usually an atom or a tuple starting with an atom) can be signaled as an exception using the BIF `throw`, e.g, `throw({myErr,{X,Y}})` (where `X` and `Y` are already bound). Likewise, many runtime [illegible] ecific exceptions, as mentioned above. Unless it is caught, an exc[illegible] tion of the entire expression.

Exceptions are [illegible] essions, with syntax

```
try Body[illegible] Body1; ...; Patn -> Bodyn end
```

Like in Erlan[illegible] uated, and if this succeeds (i.e., does not signal an exception), the result becomes the result of the whole `try` (and the handler cases are ignored). But if `Body0` signals an exception, the exception value is matched against the patterns in turn (just like in a case-expression), and if a match succeeds, the corresponding handler body is evaluated. (Any exceptions thrown by the handler are not themselves caught by *this* `try`, but may be handled by an outer one.) If none of the patterns match, the exception propagates to any enclosing handlers, or ultimately to the top level of the expression.

If any variables were bound by `Body0` before the exception was signaled, those bindings are undone; but any bindings existing before the `try` are kept. For example, after

```
X = a, try Y = b, throw(e) catch e -> Z = c end.
```

`X` will be bound to `a` and `Z` to `c`, but `Y` will be unbound.

The sequential subset of Agner is a reasonably expressive functional language in its own right. The main difference from Erlang is the lack of any significant library of functions such as `lists:map`; however, such functions can easily be defined explicitly using pattern-matching and recursion. Likewise, but with more effort, rudimentary integer arithmetic could be coded as a library, either using an unary representation, e.g, representing 3 as `{{{{}}}}`, or – far more efficiently – as a list of bits, such as `[one,one]`. However, the real interest of Agner is of course in the concurrency features, described next.

Concurrency

Agner shares Erlang's basic concurrency model: a concurrent system consists of a collection of processes that communicate only by sending messages to each other. Each process has an incoming message queue, and once delivered to the queue, all messages will be processed in the order in which they were delivered. Also, a process may selectively receive only messages of a certain form, as specified by patterns. The additional expression forms and BIFs relating to concurrency are as follows:

Process spawning The BIF `spawn(F)`, where `F` is either a named or an anonymous function of arity 0, starts a new process computing `F()`. The ultimate value (or thrown exception) computed like this is discarded, but while it is running, the process may engage in communication with other processes. The `spawn` call returns the

轻闪PDF
程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

轻闪PDF

程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com
Tutor CS

pid (process id) of the newly spawned process. A pid is a special value that can be bound to variables and tested for equality against other values, but that cannot be otherwise constructed. A pid is guaranteed to be distinct from any other pid issued in the same run of the program (i.e. a pid is *not* reissued even if the process it originally referenced has [illegible]

In Agner, pids [illegible] representation as sequences of numbers `<n1.n2...nk>`. The initial pr[illegible] the processes it spawns is called `<0>`, `<1>`, etc. Likewise, proc[illegible] `<1.0>`, `<1.1>`, and so on; But the structured pids have no seman[illegible] as linking or supervision). Also, it is not possible to "forge" a pi[illegible] d atom, as in `'<1.2>' ! a`.

Message send A *send expression* has the syntax `ExpP ! ExpM`. Here, `ExpP` must evaluate to a P that is a (possibly defunct) pid (otherwise, `{badpid, P}` is signaled), while `ExpM` can evaluate to any Agner value M. M is delivered to to the message queue of the process P, if it is still alive; otherwise the message is just discarded. Messages sent by a process are delivered in the same order in which they were sent, but if two *different* processes are sending to the same pid, it is unspecified which one gets delivered first. In particular, there are no inherent *fairness* guarantees (e.g., in a denial-of-service scenario, where one process is uninterruptedly sending messages to a single pid, any other sends to that pid may never get through.)

Message receive A *receive expression* is more involved, having the form

```
receive Pat1 -> Body1; ...; Patn -> Bodyn end
```

Like for case-expressions in Agner, there are no guards, nor is possible to specify an `after` (i.e., timeout) case.

The semantics of the expression is that the process first sleeps until its message queue becomes non-empty. Then, it takes the first message from the queue and matches it against all the patterns in turn, like for a `case`. But if none of the patterns match, the message is *not* immediately put back on the queue, since trying to match it again against the same patterns would be pointless. Instead, the message is *stashed* in an internal list inside the process, and the next message from the incoming queue is examined. (If the queue has become empty, the process goes back to sleep.) Once a message is received that matches one of the patterns, the process first *unstashes* its internal list back unto the *front* of the message queue (preserving the original order), so that the *next* receive-expression will examine all the messages again from the oldest one.

