Department of Information and Computing Sciences
Utrecht University

INFOAFP – Exam

Andres Löh

Wednesday, 15 April 2009, 09:00–12:00

Solutions

- Not all possible solutions are given.
- In many places, much less detail than I have provided in the example solution was actually required.
- Solutions may contain typos.

Contracts (48 points total, plus 5 bonus points)

Here is a GADT of contracts:

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

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:

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

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:

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

We have

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

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. •

Solution 1.

$$\mathit{true} = \mathit{Pred}\ (\mathit{const}\ \mathit{True})$$

The proof:

$$\begin{array}{ll}
 & \mathit{assert}\ \mathit{true}\ x \\
\equiv & \{\ \text{definition of } \mathit{true}\ \} \\
 & \mathit{assert}\ (\mathit{Pred}\ (\mathit{const}\ \mathit{True}))\ x \\
\equiv & \{\ \text{definition of } \mathit{assert}\ \}
\end{array}$$

$$
\begin{array}{ll}
 & \textbf{if}\ (\mathit{const}\ \mathit{True})\ x\ \textbf{then}\ x\ \textbf{else}\ \mathit{error}\ \texttt{"contract violation"} \\
\equiv & \quad \{\ \text{definition of } \mathit{const}\ \} \\
 & \textbf{if}\ \mathit{True}\ \textbf{then}\ x\ \textbf{else}\ \mathit{error}\ \texttt{"contract violation"} \\
\equiv & \quad \{\ \textbf{if}\ \mathit{True}\ \} \\
 & x
\end{array}
$$

○

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*. ●

Solution 2.

$$\mathit{assert}\ (\mathit{Fun}\ \mathit{pre}\ \mathit{post})\ f = \lambda x \to (\mathit{assert}\ (\mathit{post}\ x) \circ f \circ \mathit{assert}\ \mathit{pre})\ x$$

Haskell actually forces all clauses of a function definition to have the same number of arguments, so to be entirely correct we have to use a lambda abstraction. But putting the x to the left of $=$ counts as correct, too. ○

3 (4 points). Define a combinator

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

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

Solution 3.

$$(\rightarrowtriangle)\ \mathit{pre}\ \mathit{post} = \mathit{Fun}\ \mathit{pre}\ (\mathit{const}\ \mathit{post})$$

○

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. ●

Solution 4. We need a nested *Fun* application to get a function contract taking two arguments. The actual condition is on the integer, so both the precondition for the first argument and the postcondition are *true*.

$$
\begin{array}{l}
\mathit{lookupContract} :: \mathit{Contract}\ ([a] \to \mathit{Int} \to a) \\
\mathit{lookupContract} = \mathit{Fun}\ \mathit{true}\ (\lambda \mathit{xs} \to \\
\qquad\qquad \mathit{Fun}\ (\mathit{Pred}\ (\lambda n \to 0 \leqslant n \wedge n < \mathit{length}\ \mathit{xs}))\ (\lambda n \to \\
\qquad\qquad\quad \mathit{true}))
\end{array}
$$

○

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

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

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

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

●

Solution 5.

$$preserves\ f = Fun\ true\ (\lambda x \to Pred\ (\lambda r \to f\ x == f\ r))$$

○

6 (6 points)**.** Consider

$$\begin{aligned}
preservesPos\ &= preserves\ (>0)\\
preservesPos'\ &= pos \twoheadrightarrow pos
\end{aligned}$$

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. ●

Solution 6. Both contracts have the same type, but they behave differently. Here is an example:

$$example\ c = assert\ c\ id\ 0$$

With this definition, we get

$$\begin{aligned}
&example\ preservesPos &&\equiv 0\\
&example\ preservesPos' &&\equiv \bot
\end{aligned}$$

The reason is that *preservesPos'* requires the function argument to be positive, whereas *preservesPos* says that the result is positive if and only if the argument was positive. ○

We can add another contract constructor:

$$List :: Contract\ a \to Contract\ [a]$$

The corresponding case of *assert* is as follows:

$$assert\ (List\ c)\ xs = map\ (assert\ c)\ xs$$

7 (8 points)**.** Consider

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

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.] •

Solution 7. The differences are due to laziness:

$$\mathit{test1}\ c = \mathit{length}\ (\mathit{assert}\ c\ [-1])$$

Now:

$$\begin{aligned} \mathit{test1}\ \mathit{allPos} &\equiv 1 \\ \mathit{test1}\ \mathit{allPos}' &\equiv \bot \qquad \text{(contract violation)} \end{aligned}$$

