Department of Information and Computing Sciences
Utrecht University

INFOAFP – Exam

Andres Löh

Wednesday, 15 April 2009, 09:00–12:00

Preliminaries

- The exam consists of 6 pages (including this page). Please verify that you got all the pages.
- A maximum of 100 points can be gained.
- For every task, the maximal number of points is stated. Note that the points are distributed unevenly over the tasks.
- One task is marked as (*bonus*) and allows up to 5 extra points.
- Try to give simple and concise answers! Please try to keep your code readable!
- When writing Haskell code, you can use library functions, but make sure that you state which libraries you use.

Good luck!

1

Contracts (48 points total, plus 5 bonus points)

Here is a GADT of contracts:

$$\begin{aligned}&\textbf{data } \mathit{Contract} :: * \to * \textbf{ where}\\ &\quad \mathit{Pred} :: (a \to \mathit{Bool}) \to \mathit{Contract}\ a\\ &\quad \mathit{Fun} :: \mathit{Contract}\ a \to \mathit{Contract}\ b \to \mathit{Contract}\ (a \to b)\end{aligned}$$

A contract can be a predicate for a value of arbitrary type. For functions, we offer contracts that contain a precondition on the arguments, and a postcondition on the results.

Contracts can be attached to values by means of *assert*. The idea is that *assert* will cause run-time failure if a contract is violated, and otherwise return the original result:

$$\begin{aligned}&\mathit{assert} :: \mathit{Contract}\ a \to a \to a\\ &\mathit{assert}\ (\mathit{Pred}\ p) \qquad x = \textbf{if } p\ x \textbf{ then } x \textbf{ else } \mathit{error}\ \texttt{"contract violation"}\\ &\mathit{assert}\ (\mathit{Fun}\ \mathit{pre}\ \mathit{post})\ f\ = \mathit{assert}\ \mathit{post} \circ f \circ \mathit{assert}\ \mathit{pre}\end{aligned}$$

For function contracts, we first check the precondition on the value, then apply the original function, and finally check the postcondition on the result.

For example, the following contract states that a number is positive:

$$\begin{aligned}&\mathit{pos} :: (\mathit{Num}\ a, \mathit{Ord}\ a) \Rightarrow \mathit{Contract}\ a\\ &\mathit{pos} = \mathit{Pred}\ (>0)\end{aligned}$$

We have

$$\begin{aligned}&\mathit{assert}\ \mathit{pos}\ 2 \equiv 2\\ &\mathit{assert}\ \mathit{pos}\ 0 \equiv \bot \qquad \text{(contract violation error)}\end{aligned}$$

1 (6 points)**.** Define a contract

$$\mathit{true} :: \mathit{Contract}\ a$$

such that for all values x, the equation $\mathit{assert}\ \mathit{true}\ x \equiv x$ holds. Prove this equation using equational reasoning. •

Often, we want the postcondition of a function to be able to refer to the actual argument that has been passed to the function. Therefore, let us change the type of *Fun*:

$$\mathit{Fun} :: \mathit{Contract}\ a \to (a \to \mathit{Contract}\ b) \to \mathit{Contract}\ (a \to b)$$

The postcondition now depends on the function argument.

2 (4 points)**.** Adapt the function *assert* to the new type of *Fun*. •

2

3 (4 points)**.** Define a combinator

$$(\twoheadrightarrow) :: \mathit{Contract}\ a \to \mathit{Contract}\ b \to \mathit{Contract}\ (a \to b)$$

that reexpresses the behaviour of the old *Fun* constructor in terms of the new and more general one. •

4 (6 points)**.** Define a contract suitable for the list index function $(!!)$, i.e., a contract of type

$$\mathit{Contract}\ ([a] \to \mathit{Int} \to a)$$

that checks if the integer is a valid index for the given list. •

5 (6 points)**.** Define a contract

$$\mathit{preserves} :: \mathit{Eq}\ b \Rightarrow (a \to b) \to \mathit{Contract}\ (a \to a)$$

where $\mathit{assert}\ (\mathit{preserves}\ p)\ f\ x$ fails if and only if the value of $p\ x$ is different from the value of $p\ (f\ x)$. Examples:

$$\begin{array}{ll} \mathit{assert}\ (\mathit{preserves}\ \mathit{length})\ \mathit{reverse}\ \texttt{"Hello"} & \equiv \texttt{"olleH"} \\ \mathit{assert}\ (\mathit{preserves}\ \mathit{length})\ (\mathit{take}\ 5)\ \texttt{"Hello"} & \equiv \texttt{"Hello"} \\ \mathit{assert}\ (\mathit{preserves}\ \mathit{length})\ (\mathit{take}\ 5)\ \texttt{"Hello world"} & \equiv \bot \end{array}$$

