Chalmers | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Tuesday, March 13th, 2018, Samhällsbyggnad, 8:30 (4hs)

(including example solutions to programming problems)

Alejandro Russo, tel. 0729744968

- The maximum amount of points you can score on the exam: 60 points. The grade for the exam is as follows:

  Chalmers: **3**: 24 - 35 points, **4**: 36 - 47 points, **5**: 48 - 60 points.
  GU: Godkänd 24-47 points, Väl godkänd 48-60 points
  PhD student: 36 points to pass.

- Results: within 21 days.

- **Permitted materials (Hjälpmedel):** Dictionary (Ordlista/ordbok).

  You may bring up to two pages (on one A4 sheet of paper) of pre-written notes — a "summary sheet". These notes may be typed or handwritten. They may be from any source. If this summary sheet is brought to the exam it must also be handed in with the exam (so make a copy if you want to keep it).

- **Notes:**
  - Read through the paper first and plan your time.
  - Answers preferably in English, some assistants might not read Swedish.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each of the questions on a new page.
  - The exact syntax of Haskell is not so important as long as the graders can understand the intended meaning. If you are unsure just put in an explanation of your notation.
  - Hand in the summary sheet (if you brought one) with the exam solutions.
  - As a recommendation, consider spending around 1h 20 minutes per exercise. However, this is only a recommendation.
  - To see your exam: *by appointment (send email to Alejandro Russo)*

Problem 1: (**eta-conversion**)

An eta-conversion, written η-conversion, is adding or dropping of abstraction over a function without changing the meaning of your program. For example, *id* and $\lambda x \rightarrow id\ x$ are equivalent terms because it is possible to *eta*-convert one into the other. The term "η-conversion" can refer to the process in either direction. Extensive use of η-reduction can lead to *point free* programming, i.e., where functions are elegantly expressed as, for instance, $f = f1 \circ f2 \circ f3$. Furthermore, η-conversion is also typically used in certain compile-time optimisations.

There is an interesting interaction between forcing evaluation in Haskell and *eta*-conversion. Recall the primitive $seq :: a \rightarrow b \rightarrow b$ which forces evaluation in Haskell. It takes two arguments, forces the evaluation of the first one and returns the second one. To illustrate how it works, you can appreciate below some invocations of it.

```
Prelude> undefined `seq` 42
*** Exception: Prelude.undefined
Prelude> (5+5) `seq` 42
42
Prelude> unsafePerformIO (putStrLn "I am being forced") `seq` 42
I am being forced
42
Prelude>
```

Your task is to define f such that f '*seq*' 42 and $(\lambda x \rightarrow f\ x)$ '*seq*' 42 behave differently. In fact, it is known that *eta*-conversion *holds up to strictness* in Haskell, i.e., up to forcing evaluation of terms. In other words, if you never use *seq*, then you have η-conversion in Haskell.

Solution:

$$
\begin{aligned}
&f\ \ = \bot\\
&f' = \lambda x \rightarrow f\ x\\
&test1 = f\ \text{`}seq\text{`}\ 42 \quad \text{-- it crashes}\\
&test2 = f'\ \text{`}seq\text{`}\ 42 \quad \text{-- it returns 42}
\end{aligned}
$$

(10p)

Problem 2: (**Monads**)

a) Consider the definition of *State s a*.

$$\textbf{newtype}\ State\ s\ a = State\ \{ runState :: s \rightarrow (a, s) \}$$

Define a monad instance *Monad* (*State s*) as well as two functions $put :: s \rightarrow State\ s\ ()$ and $get :: State\ s\ s$ such that the following five laws are satisfied:

$$
\begin{aligned}
&put\ s \gg put\ s' &&\equiv put\ s'\\
&put\ s \gg get &&\equiv put\ s \gg return\ s\\
&get \ggg put &&\equiv return\ ()\\
&get \ggg \lambda s \rightarrow get \ggg k\ s &&\equiv get \ggg \lambda s \rightarrow k\ s\ s\\
&return\ () &&\not\equiv \bot
\end{aligned}
$$

The symbol $\bot$ denotes `undefined` in Haskell. Recall the definition $ma \gg mb = ma \gg\!= \lambda\_ \rightarrow mb$.