Another situation:

$$\mathit{test2}\ c = \mathit{take}\ 10\ (\mathit{assert}\ c\ [1\,..])$$

Now:

$$\begin{aligned} \mathit{test2}\ \mathit{allPos} &\equiv [1,2,3,4,5,6,7,8,9,10] \\ \mathit{test2}\ \mathit{allPos}' &\equiv \bot \qquad \text{(nontermination)} \end{aligned}$$

The contract *allPos* applies *pos* to the list elements lazily and can therefore miss errors, whereas *allPos′* forces evaluation and can thereby change the strictness behaviour of the program.

Generally, *Pred* is more flexible than *List* because it can describe properties that are not uniform over the list elements. However, *List* can be combined with function contracts, whereas *Pred* cannot. ◦

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.] •

Solution 8. Some advantages of QuickCheck: automatic generation of test cases, can be used to formulate algebraic properties that relate several functions. Disadvantages: test data may not reflect actually used data, has to be run explicitly.

Some advantages of Contracts: actual program runs are checked, allows for design by contract (for instance, blame assignment possible), can easily be switched off. Disadvantages: only actualy program runs are checked, difficult to express interaction between several functions. ◦

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.] ●

Solution 9. In short: Translation for predicates is easy. They contain boolean properties that are immediately testable. Translation for first-order functions is also possible if the domain type is in the class *Arbitrary*. We can then map a function contract to a property that generates arbitrary candidate values, and rejects those that do not fulfill the precondition. Higher-order functions are not so easy to translate. ○

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. ●

Solution 10. Case $[\,]$:

$$\begin{array}{ll}
 & (\mathit{foldr}\ f\ z \circ \mathit{map}\ g)\ [\,] \\
\equiv & \{\ \text{definition of } (\circ)\ \} \\
 & \mathit{foldr}\ f\ z\ (\mathit{map}\ g\ [\,]) \\
\equiv & \{\ \text{definition of } \mathit{map}\ \} \\
 & \mathit{foldr}\ f\ z\ [\,] \\
\equiv & \{\ \text{definition of } \mathit{foldr}\ \} \\
 & [\,] \\
\equiv & \{\ \text{definition of } \mathit{foldr}\ \} \\
 & (\mathit{foldr}\ (f \circ g)\ z)\ [\,]
\end{array}$$

Case $x : xs$:

$$\begin{array}{ll}
 & (\mathit{foldr}\ f\ z \circ \mathit{map}\ g)\ (x : xs) \\
\equiv & \{\ \text{definition of } (\circ)\ \} \\
 & \mathit{foldr}\ f\ z\ (\mathit{map}\ g\ (x : xs)) \\
\equiv & \{\ \text{definition of } \mathit{map}\ \} \\
 & \mathit{foldr}\ f\ z\ (g\ x : \mathit{map}\ g\ xs) \\
\equiv & \{\ \text{definition of } \mathit{foldr}\ \} \\
 & f\ (g\ x)\ (\mathit{foldr}\ f\ z\ (\mathit{map}\ g\ xs)) \\
\equiv & \{\ \text{definition of } (\circ)\text{, twice}\ \} \\
 & (f \circ g)\ x\ ((\mathit{foldr}\ f\ z \circ \mathit{map}\ g)\ xs) \\
\equiv & \{\ \text{induction hypothesis}\ \} \\
 & (f \circ g)\ x\ ((\mathit{foldr}\ (f \circ g)\ z)\ xs) \\
\equiv & \{\ \text{definition of } \mathit{foldr}\ \} \\
 & (\mathit{foldr}\ (f \circ g)\ z)\ (x : xs)
\end{array}$$

○

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.