Self-identification A process can find out its own pid (tyically to give it to someone else) by the BIF `self()`.

Logging A final, Agner-specific BIF (but roughly corresponding to Erlang's. `io:format/2`, when used for debugging information, though without any actual formatting functionality) is `log(V)` where `V` can be an arbitrary value. This adds adds (a textual representation of) `V` to the end of an event log that can be printed once the program stops (whether normally or by a deadlock). Also, the Agner system itself is allowed to add its own messages to the log; such messages will be specially tagged so as not to interfere with the user's. The `log` function always returns the atom `ok`.

轻闪PDF

Log entries produced by the same process are recorded in the order in which they are made, but messages from different processes that are not causally related may be arbitrarily interleaved.

程序代写代做 CS编程辅导

For simplicity, Agne[illegible]*eemptive* multi-threading: a well-behaved process is expected to eventu[illegible] or receive, at which time a different process may be scheduled. Thus, [illegible] a tight loop may cause the whole system to grind to a halt. Also, “fork [illegible]s spawning unbounded numbers of subprocesses), like infinite recursio[illegible]racefully in the basic version.

Examples

We can use the system to run the example from the introduction

WeChat: cstutorcs

```
$ agner "P = self(), spawn(fun () -> P ! a end), P ! b, receive X -> X end."
===== result : value b (with deterministic scheduler)
```

Assignment Project Exam Help

This (coincidentally) agrees with the standard Erlang implementation. However, we can also explore the space of possibilites using the backtracking scheduler, which systematically tries all possible interleavings of runnable processes:

Email: tutorcs@163.com

```
$ agner -bt "P = self(), spawn(fun () -> P ! a end), P ! b, receive X -> X end."
===== result : value b (1 scenarios)
===== result : value a (1 scenarios)
```

QQ: 749389476

If it seems puzzling how some of the final results can arise, we can ask the system to print out its detailed event log leading to a particular outcome. (If there are multiple ways to reach a particular result, only one, representative, scenario is shown.)

https://tutorcs.com

```
$ agner -sl -bt "P = self(), spawn(fun () -> P ! a end), P ! b, receive X -> X end."
===== result : value b (1 scenarios)
  system: Process <> spawning process <0>
  system: selecting #1 of 2 ready processes
  system: process <> sending b to self
  system: Process <> inspecting b
  system: Process <> unstashed 0 messages
  system: Process <> terminated with value b
  system: finished!
===== result : value a (1 scenarios)
  system: Process <> spawning process <0>
  system: selecting #2 of 2 ready processes
  system: process <0> sending a to ready <>
  system: Process <0> terminated with value a
  system: process <> sending b to self
  system: Process <> inspecting a
  system: Process <> unstashed 0 messages
  system: Process <> terminated with value a
  system: finished!
```

轻闪PDF

Note that the actual contents of the system log is *not* specified; you can be as chatty or silent as you prefer, and the exact phrasing of particular events is also entirely up to you.

程序代写代做 CS编程辅导

For a slightly larger example, we may define some helper functions to be used in the top-level expression. We also select a Monto-Carlo scheduler that makes random scheduling choices. (Newlines a[illegible]mand line for readabilty):

```
$ agner -mc 1000 [illegible]
        -p "r() -[illegible] end.
            ss(P,[illegible] () -> P ! X end)."
        "P = self[illegible]P,b), ss(P,c), {r(), r(), r()}."
===== result : va[illegible] {b,a,c} (163 / 1000 scenarios)
===== result : value {a,b,c} (173 / 1000 scenarios)
===== result : value {b,c,a} (183 / 1000 scenarios)
===== result : value {c,a,b} (169 / 1000 scenarios)
===== result : value {c,b,a} (157 / 1000 scenarios)
===== result : value {a,c,b} (155 / 1000 scenarios)
```

WeChat: cstutorcs

Running the tool with no arguments gives a brief usage summary.

