

Decorated specifications for states and exceptions

Dominique Duval

with J.-G. Dumas, L. Fousse, J.-C. Reynaud

LJK, University of Grenoble

January 7, 2011

IFIP WG1.3 meeting, Aussois

Outline

Introduction

Effects as decorated specifications

States

Exceptions

Conclusion

Semantics of programming languages

- ▶ several paradigms (functional, imperative, object-oriented,...)
- ▶ several kinds of semantics (denotational, operational,...)

Semantics of functional languages

The Curry-Howard-Lambek correspondence

logic	programming	categories
<i>propositions</i>	<i>types</i>	<i>objects</i>
<i>proofs</i>	<i>terms</i>	<i>morphisms</i>
intuitionistic logic	simply typed lambda calculus	cartesian closed categories

Semantics of non-functional languages?

Semantics of computational effects

Computational **effects** = non-functional features

Ex. states, exceptions, input-output, ...

- ▶ effects as **monads**:
Moggi [1989,...], Haskell
- ▶ effects as **Lawvere theories**:
Plotkin & Power [2001,...]

Here:

- ▶ effects as **decorated specifications**:
Duval & Lair & Reynaud [2003,...]

Underlying the three approaches:

category theory

Outline

Introduction

Effects as decorated specifications

States

Exceptions

Conclusion

Beyond monads (1)

[Moggi 1991, section 1]

*The basic idea behind the categorical semantics below is that, in order to interpret a programming language in a category \mathbf{C} , we distinguish **the object A of values** (of type A) from **the object TA of computations** (of type A), and take as denotations of programs (of type A) the elements of TA . In particular, we identify the type A with the object of values (of type A) and obtain the object of computations (of type A) by applying an unary type-constructor T to A . We call T a **notion of computation**, since it abstracts away from the type of values computations may produce. There are many choices for TA corresponding to different notions of computations.*

Beyond monads (2)

[Moggi 1991, section 1]

*Since the denotation of programs of type B are supposed to be elements of TB , programs of type B with a parameter of type A ought to be interpreted by **morphisms with codomain TB** , but for their domain there are **two alternatives**, either A or TA , depending on whether parameters of type A are identified with values or computations of type A . **We choose** the first alternative, because it entails the second. Indeed computations of type A are the same as values of type TA .*

The examples proposed by Moggi include

- ▶ the **states** monad $TA = (A \times St)^{St}$
where St is the set of states
- ▶ the **exceptions** monad $TA = A + Exc$
where Exc is the set of exceptions

Beyond monads (3)

Yes a morphism $A \rightarrow B + Exc$ provides a denotation for a program $A \rightarrow B$ which may **throw** an exception by mapping $a \in A$ to $e \in Exc$

And a morphism $A + Exc \rightarrow B + Exc$ provides a denotation for a program $A \rightarrow B$ which may **catch** an exception by mapping $e \in Exc$ to $b \in B$

We keep, and even emphasize, Moggi's distinction between several kinds of programs:

For states and for exceptions we distinguish **3 kinds of programs** and **2 kinds of equations**

The **decorations** (keywords or colors) are used for denoting this distinction

The bank account example

```
Class BankAccount {...  
    int balance ( ) const ;  
    void deposit (int) ;  
...}
```

from this C++ syntax to a signature?

- ▶ **apparent** signature \mathbf{ACC}_{app}

balance : void \rightarrow int

deposit : int \rightarrow void

the intended interpretation is **not** a model

- ▶ **explicit** signature \mathbf{ACC}_{exp}

balance : acc_st \rightarrow int

deposit : int \times acc_st \rightarrow acc_st

the intended interpretation is a model,

but the object-oriented flavour is **lost**

Decorations

m for **modifiers**

a for **accessors** (const methods)

p for **pure** functions

- ▶ **decorated** signature **ACC**_{dec}
 balance^a : void → int
 deposit^m : int → void

the intended interpretation is a model
and the object-oriented flavour is preserved
but this is **not** a signature

It is called a **decorated signature**

Morphisms

forget
the decorations

explain
the decorations

\mathbf{ACC}_{dec}
$b^a : \text{void} \rightarrow \text{int}$
$d^m : \text{int} \rightarrow \text{void}$