Solution:

$$\begin{aligned}
&\textbf{instance } \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where} \\
&\quad \mathit{return}\ a\ = \mathit{State}\ \$\ \lambda s \rightarrow (a, s) \\
&\quad \mathit{ma} \gg\!= k = \mathit{State}\ \$\ \lambda s \rightarrow (\lambda \mathit{as} \rightarrow \mathit{runState}\ (k\ (\mathit{fst}\ \mathit{as}))\ (\mathit{snd}\ \mathit{as}))\ (\mathit{runState}\ \mathit{ma}\ s) \\
&\mathit{put} :: s \rightarrow \mathit{State}\ s\ () \\
&\mathit{put}\ s = \mathit{State}\ \$\ \lambda_ \rightarrow ((), s) \\
&\mathit{get} :: \mathit{State}\ s\ s \\
&\mathit{get} = \mathit{State}\ \$\ \lambda s \rightarrow (s, s)
\end{aligned}$$

(8p)

b) We assume that we have only *total monadic computations*, that is, computations which always terminate (e.g., there is no $\bot$ or infinite loops anywhere). Under that assumption, prove that your definitions satisfy the monadic laws as well as the laws described in **a)**.

You can assume the following (equivalent) properties:

(STATE ID 1)	(STATE ID 2)
$\mathit{State}\ (\mathit{runState}\ \mathit{ma}) \equiv \mathit{ma}$	$\mathit{runState}\ (\mathit{State}\ f) \equiv f$

Solution:

Conventions:

$$\begin{aligned}
&\mathit{fst}\ x \equiv x.1 \\
&\mathit{snd}\ x \equiv x.2 \\
&\mathit{runState}\ \mathit{ma}\ s \equiv [\![\mathit{ma}]\!]\ s
\end{aligned}$$

Left id:

$$\begin{aligned}
&\mathit{return}\ x \gg\!= f \\
&\quad \text{-- by definition of return and bind} \\
&\equiv \mathsf{State}\ (\lambda s \rightarrow (\lambda as \rightarrow [\![f\ \ as.1]\!]\ as.2)\ ([\![\mathsf{State}\ (\lambda s \rightarrow \langle x, s\rangle)]\!]\ s)) \\
&\quad \text{-- by state id 2} \\
&\equiv \mathsf{State}\ (\lambda s \rightarrow (\lambda as \rightarrow [\![f\ \ as.1]\!]\ as.2)\ ((\lambda s \rightarrow \langle x, s\rangle)\ s)) \\
&\quad \text{-- by } \lambda\text{-application} \\
&\equiv \mathsf{State}\ (\lambda s \rightarrow (\lambda as \rightarrow [\![f\ \ as.1]\!]\ as.2)\ \langle x, s\rangle) \\
&\quad \text{-- by definition of fst, snd and } \lambda\text{-application} \\
&\equiv \mathsf{State}\ (\lambda s \rightarrow [\![f\ \ x]\!]\ s) \\
&\quad \text{-- by } \eta\text{-conversion} \\
&\equiv \mathsf{State}\ ([\![f\ \ x]\!]) \\
&\quad \text{-- by state id 1} \\
&\equiv f\ x
\end{aligned}$$

Rigth id:

$$
\begin{aligned}
& f \gg\!= \mathit{return} \\
& \quad \text{-- by definition of return and bind} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as \to [\![\mathsf{State}\ (\lambda s' \to \langle as.1, s' \rangle)]\!]\ as.2)\ ([\![f]\!]\ s)) \\
& \quad \text{-- by state id 2} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as \to (\lambda s' \to \langle as.1, s' \rangle)\ as.2)\ ([\![f]\!]\ s)) \\
& \quad \text{-- by } \lambda\text{-application} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as \to \langle as.1, as.2 \rangle)\ ([\![f]\!]\ s)) \\
& \quad \text{-- by } \eta\text{-conversion and } \lambda\text{-application} \\
& \equiv \mathsf{State}\ (\lambda s \to [\![f]\!]\ s) \\
& \quad \text{-- by } \eta\text{-conversion and state id 1} \\
& \equiv f
\end{aligned}
$$