•

6 (6 points)**.** Consider

$$\begin{array}{l} \mathit{preservesPos}\ \ = \mathit{preserves}\ (>0) \\ \mathit{preservesPos}' = \mathit{pos} \twoheadrightarrow \mathit{pos} \end{array}$$

Is there a difference between *assert preservesPos* and *assert preservesPos*′? If yes, give an example where they show different behaviour. If not, try to prove their equality using equational reasoning. •

We can add another contract constructor:

$$\mathit{List} :: \mathit{Contract}\ a \to \mathit{Contract}\ [a]$$

The corresponding case of *assert* is as follows:

$$\mathit{assert}\ (\mathit{List}\ c)\ xs = \mathit{map}\ (\mathit{assert}\ c)\ xs$$

7 (8 points)**.** Consider

$$\begin{array}{l} \mathit{allPos}\ \ = \mathit{List}\ \mathit{pos} \\ \mathit{allPos}' = \mathit{Pred}\ (\mathit{all}\ (>0)) \end{array}$$

Describe the differences between *assert allPos* and *assert allPos*′, and more generally between using *List* versus using *Pred* to describe a predicate on lists. (Hint: Think carefully and consider different situations before giving your answer. What about using the *allPos* and *allPos*′ contracts as parts of other contracts? What about lists of functions? What about infinite lists? What about strict and non-strict functions working on lists?) [No more than 60 words.] •

8 (8 points)**.** Discuss the advantages and disadvantages of using contracts and using QuickCheck properties. What is similar, what are the differences? [No more than 60 words.] •

9 (5 *bonus points*)**.** Can contracts be translated into QuickCheck properties automatically? If yes, try to define a function that does this. If not, discuss the difficulties. [No more than 60 words.] •

Maps and folds (29 points total)

10 (8 points)**.** For all f, g and z of suitable type, the equation

$$\mathit{foldr}\ f\ z \circ \mathit{map}\ g \equiv \mathit{foldr}\ (f \circ g)\ z$$

holds. Prove this theorem using equational reasoning and induction on lists. •

11 (6 points)**.** Translate the following program into System F, i.e., make all type abstractions and type applications explicit, and annotate all value-level lambda abstractions with their types.

$$\begin{aligned} &\mathit{mm} :: (a \to b) \to [[a]] \to [b] \\ &\mathit{mm} = \lambda f\ \mathit{xss} \to \mathit{head}\ (\mathit{map}\ (\mathit{map}\ f)\ \mathit{xss}) \end{aligned}$$

(Hint: It is not necessary to translate *head* and *map*, but writing down their System F types with explicit quantification will help you to know where to put type arguments.) •

The following data type is known as a generalized rose tree:

$$\textbf{data}\ \mathit{GRose}\ f\ a = \mathit{GFork}\ a\ (f\ (\mathit{GRose}\ f\ a))$$

12 (3 points)**.** What is the kind of *GFork*? •

If we instantiate f to $[\,]$, we get a rose tree, a tree that in every node can have arbitrarily many subtrees. Leaves can be represented by choosing an empty list:

$$\begin{aligned} &\mathit{leaf} :: a \to \mathit{GRose}\ [\,]\ a \\ &\mathit{leaf}\ x = \mathit{GFork}\ x\ [\,] \end{aligned}$$

13 (6 points)**.** What if we instantiate f to *Identity* (where

$$\textbf{newtype}\ \mathit{Identity}\ a = \mathit{Identity}\ a$$

is the identity on the type level)? And what if we instantiate f to *Maybe*? What kind of trees do we get, and what kind of familiar data structures are they similar to? [No more than 40 words.] •

14 (6 points)**.** Define an instance of class *Functor* for *GRose*, assuming that f is a *Functor*, and defining a function *fmap* such that the passed function is applied to all the elements of type a. •

Simulating inheritance (23 points total)

Using open recursion and an explicit fixed-point operator similar to

$$fix\ f = f\ (fix\ f)$$

we can simulate some features commonly found in OO languages in Haskell. In many OO languages, objects can refer their own methods using the identifier *this*, and to methods from a base object using *super*.