Assignment Project Exam Help

The Agner implementation

Email: tutorcs@163.com

The Agner system is divided into 4 main modules (as well as a couple of extra ones, containing shared type definitions and the like), as follows:

QQ: 749389476

- The Parser reads in the concrete syntax of Agner programs and top-level expressions into abstract syntax trees.

https://tutorcs.com

- The Evaluator handles the sequential part of the language, It also contains part of the support for the concurrency-related features with significant syntax-related aspects, specifically receive-expressions.
- The Scheduler provides the main support for the concurrent features of the language. It orchestrates the sequential evaluation of expressions in the individual processes, as well as handling the communications aspects. It also allows various scheduling choices to be explored.
- The main program, which contains the top-level stand-alone tool (including command-line option processing, file I/O, log filtering, and the like).

The main program is already provided, so your task is to implement the remaining three modules, according to the detailed specifications in the following. The sections are weighted roughly as indicated below; but as always, the exam is graded as a whole, and the various modules demonstrate complementary skills.

轻闪PDF

程序代写代做 CS编程辅导

```
Program ::= epsi
         |  FDecl

FDecl ::= atom "(                    Body "."

Top ::= Body "."

Exp ::= atom                                      Pat ::= atom
     |  var                                            |  var
     |  "(" Exp ")"                                    |  "(" Pat ")"
     |  "{" Expz "}"                                   |  "{" Patz "}"
     |  "[" ExpL "]"                                   |  "[" PatL "]"
     |  Pat "=" Exp                                    |  "_"
     |  "begin" Body "end"
     |  "case" Exp of Cases "end"
     |  Exp "(" Expz ")"
     |  "fun" "(" Varz ")" "->" Body "end"
     |  "try" Body "catch" Cases "end"
     |  Exp "!" Exp
     |  "receive" Cases "end"

Cases ::= Case
       |  Case ";" Cases

Case ::= Pat "->" Body

Body ::= Exps

Expz ::= epsilon          Patz ::= epsilon          Varz ::= epsilon
      |  Exps                   |  Pats                   |  Vars

Exps ::= Exp              Pats ::= Pat              Vars ::= var
      |  Exp "," Exps           |  Pat "," Pats           |  var "," Vars

ExpL ::= epsilon          PatL ::= epsilon
      |  Exps                   |  Pats
      |  Exps "|" Exp           |  Pats "|" Pat
```

Figure 1: Concrete grammar of AGNER

WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com
轻闪PDF

轻闪PDF

Question 1.1: The Parser (40%)

程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

The concrete syntax of AGNER programs is given in Figure 1 (in a notation similar to the parser notes). This grammar is supplemented with the following stipulations:

Disambiguation [illegible] nd "!" have the same precedence and are right-associative. Functio[illegible] higher precedence than match and send, so that, e.g., `self() ! sel`[illegible]`f()) ! (self())`. Also, slightly unusually for a functional langua[illegible]ion is non-associative, so that `f(a)(b)` is illegal syntax; it must be w[illegible]

Complex lexical tokens An *identifier atom* is a non-empty sequence of (Unicode) letters, digits(`0` through `9`), underscores (`_`) and at-signs (`@`), starting with a lowercase letter. The following AGNER keywords are not allowed as identifiers: `begin`, `case`, `catch`, `end`, `fun`, `receive`, and `try`. Other ERLANG keywords, such as `after` are allowed, though it may be prudent to avoid them.

For pseudo-compatibilty with ERLANG, colons (`:`) are also allowed as identifier characters (similarly to `@`), so that one could directly declare and call, e.g., a utility function named `lists:reverse`.

Alternatively, a *general atom* is any (possibly even empty) sequence of characters, enclosed in single quotes (`'`). If this sequence is to contain single-quote or a backslash (`\`) characters, they must be written as `\'` and `\\`, respectively. The escape sequences `\n` and `\t` stand for a newline and tab character; all others stand for just the second character. The outer quotes are not included in the atom name.

The rules for variable names var are the same as for identifier atoms (including the special inclusion of `:`, except that the first character must now be an uppercase letter or an underscore. (However, it cannot consist of *only* an underscore, since that is reserved for the wildcard pattern.) There is no analog of the general-atom syntax for variables.

Whitespace and comments Tokens may be surrounded by arbitrary whitespace, which is ignored, except for separating tokens that would otherwise run together, such as consecutive identifiers, variables, and keywords. Comments start with a percent sign (`%`) and run until the following newline (or end of file). Comments also act as whitespace for token separation.