Assoc:

$$
\begin{aligned}
& (m \gg\!= f) \gg\!= g \\
& \quad \text{-- by definition of bind} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as \to [\![g\ \ as.1]\!]\ as.2)\ ([\![m \gg\!= f]\!]\ s)) \\
& \quad \text{-- by definition of bind} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as \to [\![g\ \ as.1]\!]\ as.2)\ ([\![\mathit{State}\ (\lambda s' \to (\lambda as' \to [\![f\ \ as'.1]\!]\ as'.2)\ ([\![m]\!]\ s'))]\!]\ s)) \\
& \quad \text{-- by state id 2 and } \lambda\text{-application} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as \to [\![g\ \ as.1]\!]\ as.2)\ ((\lambda as' \to [\![f\ \ as'.1]\!]\ as'.2)\ ([\![m]\!]\ s))) \\
& \quad \text{-- by } \eta\text{-conversion} \\
& \equiv \mathsf{State}\ (\lambda s \to ((\lambda as \to [\![g\ \ as.1]\!]\ as.2) \circ (\lambda as' \to [\![f\ \ as'.1]\!]\ as'.2))\ ([\![m]\!]\ s)) \\
& \quad \text{-- by } \eta\text{-conversion} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as' \to (\lambda as \to [\![g\ \ as.1]\!]\ as.2)\ ([\![f\ \ as'.1]\!]\ as'.2))\ ([\![m]\!]\ s)) \\
& \quad \text{-- by } \eta\text{-conversion} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as' \to (\lambda s' \to (\lambda as \to [\![g\ \ as.1]\!]\ as.2)\ ([\![f\ \ as'.1]\!]\ s'))\ as'.2)\ ([\![m]\!]\ s)) \\
& \quad \text{-- by state id 2} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as' \to [\![\mathsf{State}\ (\lambda s' \to (\lambda as \to [\![g\ \ as.1]\!]\ as.2)\ ([\![f\ \ as'.1]\!]\ s'))]\!]\ as'.2)\ ([\![m]\!]\ s)) \\
& \quad \text{-- by definition of bind} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as' \to [\![f\ \ as'.1 \gg\!= g]\!]\ as'.2)\ ([\![m]\!]\ s)) \\
& \quad \text{-- by } \eta\text{-conversion} \\
& \equiv \mathsf{State}\ (\lambda s \to (\lambda as' \to [\![(\lambda x \to f\ \ x \gg\!= g)\ \ as'.1]\!]\ as'.2)\ ([\![m]\!]\ s)) \\
& \quad \text{-- by definition of bind} \\
& \equiv m \gg\!= (\lambda x \to f\ \ x \gg\!= g)
\end{aligned}
$$

Put-put:

$$
\begin{aligned}
& \mathit{put}\ s \gg \mathit{put}\ s' \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![\mathit{put}\ s']\!]\ as.2)\ ([\![\mathit{put}\ s]\!]\ s_0) \\
& \quad \equiv \quad \{\text{-For all } s \text{ and } s'\text{, we have } [\![\mathit{put}\ s]\!]\ s' \equiv [\![\mathsf{State}\ \lambda\_ \to \langle \top, s \rangle]\!]\ s' \equiv \langle \top, s \rangle.\ \text{-}\} \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to \langle \top, s' \rangle)\ ([\![\mathit{put}\ s]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to \langle \top, s' \rangle \\
& \quad \equiv \\
& \mathit{put}\ s'
\end{aligned}
$$

Put-get:

$$
\begin{aligned}
& \mathit{put}\ s \gg \mathit{get} \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![\mathit{get}]\!]\ as.2)\ ([\![\mathit{put}\ s]\!]\ s_0) \\
& \quad \equiv \quad \{\text{-For all } s\text{, we have } [\![\mathit{get}]\!]\ s \equiv [\![\mathsf{State}\ \lambda s_0 \to \langle s_0, s_0 \rangle]\!]\ s \equiv \langle s, s \rangle.\ \text{-}\} \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to \langle as.2, as.2 \rangle)\ \langle \top, s \rangle \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to \langle s, s \rangle \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to \langle s, as.2 \rangle)\ \langle \top, s \rangle \\
& \quad \equiv \quad \{\text{-For all } a \text{ and } s\text{, we have } [\![\mathit{return}\ a]\!]\ s \equiv [\![\mathsf{State}\ \lambda s_0 \to \langle a, s_0 \rangle]\!]\ s \equiv \langle a, s \rangle.\ \text{-}\} \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![\mathit{return}\ s]\!]\ as.2)\ ([\![\mathit{put}\ s]\!]\ s_0) \\
& \quad \equiv \\
& \mathit{put}\ s \gg \mathit{return}\ s
\end{aligned}
$$

Get-put:

$$
\begin{aligned}
& \mathit{get} \gg\!= \mathit{put} \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![(\lambda s \to \mathit{put}\ s)\ as.1]\!]\ as.2)\ ([\![\mathit{get}]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![\mathit{put}\ as.1]\!]\ as.2)\ ([\![\mathit{get}]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to \langle \top, as.1 \rangle)\ \langle s_0, s_0 \rangle \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to \langle \top, s_0 \rangle \\
& \quad \equiv \\
& \mathit{return}\ \top
\end{aligned}
$$