We model this by abstracting from both *this* and *super*:

$$\begin{aligned}
&\textbf{type}\ Object\ a = a \to a \to a\\
&\textbf{data}\ X = X\ \{n :: Int, f :: Int \to Int\}\\
&x, y, z :: Object\ X\\
&x\ super\ this = X\ \{n = 0, f = \lambda i \to i + n\ this\}\\
&y\ super\ this = super\ \{n = 1\}\\
&z\ super\ this = super\ \{f = f\ super \circ f\ super\}
\end{aligned}$$

We can extend one "object" by another using *extendedBy*:

$$\begin{aligned}
&extendedBy :: Object\ a \to Object\ a \to Object\ a\\
&extendedBy\ o_1\ o_2\ super\ this = o_2\ (o_1\ super\ this)\ this
\end{aligned}$$

By extending an object o_1 with an object o_2, the object o_1 becomes the super object for o_2.

Once we have built an object from suitable components, we can close it to make it suitable for use using a variant of *fix*:

$$fixObject\ o = o\ (error\ \texttt{"super"})\ (fixObject\ o)$$

We close the object o by instantiating it with an error super object and with itself as *this*.

15 (3 points)**.** What is the (most general) type of *fixObject*? •

16 (8 points)**.** What are the values of the following expressions?

$$\begin{aligned}
&n\ (fixObject\ x)\\
&f\ \ (fixObject\ x)\ 5\\
&n\ (fixObject\ y)\\
&f\ \ (fixObject\ y)\ 5\\
&n\ (fixObject\ (x\ `extendedBy`\ y))\\
&f\ \ (fixObject\ (x\ `extendedBy`\ y))\ 5\\
&f\ \ (fixObject\ (x\ `extendedBy`\ y\ `extendedBy`\ z))\ 5\\
&f\ \ (fixObject\ (x\ `extendedBy`\ y\ `extendedBy`\ z\ `extendedBy`\ z))\ 5
\end{aligned}$$

•

17 (4 points)**.** Define an object

$$zero :: Object\ a$$

such that for all types t and objects $x :: Object\ \ t$, the equation $x\ `extendedBy`\ zero \equiv zero\ `extendedBy`\ x \equiv x$ hold. [No proof required, just the definition.] •

A more interesting use for these functional objects is for adding effects to functional programs in an aspect-oriented way.

In order to keep a function extensible, we write it as an object, and keep the result value monadic:

$$\begin{array}{l} fac :: Monad\ m \Rightarrow Object\ (Int \rightarrow m\ Int) \\ fac\ super\ this\ n = \\ \quad \textbf{case}\ n\ \textbf{of} \\ \quad\quad 0 \rightarrow return\ 1 \\ \quad\quad n \rightarrow liftM\ (n*)\ (this\ (n-1)) \end{array}$$

Note that recursive calls have been replaced by calls to *this*. We can now write a separate aspect that counts the number of recursive calls:

$$\begin{array}{l} calls :: MonadState\ Int\ m \Rightarrow Object\ (a \rightarrow m\ b) \\ calls\ super\ this\ n = \\ \quad \textbf{do} \\ \quad\quad modify\ (+1) \\ \quad\quad super\ n \end{array}$$

We can now run the factorial function in different ways:

$$\begin{array}{lll} runIdentity & (fixObject\ fac\ 5) & \equiv 120 \\ runState & (fixObject\ (fac\ `extendedBy`\ calls)\ 5)\ 0 & \equiv (120, 6) \end{array}$$

18 (8 points)**.** Write an aspect *trace* that makes use of a writer monad to record whenever a recursive call is entered and whenever it returns. Also give a type signature with the most general type. Use a list of type

$$\begin{array}{l} \textbf{data}\ Step\ a\ b = Enter\ a \\ \qquad\qquad\quad \mid Return\ b \\ \quad \textbf{deriving}\ Show \end{array}$$

to record the log. As an example, the call

$$runWriter\ (fixObject\ (fac\ `extendedBy`\ trace)\ 3)$$

yields

$$(6, [Enter\ 3, Enter\ 2, Enter\ 1, Enter\ 0, Return\ 1, Return\ 1, Return\ 2, Return\ 6])$$

•