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

(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.) •

Solution 11. Type arguments are in brackets. Syntax wasn't important, though, as long as the abstractions and applications were properly indicated.

$$\begin{array}{l} \mathit{mm} = \lambda \langle a \rangle\ \langle b \rangle\ (f :: a \to b)\ (\mathit{xss} :: [[a]]) \to \\ \qquad \mathit{head}\ \langle [b] \rangle\ (\mathit{map}\ \langle [a] \rangle\ \langle [b] \rangle\ (\mathit{map}\ \langle a \rangle\ \langle b \rangle\ f)\ \mathit{xss}) \end{array}$$

○

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 *GRose*? •

Solution 12.

$$\mathit{GRose} :: (* \to *) \to * \to *$$

○

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:

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

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.] •

Solution 13. In the case of *Identity* we get trees where every node has exactly one child. This is similar to streams (infinite lists). In the case of *Maybe*, every node can have one child or no children. This is like a non-empty list. ○

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*. ●

Solution 14.

```
instance (Functor f) ⇒ Functor (GRose f) where
  fmap f (GFork x xs) = GFork (f x) (fmap (fmap f) xs)
```

○

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*:

```
type Object a = a → a → a
data X = X {n :: Int, f :: Int → Int}
x, y, z :: Object X
x super this = X {n = 0, f = λi → i + n this}
y super this = super {n = 1}
z super this = super {f = f super ∘ f super}
```

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

```
extendedBy :: Object a → Object a → Object a
extendedBy o₁ o₂ super this = o₂ (o₁ super this) this
```

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 "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*? ●

Solution 15. It really is

```
fixObject :: (a → b → b) → b
```

but

$$\mathit{fixObject} :: (a \to a \to a) \to a$$

or equivalently

$$\mathit{fixObject} :: \mathit{Object}\ a \to a$$

are morally correct. ○

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

$$
\begin{array}{l}
n\ (\mathit{fixObject}\ x) \\
f\ \ (\mathit{fixObject}\ x)\ 5 \\
n\ (\mathit{fixObject}\ y) \\
f\ \ (\mathit{fixObject}\ y)\ 5 \\
n\ (\mathit{fixObject}\ (x\ \text{`}\mathit{extendedBy}\text{`}\ y)) \\
f\ \ (\mathit{fixObject}\ (x\ \text{`}\mathit{extendedBy}\text{`}\ y))\ 5 \\
f\ \ (\mathit{fixObject}\ (x\ \text{`}\mathit{extendedBy}\text{`}\ y\ \text{`}\mathit{extendedBy}\text{`}\ z))\ 5 \\
f\ \ (\mathit{fixObject}\ (x\ \text{`}\mathit{extendedBy}\text{`}\ y\ \text{`}\mathit{extendedBy}\text{`}\ z\ \text{`}\mathit{extendedBy}\text{`}\ z))\ 5
\end{array}
$$

●

Solution 16. In order: 0, $\perp$ (but 1 is morally correct), $\perp$, 1, 6, 7, 9. ○

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

$$\mathit{zero} :: \mathit{Object}\ a$$

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

Solution 17.

$$\mathit{zero\ super\ this} = \mathit{super}$$

Here is the proof for completeness:

$$
\begin{array}{l}
\quad x\ \text{`}\mathit{extendedBy}\text{`}\ \mathit{zero} \\
\equiv \\
\quad (\lambda \mathit{super\ this} \to \mathit{zero}\ (x\ \mathit{super\ this})\ \mathit{this}) \\
\equiv \\
\quad (\lambda \mathit{super\ this} \to x\ \mathit{super\ this}) \\
\equiv \\
\quad x
\end{array}
$$

○

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:

```
fac :: Monad m ⇒ Object (Int → m Int)
fac super this n =
  case n of
    0 → return 1
    n → liftM (n∗) (this (n − 1))
```

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:

```
calls :: MonadState Int m ⇒ Object (a → m b)
calls super this n =
  do
    modify (+1)
    super n
```

We can now run the factorial function in different ways:

```
runIdentity (fixObject fac 5)                      ≡ 120
runState    (fixObject (fac `extendedBy` calls) 5) 0 ≡ (120, 6)
```

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

```
data Step a b = Enter a
              | Return b
  deriving Show
```

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])
```

•

Solution 18.

```
trace :: MonadWriter [Step a b] m ⇒ Object (a → m b)
trace super this a =
do
  $tell\ [Enter\ a]$
  $b \leftarrow super\ a$
  $tell\ [Return\ b]$
  $return\ b$
```

In fact, *trace* has an even more general type, but the type above was sufficient. ◦