The abstract syntax is defined in file `AST.hs`:

```
data Exp =
    EAtom AName
  | EVar VName
  | ETup [Exp]
  | EMatch Pat Exp
  | ESeq Exp Exp
  | ECase Exp Cases
  | ECall Exp [Exp]
```

轻闪PDF

```
  | EFun [VName] Exp
  | ETry Exp Cases
  | ESend Exp Exp
  | EReceive Cases

data Pat =
    PAtom AName
  | PVar VName
  | PTup [Pat]
  | PWild

type Cases = [(Pat, Exp)]

type AName = String
atomNil = "[]"
atomCons = "[|]"

type VName = String

data FDecl = FDecl AName [VName] Exp

type Program = [FDecl]
```

程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

The correspondence between the concrete and abstract syntax should be largely straightforward. Note, however, that there is no separate AST type corresponding to Body; instead, commas used for sequencing correspond to the ESeq constructor.

The Parser should export two functions, corresponding to the two possible start symbols of the grammar:

```
parseProgram :: String -> Either String Program
parseTop :: String -> Either String Exp
```

Note that the parsing functions should also check that all functions are well formed, i.e., that there are no repeated variable names in a formal-parameter list.

For implementing your parser, you may use either the ReadP or the Parsec combinator library. 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.Parsec); in particular you are *disallowed* to use Text.Parsec.Token, Text.Parsec.Language, and Text.Parsec.Expr. As always, don't worry about providing informative syntax-error messages if you use ReadP.

With ReadP's lack of error messages, it may be a bit challenging to track down the location of syntax errors in large programs. However, since Agner syntax is very close to Erlang, you may be able to use the standard Erlang compiler on the failing file to see where the error is.

轻闪PDF

Question 1.2: The Evaluator (35%)

程序代写代做 CS编程辅导

The evaluation model of AGNER is value-oriented, where a value is given by the following datatype (in Runtime.hs):

```
data Value =
    VAtom AName
  | VTup [Value]
  | VClos Env [V
  | VPid Pid

type Env = [(VName,Value)]

type Exn = Value
mkExn s v = VTup [VAtom s, v]

type Outcome = Either Exn Value
```

WeChat: cstutorcs

Assignment Project Exam Help

The meanings of the constructors VAtom, VTup, and VPid should be obvious. VClos is a *function closure*, used for functional values. It consists of the formal parameters and body of the function expression, together with a copy of the environment at the time the function-expression was evaluated. An environment is simply a list of variables and their corresponding values, if any; unbound variables are not mentioned. As usual, only the first occurrence of a variable in the list matters.

Email: tutorcs@163.com

QQ: 749389476

Finally, an AGNER exception is simply a value; we also define a convenience function for constructing exception values in the standard format. Outcome is a convenient abbreviation for the final result of an evaluation.

https://tutorcs.com

(The file also includes a simple pretty-printer showV for rendering values in a more readable form. To avoid potential confusion, it is *not* used as the default Show instance. The concurrency-related definitions in Runtime will be discussed later.)

The central type definition in the Evaluator is the Eval monad:

```
newtype Eval a = Eval {runEval :: FEnv -> Env -> Req (Either Exn (a, Env))}

type FEnv = [(AName, (Int, [Value] -> Eval Value))]
```

An Eval-computation may draw upon a number of features of the evaluation context. First, it has Reader-like access to the *function environment*, in which all available *named* functions (both program-defined and built-in) are defined. The environment binds each such function name to a tuple of the function's arity and the function itself, represented as taking a list of argument values, and returns (a computation of) the result value.

Second, an Eval computation receives the current *(value) environment*, as previously mentioned. Since expression evaluation may add bindings to the environment (using the match-operator =), the updated environment is also returned together with the (successful) computation result.

轻闪PDF

Since evaluation may fail, the result is actually an Either-type, with the left alternative being a thrown exception. Note that no updated environment is returned in this case; rather, the set of available bindings is rolled back to what it was when the failing computation was started in a try-expression.

程序代写代做 CS编程辅导

Finally, the entire c [illegible] rmed in the Req monad, a generalization of the SimpleIO monad fr [illegible] used only by the concurrency features discussed later; so for now, yo [illegible] s simply the identity monad.

Your first (minor) tas [illegible] Monad instance for Eval. This should not depend on what Req is, but [illegible] e monad.