Get-get:

$$
\begin{aligned}
& get \gg\!= \lambda s \to get \gg\!= k\ s \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as_0 \to [\![(\lambda s \to get \gg\!= k\ s)\ as_0.1]\!]\ as_0.2)\ ([\![get]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as_0 \to [\![get \gg\!= k\ as_0.1]\!]\ as_0.2)\ ([\![get]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as_0 \to [\![\mathsf{State}\ \lambda s_1 \to (\lambda as_1 \to [\![k\ as_0.1\ as_1.1]\!]\ as_1.2)\ ([\![get]\!]\ s_1)]\!]\ as_0.2)\ ([\![get]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as_0 \to (\lambda as_1 \to [\![k\ as_0.1\ as_1.1]\!]\ as_1.2)\ ([\![get]\!]\ as_0.2))\ ([\![get]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as_0 \to (\lambda as_1 \to [\![k\ as_0.1\ as_1.1]\!]\ as_1.2)\ \langle as_0.2, as_0.2\rangle)\ \langle s_0, s_0\rangle \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to [\![k\ s_0\ s_0]\!]\ s_0 \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![k\ as.1\ as.1]\!]\ as.2)\ \langle s_0, s_0\rangle \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![k\ as.1\ as.1]\!]\ as.2)\ ([\![get]\!]\ s_0) \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to (\lambda as \to [\![(\lambda s \to k\ s\ s)\ as.1]\!]\ as.2)\ ([\![get]\!]\ s_0) \\
& \quad \equiv \\
& get \gg\!= \lambda s \to k\ s\ s
\end{aligned}
$$

Return:

$$
\begin{aligned}
& return\ \top \\
& \quad \equiv \\
& \mathsf{State}\ \lambda s_0 \to \langle \top, s_0\rangle \\
& \quad \not\equiv \\
& \bot
\end{aligned}
$$

(5p)

c) Now, let us assume that you are in Haskell. There is one monadic law that does not hold in Haskell due to *eta*-conversion (recall Problem 1). Which one is it? Justify your answer.

Solution:

$$
\begin{aligned}
\bot \gg\!= return &\equiv State\ \$\ \lambda s \to (\lambda as \to runState\ (return\ (fst\ as))\ (snd\ as))\ (runState\ \bot\ s) \\
&\equiv State\ \$\ \lambda s \to \bot \\
&\not\equiv \bot
\end{aligned}
$$

(8p)

Problem 3: (**DSLs**)

Consider the following type of expressions with explicit application

$$\begin{aligned}\textbf{data}\ \mathit{Expr} &= \mathit{Lit}\ \mathit{Int}\\ &\mid \mathit{Plus}\\ &\mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr} \quad \text{-- the application of a function expression to an argument}\end{aligned}$$

In this language the expression 1 + 2 is modelled as $\mathit{App}\ (\mathit{App}\ \mathit{Plus}\ (\mathit{Lit}\ 1))\ (\mathit{Lit}\ 2)$. The following terms are valid elements of the *Expr* type but they do not correspond to well-formed expressions: $\mathit{App}\ (\mathit{Lit}\ 1)\ (\mathit{Lit}\ 2)$ and $\mathit{App}\ \mathit{Plus}\ \mathit{Plus}$.

a) Define a generalised datatype (GADT) $\mathit{Expr}\ t$ whose elements correspond only to well-formed expressions of type t. For instance,

$$\mathit{App}\ (\mathit{App}\ \mathit{Plus}\ (\mathit{Lit}\ 1))\ (\mathit{Lit}\ 2) :: \mathit{Expr}\ \mathit{Int}$$
$$\mathit{App}\ \mathit{Plus}\ (\mathit{Lit}\ 1) :: \mathit{Expr}\ (\mathit{Int} \to \mathit{Int})$$

Solution:

$$\begin{aligned}&\textbf{data}\ \mathit{Expr}\ t\ \textbf{where}\\ &\quad \mathit{Lit}\ \ :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\\ &\quad \mathit{Plus} :: \mathit{Expr}\ (\mathit{Int} \to \mathit{Int} \to \mathit{Int})\\ &\quad \mathit{App} :: \mathit{Expr}\ (a \to b) \to \mathit{Expr}\ a \to \mathit{Expr}\ b\end{aligned}$$

(5p)

b) Implement an evaluator $\mathit{eval} :: \mathit{Expr}\ t \to t$ for your expressions.