\mathbf{ACC}_{app}
$b : \text{void} \rightarrow \text{int}$
$d : \text{int} \rightarrow \text{void}$

\mathbf{ACC}_{exp}
$b : \text{acc_st} \rightarrow \text{int}$
$d : \text{int} \times \text{acc_st} \rightarrow \text{acc_st}$

Outline

Introduction

Effects as decorated specifications

States

Exceptions

Conclusion

States as effects

In imperative programming the state of the memory may be observed (**lookup**) and modified (**update**)

However, the state never appears explicitly in the syntax: there no “type of states”

We define three specifications for dealing with states

Notations

Loc = the set of **locations**

1 = the unit type

The apparent specification

From the **syntax** we get the apparent specification \mathbf{ST}_{app}

- For each location $i \in Loc$:

type V_i for the values of i

operations lookup $l_i : 1 \rightarrow V_i$

 update $u_i : V_i \rightarrow 1$

equations $l_i \circ u_i \equiv \text{id}_{V_i}$

$l_j \circ u_i \equiv l_j \circ ()_{V_i}$ for all $j \neq i$

EFFECT: the intended semantics *is not* a model of \mathbf{ST}_{app} .

The explicit specification

Notation

St = the “type of **states**” (e.g., $St = \prod_{i \in Loc} V_i$)

From the **semantics** we get the explicit specification **ST**_{exp}

- For each location $i \in Loc$:

type V_i for the values of i

operations lookup $l_i : St \rightarrow V_i$

 update $u_i : V_i \times St \rightarrow St$

equations $l_i \circ u_i \equiv pr_i$

$l_j \circ u_i \equiv l_j \circ pr'_i$ for all $j \neq i$

EFFECT: the intended semantics *is* a model of **ST**_{exp}

but **ST**_{exp} does not fit with the syntax

because of the “type of states” St

The decorated specification

Decorations for functions:

m for **modifiers**

a for **accessors** (= inspectors)

p for **pure** functions

AND decorations for equations

\sim for **weak** equations (equality on values only)

\equiv for **strong** equations (equality on values and state)

With the decorations we get the decorated specification \mathbf{ST}_{dec}

- For each location $i \in Loc$:

type V_i for the values of i

operations lookup $l_i^a : 1 \rightarrow V_i$

 update $u_i^m : V_i \rightarrow 1$

equations $l_i^a \circ u_i^m \sim \text{id}_{V_i}^p$

$l_i^a \circ u_i^m \sim l_j^a \circ ()_{V_i}^p$ for all $j \neq i$

Morphisms

forget
the decorations

explain
the decorations

\mathbf{ST}_{dec}
$l_i^a : 1 \rightarrow V_i$
$u_i^m : V_i \rightarrow 1$
2 weak equations



\mathbf{ST}_{app}
$l_i : 1 \rightarrow V_i$
$u_i : V_i \rightarrow 1$
2 equations



\mathbf{ST}_{exp}
$l_i : St \rightarrow V_i$
$u_i : V_i \times St \rightarrow St$
2 equations

Relevance of decorations

CLAIM

The decorated specification \mathbf{ST}_{dec} is “the most relevant”:

- ▶ both the apparent and the explicit specification may be recovered from \mathbf{ST}_{dec}
- ▶ \mathbf{ST}_{dec} fits with the syntax (no type St)
- ▶ the intended semantics is a “decorated model” of \mathbf{ST}_{dec}
- ▶ “decorated proofs” may be performed from \mathbf{ST}_{dec} , e.g.

$$u_i^m \circ l_i^a \equiv id_1$$

NOTE

Decorated models and decorated proofs refer to a decorated logic defined in the categorical framework of diagrammatic logics [Duval & Lair & Reynaud 2003...]

Outline

Introduction

Effects as decorated specifications

States

Exceptions

Conclusion

Exceptions as dual of states?

Monads:

states	$T(X) = (X \times St)^{St}$
exceptions	$T(X) = X + Exc$

Lawvere theories:

states	$lookup : Val \rightarrow Loc$ $update : 1 \rightarrow Loc \times Val$ with 7 equations
exceptions	$raise_e : 0 \rightarrow 1$ for $e \in Exc$ with no equation

Exceptions as dual of states!