Next, as usual, we d [illegible] accessing the features of the monad:

```
askFEnv :: Eval FEnv

getVar :: VName -> Eval Value
setVar :: VName -> Value -> Eval ()
getEnv :: Eval Env
inEnv :: Env -> Eval a -> Eval a

raise :: Exn -> Eval a
handle :: Eval a -> (Exn -> Eval a) -> Eval a

request :: Req a -> Eval a
wrapup :: FEnv -> Eval Value -> Req (Either Exn Value)
```

WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

askFEnv simply returns the function environment.

getVar *x* returns the binding of variable *x* in the current (value) environment. If *x* is unbound, the function signals the exception mkExn "unbound" (VAtom *x*). Conversely, setVar *x* *v* binds *x* to the value *v*, unless *x* is already bound to a *different* value; in the latter case, the analogous exception with "bound" is signaled.

getEnv returns the current value environment. The companion operation inEnv *env* *m* runs the computation *m* in the environment *env* (and the unmodified function environment), and returns its result (value or raised exception). The current environment is *not* modified.

raise *ex* signals the exception *ex*. Conversely handle *m* *h* first tries to run the computation *m*. If that succeeds, the result is also returned from handle. However, if *m* raises an exception *ex*, the result is given by applying *h* to *ex*.

Finally, request *r* performs the request *r* in the inner Req monad, and returns its result. And wrapup fenv *m* runs the computation *m* in function environment *fenv* and the empty value environment; it returns (as a Req-computation) the final result (value or exception), and simply discards the final environment.

The remainder of the Evaluator (and the other modules) should not depend on the definition of the Eval monad, but only rely on the above accessor functions.

Using the monad, we first implement a central Agner operation:

轻闪PDF

```
match :: Pat -> Value -> Eval ()
```

程序代写代做 CS编程辅导

`match` *p* *v* attempts to match the value *v* against the pattern *p*. If it succeeds, it updates the environment with any new bindings and returns just `()`; if it fails, it raises `mkExn "badmatch" v`. (Thi [illegible] ceptions are as specified in the previous overview of AGNER.)

Then, we define the [illegible] r as three mutually recursive functions:

```
evalExp :: Exp -> [illegible]
apply :: Value -> [illegible] Value
evalCases :: Cases [illegible] al () -> Eval Value -> Eval Value
```

The first is the main function for evaluating an expression; it has the obvious functionality. `apply` *vf* *vs* calls the function specified by *vf* (either an atom or an anonymous function value) on the arguments *vs*, raising exceptions for various error conditions, as previously specified.

WeChat: cstutorcs

`evalCases` *cs* *v* *hook* *nm* matches the value *v* against the patterns in *cs* and evaluates the corresponding expression in the first matching case. The extra computation *hook* is run *after* the matching has successfully completed, but *before* the selected body expression. (If no such computation is relevant, it can be simply specified as `return ()`.) If no case matches the value, the computation *nm* is run instead; *hook* is *not* invoked in this case.

Assignment Project Exam Help

Email: tutorcs@163.com

Finally, the Evaluator defines:

QQ: 749389476

```
fenv0 :: FEnv
evalTop :: Program -> Exp -> Req Outcome
```

https://tutorcs.com

`fenv0` is the initial function environment, containing bindings for the BIFs. (A sample implementation for `log` is already provided) You should add the remaining ones. You *may* also include additional BIFs, but your black-box tests should not rely on any such non-standard functions being available. Any definitions in the program override the BIFs; this is rarely a good idea.

Finally, `evalTop` *pgm* *e* runs the top-level expression *e* with the function declarations in *pgm*. If *e* (and the program functions it calls) do not involve any concurrency features, the computation should just be the trivial `Req`-computation of the final result of the expression.

A fair bit of the evaluator can be implemented without using `Req`: pattern matching, case-expressions, functions (both named and anonymous), and exceptions. However, for send, receive and spawn, need to look at `Req` in the next section.

Question 1.3: The Scheduler (25%)

The central interface between the Evaluator and the Scheduler is the `Req` monad:

```
data Req a =
    RDone a
```

轻闪PDF

```
  | RLog String (Req a)
  | RSend Pid Msg (Req a)
  | RRcv (Msg -> Req a)
  | RUnstash [Msg] (Req a)
  | RSpawn (Req ([illegible]Req a)
  | RSelf (Pid -[illegible]

type Msg = Value [illegible]

newtype Pid = Pi[illegible]
```

程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

A Req-computation result is either an already computed value (RDone), or a request to perform some operation, together with a further computation to perform once that operation has completed. That further computation may also depend on the result of the previous operation, where relevant. The meanings of the various requests are as follows:

- RLog *s* *r*: Add string *s* to the log, then continue with *r*.
- RSend *p* *m* *r*: Send message *m* to pid *p*, then continue with *r*.
- RRcv *h*: Wait for the requesting process's message queue to become non-empty, then retrieve the first (i.e., oldest) message from the queue and pass it to the function *h*, which will determine the next request, if any. Note that it becomes the responsibility of the caller to preserve the retrieved message if it cannot be immediately processed (for example, if it doesn't match any of the patterns in the receive that led to the RRcv request). Normally, such messages are *stashed*, to be looked at later.
- RUnstash *q* *r*: Add the previously stashed messages *q* back to the *front* of the caller's message queue, so that they will be retrieved again. While it is in principle possible to unstash messages that had been not previously received (i.e., in effect, send a priority message to oneself) using this mechanism, that is considered poor practice, as it makes interpreting the traces event traces confusing.

  On the other hand, it is often useful to explicitly unstash an empty queue, to indicate that the immediately previously retrieved message was in fact accepted for further processing, and will not need to be considered again.
- RSpawn *r* *h*: Spawn a new, independent process running request *r*, and pass its pid to *h*, which will determine the next request. The spawned process may make its own requests, including further spawns. When the spawned process makes a RDone request (meaning: no further requests), the process terminates.
- RSelf *h*: Pass the caller's pid to *h*, which will determine the next request.

The Scheduler itself defines two central types:

```
data Process = Proc Pid (Req Outcome) Int [Msg]
type ProcSys = Either Outcome ([Process], [Process])
```

轻闪PDF

程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

A *process* consists of a pid, a Req-computation of an Outcome, a local counter for naming spawned subprocesses, and a message queue. Depending on the exact contents of the request and message queue, the process is said to be in one of three states:

Busy A *busy* proce[illegible] can be immediately fulfilled, without requiring coordination [illegible] processes, so there is no reason to postpone it. This covers m[illegible]uctors, with the following two exceptions:

Ready A process is [illegible] is making a request that could be fulfilled, but doing so might [illegible]e subsequent behavior of the system, so an explicit choice must be [illegible] now, rather than waiting. The only such situation is a RSend request, because if two more more processes want to send a messages, there is a choice to be made about which one goes first.

Blocked A *blocked*, or *sleeping* process is one that has made a RRcv request *and* its message queue is empty, so that the process must wait until something is sent to it by another process.

A *process system* is either completed with a final outcome (the result from the initial process), or two lists. The former of these is the *ready queue*, and contains only ready processes. The scheduler will repeatedly pick a process from this queue and execute the send request. The simple round-robin scheduler always takes the first process on the queue, if there is more than one; the others may make different choices. If the ready queue should ever become empty, before the initial process has completed, the system is said to be *deadlocked*.

The second list in a non-completed system contains only the blocked processes. The order of this list is *not* significant, since processes will in general be unblocked from the middle of the list, as they receive messages.

There can only be one busy process at a time, the currently executing one. It will evolve until it either terminates, or becomes ready or blocked. This evolution may spawn other processes, which are themselves evolved until they enter one of the two non-busy states.

As processes evolve, they may add event entries to a (global) log maintained by the scheduler. Such entries may be requested by RLog, or they may be generated by the scheduler itself, and may be useful for debugging. The event log is just a list of String pairs, where the first string is a *tag* (in the simple version, just "system" or "user"), and the second is the associated text. The log is conveniently maintained in a standard Writer monad:

```
type Logger = Writer [(String, String)]
```

Most of the Scheduler's work is done by two functions:

```
runLocal :: Process -> Logger ProcSys
runReady :: Process -> ([Process], [Process]) -> Logger ProcSys
```

The former takes a (potentially busy) process and processes all its requests, until it can be placed in either of the two lists in a process system. If the process has no further requests (RDone), it just disappears, unless it was the initial process (Pid []), in which case the whole system goes into the completed state. Any processes spawned as part of running

轻闪PDF

runLocal are also evolved to become part of the system. If any such processes end up on the ready queue, it will be *after* the process that spawned them.

程序代写代做 CS编程辅导

The second function takes a *ready* process (i.e., about to send), together with the remaining process in the system, and returns the new state of the system. In general, after a send, both the new processes a[illegible] become busy, and both are individually evolved, starting with the se[illegible] are put on the *end* of the ready queue if they end up in a ready state a[illegible] blocked, their position on the blocked list doesn't matter.)