Solution:

$$\begin{aligned}&\mathit{eval} :: \mathit{Expr}\ t \to t\\ &\mathit{eval}\ (\mathit{Lit}\ n) = n\\ &\mathit{eval}\ \mathit{Plus} \quad\ = (+)\\ &\mathit{eval}\ (\mathit{App}\ e1\ e2) = (\mathit{eval}\ e1)\ (\mathit{eval}\ e2)\end{aligned}$$

(5p)

Problem 4: (**Singleton types**)

A singleton type is a type with exactly one value—note that `undefined` is not a value! Because of this, learning something about the value of a singleton type tells you about the type, and vice versa. For instance, we have the following definition of natural numbers as singleton types.

$$\begin{aligned}&\textbf{data}\ Z \quad\ = \mathit{Zero}\\ &\textbf{data}\ S\ n = \mathit{Succ}\ n\end{aligned}$$

a) Given a function $f :: S\ (S\ Z) \to Int$, what is the value (or values) that it receives as argument? Given the value $Succ\ (Succ\ (Succ\ Zero))$, what is its type? Justify your answers.

Solution:

The only possible value of type $S\ (S\ Z)$ is:

$$Succ\ (Succ\ Zero) :: S\ (S\ Z)$$

The type of $Succ\ (Succ\ (Succ\ Zero))$ is:

$$Succ\ (Succ\ (Succ\ Zero)) :: S\ (S\ (S\ Z))$$

(1p)

b) Sometimes, we need to take type-level natural numbers and cast them into simple integers. For that, you should envision the function *toInt* which gets applied to values of singleton types and returns an integer. Please, see below many invocations to that function.

```
Prelude> toInt Zero
0
Prelude> toInt (Succ (Succ Zero))
2
```

Your task is to implement such function. What is its type?

Solution:

$$
\begin{array}{l}
\textbf{class}\ ToInt\ a\ \textbf{where} \\
\quad toInt :: a \to Int \\
\textbf{instance}\ ToInt\ Z\ \textbf{where} \\
\quad toInt\ Zero = 0 \\
\textbf{instance}\ ToInt\ n \Rightarrow ToInt\ (S\ n)\ \textbf{where} \\
\quad toInt\ (Succ\ n) = 1 + toInt\ n
\end{array}
$$

The type of *toInt* is $ToInt\ a \Rightarrow a \to Int$.

(8p)

c) Now, we would like to write a function which converts integers into a value of a singleton type. Observe that we want a function that, when given an argument at runtime, it generates the right value of a singleton type. However, singleton types (as any other types) are compile-time information. How come can we transform some information at runtime into information at compile-time? Well, we cannot! What we can do instead is to verify (at runtime) that the argument of the function coincides with the value associated to the singleton type assumed at compile-time. Let us assume that the function *toSZ* is the one taking an integer as an argument and returning a value of a singleton type. Observe the following invocations.

```
Prelude> let p = do  putStr "Number? "
                     str <- getLine
                     return (toInt (toSZ (read str) :: (S (S Z))))
Prelude> p
Number? 4
*** Exception: Non-exhaustive patterns in function toSZ
Prelude> p
Number? 2
2
```

In the example, the argument of *toSZ* is runtime information, i.e., obtained when running the program. The argument is the result of converting the input string *str* into an integer via *read str*. In contrast, the type signature $S\ (S\ Z))$ is information given to the type-system, i.e., before running anything. As you can see in the example, the definition of p is well typed. Now, when calling p, if the input is different from 2, the program halts. This is because, at compile time, we assume that the argument of *toSZ* will be converted into the value of the singleton type $S\ (S\ Z)$ and for that, the user needs to enter the number 2 at runtime.

You task is to implement *toSZ*. What is its type?

Solution:

$$
\begin{aligned}
&\textbf{class}\ \mathit{ToSZ}\ a\ \textbf{where}\\
&\quad \mathit{toSZ} :: \mathit{Int} \to a\\
&\textbf{instance}\ \mathit{ToSZ}\ Z\ \textbf{where}\\
&\quad \mathit{toSZ}\ 0 = \mathit{Zero}\\
&\textbf{instance}\ \mathit{ToSZ}\ n \Rightarrow \mathit{ToSZ}\ (S\ n)\ \textbf{where}\\
&\quad \mathit{toSZ}\ n = \mathit{Succ}\ (\mathit{toSZ}\ (n-1))
\end{aligned}
$$

(10p)