- ▶ States involve the functor $X \times St$
for some distinguished “type of states” St
- ▶ Exceptions involve the functor $X + Exc$
for some distinguished “type of exceptions” Exc

CLAIM

**The duality between $X \times St$ and $X + Exc$
extends as a duality between states and exceptions**

l_i	lookup	dual to	t_i	“throw”
u_i	update	dual to	c_i	“catch”

Exceptions as effects

An exception may be raised (**raise** or **throw**)
and handled (**handle** or **try/catch**)

The “type of exceptions” does not appear explicitly
in the type of programs

For dealing with exceptions:

- ▶ first **dualize** the specifications for states
→ two key functions for exceptions
- ▶ then **encapsulate** the key functions
→ the usual functions for exceptions

Notations for exceptions

0 = the empty type

$Etype$ = the set of **exceptional types**

P_i = the type of parameters for exceptions of type i

Exc = the “type of **exceptions**” (e.g., $Exc = \sum_{i \in Etype} P_i$)

Decorations for functions:

m for functions which may catch exceptions

a for functions which propagate exceptions

p for pure functions

AND decorations for equations

\sim for **weak** equations (equality on non-exceptional arguments)

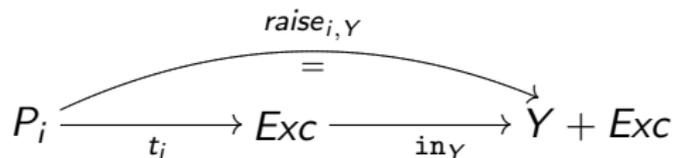
\equiv for **strong** equations (equality on all arguments)

Raising an exception, explicit

For raising an exception (of type i) into a type Y ,

- ▶ first the key operation t_i builds the exception
- ▶ then this exception is converted to type $Y + Exc$

$$raise_{i,Y} = \text{in}_Y \circ t_i$$

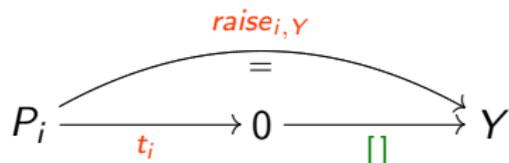


Raising an exception, decorated

For **raising** an exception (of type i) into a type Y ,

- ▶ first the key operation t_i^a builds the exception (of type i)
- ▶ then this exception is converted to type Y

$$\text{raise}_{i,Y}^a = []_Y^P \circ t_i^a$$



Handling an exception, explicit (1)

For handling an exception (of type i) raised by $f : X \rightarrow Y + Exc$ using $g : P_i \rightarrow Y + Exc$, the handling process builds

$$try\{f\} catch i \{g\} : X \rightarrow Y + Exc$$

using 2 nested conditionals

For each $x \in X$, $(try\{f\} catch i \{g\})(x) \in Y + Exc$ is

```
compute  $y = f(x) \in Y + Exc$ ;  
if  $y \in Y$   
  then return  $y \in Y \subseteq Y + Exc$   
  else //  $y$  is denoted  $e$   
    if  $e$  has type  $i$   
      then let  $a \in P_i$  be such that  $e = t_i(a)$   
        return  $g(a) \in Y + Exc$   
      else return  $e \in Exc \subseteq Y + Exc$ 
```

Handling an exception, explicit (2)

The key operation $c_i : Exc \rightarrow P_i + Exc$

- ▶ recognizes whether the given exception e has type i
- ▶ if so, returns a in P_i such that $e = t_i(a)$
- ▶ otherwise, returns $e \in Exc$

which means that

$$c_i \circ t_i \equiv \text{in}_i$$

$$c_i \circ t_j \equiv \text{in}'_i \circ t_j \text{ for all } j \neq i$$

DUAL to:

$$l_j \circ u_i \equiv \text{pr}_i$$

$$l_j \circ u_i \equiv l_j \circ \text{pr}'_i \text{ for all } j \neq i$$

Handling an exception, explicit (3)

The handling process builds $try\{f\} catch i \{g\} : X \rightarrow Y + Exc$ using