Note that there are a[illegible] situations that can arise in runReady, depending on the send destinat[illegible] end a message to itself, to another ready process (which should not affect its position in the queue), to a blocked process (which will unblock it), or to a no longer existing process (which means that the message is discarded). You may be able to handle some of these situations together, but think carefully about what should happen in each one.

WeChat: cstutorcs

Using these functions, we can specify the main scheduler function:

```
scheduler :: Monad m => (Int -> m Int) -> Req Outcome -> Int ->
  m (Outcome, [(String, String)]
```

Assignment Project Exam Help

The call scheduler *sel rq n* runs *rq* as the initial process, for at most *n* "big" scheduling steps (i.e., sends). If the initial process terminates within that limit, its outcome is returned. If the system has not completed after *n* steps, the outcome will be the exception (atom) timeout. If the ready-queue runs empty, the outcome will be the exception deadlock. Regardless of the outcome, the event log is included in the result.

Email: tutorcs@163.com

QQ: 749389476

The function *sel* is used to select which of the ready process to run in each step. It may be called with positive integer *k* (representing the length of the ready list), and it should return an integer in the range 0 through $(k-1)$. To get the round-robin behavior, we can take *m* to be just the *identity* monad, and make the the *sel* function can just always return 0, making the scheduler always select the first ready process from the queue.

https://tutorcs.com

For the backtracking scheduler, we take *m* to be the *list* monad, and make *sel* return a nondeterministic choice between all the valid selections. This means that every time there is a scheduling choice to be made, we consider all of the possible choices and collect all of the possible scenarios in a big list which can then be inspected. This guarantees that we do not miss any potential scenarios, but for complex process systems with lots of concurrent communications, the number of possible interleavings may explode, making a full backtracking search infeasible.

Therefore, another possibility is to make *sel* return a (pseudo-)*random* number in the range, and run the scheduler from the top a large number of times. But since Haskell is purely functional, that would just make the same psedo-random choice every time; instead, we must carry a random-number *seed* through the computation by taking *m* as a suitable *state* monad. Then for each run, we start with a different seed (could be just the number of the run), to hopefully get a reasonable exploration of the space.

All three strategies can be selected from the handed-out main program.

轻闪PDF

Getting started

程序代写代做 CS编程辅导

As usual, the recommended strategy is to develop a partial implementation of all three modules, possibly postponing some of the more difficult aspects, rather than aiming for completing each on[illegible]to the next. Within the modules, we recommend the following priorit[illegible]

In the Parser, omit fi[illegible]ression and patterns), general atoms, and the finer points of operator p[illegible]ociativity, whitespace, etc. (But do make a list of what you skip.)

In the Evaluator, foc[illegible]gnment environment and basic expression forms: atoms, variables, tuples, cases, and named functions. Wait with try/catch, function values, and send/receive/spawn support. In particular, postpone “stashing” in `receive`s; initially, make unmatched messages just throw an exception.

In the Scheduler, first set up logging infrastructure. Then do as many cases for `runLocal` as you can, before moving on to `runReady`. Implement just a round-robin scheduler first, which simply ignores the selection function and get it to work, before considering the generalization to arbitrary monads and selection functions.

WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

General instructions

All your code should be put into the provided skeleton files under `code/agner/`, as indicated. In particular, the actual code for each module *Mod* should go into `src/ModImpl.hs`. Do not modify `src/AST.hs` and `src/Runtime.hs`, which contains the common type definitions presented above; nor should you modify the type signatures of the exported functions in the APIs. Doing so will likely break our automated tests, which may affect your grade. Be sure that your codebase builds with the provided `app/Main.hs` with `stack build`; if any parts of your code contain syntax or type errors, be sure to comment them out for the submission.

In your report, you should describe your main design and implementation choices for each module in a separate (sub)section. Be sure to cover *at least* the specific points for each module asked about in the text above (and emphasized with *** in the margin), as well as anything else you consider significant. Low-level, technical details about the code should normally be given as comments in the source itself.

For the assessment, we recommend that you use the same (sub)headings as in the weekly Haskell assignments (Completeness, Correctness, Efficiency, Robustness, Maintainability, Other). Feel free to skip aspects about which you have nothing significant to say. You may assess each module separately, or make a joint assessment in which you assess each aspect for both modules.