- ▶ the key operation c_i
- ▶ and 2 nested conditionals

For each $x \in X$, $(try\{f\} catch i \{g\})(x) \in Y + Exc$ is

```
compute  $y = f(x) \in Y + Exc$ ;  
if  $y \in Y$   
  then return  $y \in Y \subseteq Y + Exc$   
  else  
    compute  $z = c_i(y) \in P_i + Exc$ ;  
    if  $z \in P_i$   
      then return  $g(z) \in Y + Exc$   
      else return  $z \in Exc \subseteq Y + Exc$ 
```

Handling an exception, decorated (1)

For **handling** an exception (of type i) raised by $f^a : X \rightarrow Y$ using $g^a : P_i \rightarrow Y$, the handling process builds $(\text{try}\{f\} \text{catch } i \{g\})^a : X \rightarrow Y$ using

- ▶ the key operation c_i^m
- ▶ and 1 conditional

where the key operation $c_i^m : 0 \rightarrow P_i$ satisfies

$$c_i^m \circ t_i^a \sim \text{id}_{P_i}^P$$

$$c_i^m \circ t_j^a \sim []_{P_i}^P \circ t_j^a \text{ for all } j \neq i$$

DUAL to:

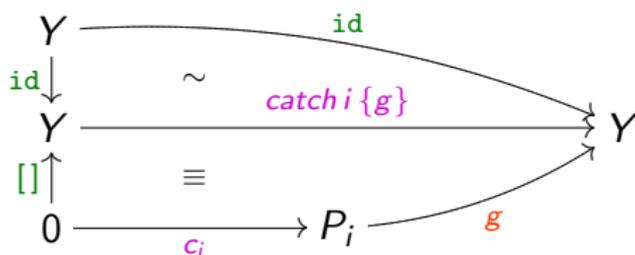
$$l_i^a \circ u_i^m \sim \text{id}_{V_i}^P$$

$$l_j^a \circ u_i^m \sim l_j^a \circ ()_{V_i}^P \text{ for all } j \neq i$$

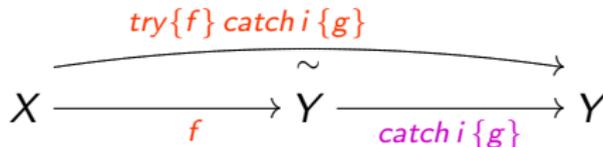
Handling an exception, decorated (2)

$(\text{try}\{f\}\text{ catch }i\{g\})^a$ using c_i^m and 1 conditional

Catching: $(\text{catch }i\{g\})^m$: catch with g^a an exception of type i



Handling: $(\text{try}\{f\}\text{ catch }i\{g\})^a$: compute f^a , then $(\text{catch }i\{g\})^m$, don't forget that exceptions "from outside" must be propagated!



Morphisms

forget
the decorations

EXC_{dec}
$t_j^a : P_j \rightarrow 0$
$c_j^m : 0 \rightarrow P_j$
2 weak equations

explain
the decorations



EXC_{app}
$t_j : P_j \rightarrow 0$
$c_j : 0 \rightarrow P_j$
2 equations



EXC_{exp}
$t_j : P_j \rightarrow Exc$
$c_j : Exc \rightarrow P_j + Exc$
2 equations

Outline

Introduction

Effects as decorated specifications

States

Exceptions

Conclusion

Effect = decorated specification =
apparent mismatch between syntax and semantics

- ▶ a new point of view on states
- ▶ a categorical formalization of exceptions with handling
- ▶ a duality between states and exceptions

Future work:

- ▶ other effects
- ▶ combining effects
- ▶ operational semantics

Some papers

- ▶ J.-G. Dumas, D. Duval, L. Fousse, J.-C. Reynaud.
States and exceptions are dual effects.
Workshop on Categorical Logic, Brno, 2010.
- ▶ J.-G. Dumas, D. Duval, J.-C. Reynaud.
Cartesian effect categories are Freyd-categories.
JSC (2010).
- ▶ C. Dominguez, D. Duval.
Diagrammatic logic applied to a parameterization process.
MSCS 20(04) p. 639-654 (2010).
- ▶ D. Duval, J.-C. Reynaud.
Dynamic logic and exceptions: an introduction.
Mathematics, Algorithms, Proofs. Dagstuhl Seminar 05021 (2005).
- ▶ D. Duval.
Diagrammatic Specifications.
MSCS (13) 857-890 (2003).