defining semantics for complex systems part 3: iTask semantics

Pieter Koopman, Rinus Plasmeijer

Radboud University Nijmegen
The Netherlands

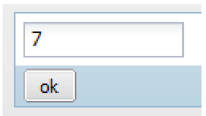
iTasks are wonderful

- workflow management systems supports and guides (administrative) tasks of humans and computers
  - entering data, approving transactions, ..
- the iTasks system is a combinator library to
  - specify workflows
  - execute tasks using a multi user web-interface
- distinguishing features of the iTask system
  - data dependent tasks
  - dynamic task creation and adaptation
- iTasks are in the Clean distribution
  - require generics, dynamics, iData and a little uniqueness

2

primitive tasks

- return e.g. return 1
  - returns the given value



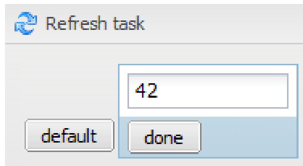
- editTask e.g. editTask "ok" 7
  - allows to edit a value until you press the button
- buttonTask e.g. buttonTask "ok" (return 5)

  - actually for every task t:
    buttonTask s t = editTask s Void >>= _ -> t

3

compose task: choose a task

```
task1 :: Task Int
task1 =  buttonTask "default" (return 1)
         -||- editTask "done" 42
```



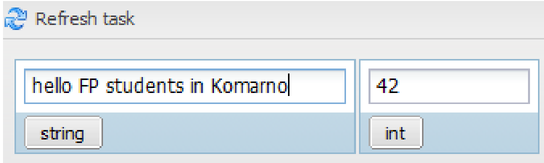
- iTask-systems generates a web-interface for task
- -||- is or-operator for tasks
- task is finished if either
  - the user presses the default button, or
  - presses the done button in the editor

(-||-) infixr 3 :: (Task a) (Task a) -> Task a

4

compose task: tasks in parallel

```
task2 :: Task (String, Int)
task2 = editTask "string" "u" -&&- editTask "int" 2
```



- -&&- is the and-operator
- task2 is finished if both
  - the user presses the string button, and
  - presses the int button

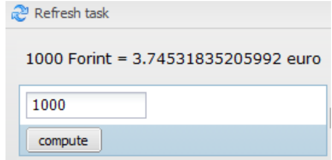
(-&&-) infixr 4 :: (Task a) (Task b) -> Task (a, b)

5

compose task: sequence of tasks

```
task3 :: Real -> Task Real
task3 x
 = [ Text (toString x +
     " Forint = " +
     toString (x * exchange_rate) + " euro"), BrTag [] ]
   ?>> editTask "compute" x
   >>= task3
```

a recursive task



- >>= is the monadic bind-operator
  - first do the task on the left, if that is finished
  - function on the right *generates* the next task

(>>=) infixl 1 :: (Task a) (a -> Task b) -> Task b

data dependent

6

bind enables data dependency

- the bind operator, >>=, gives the power to create data dependent tasks

```
task4 :: Task [Int]
task4
 =[Text "number of bids?"]
   ?>> editTask "go" 2
   >>= \n. if ( n<1 || n>10 )
             ([Text "between 1 and 10!"] ?>> task4)
             (andTasks  [ ("bid",editTask "ok" 1)
                        \\ i<-[1..n]
                        ])
```

7

some semantic questions about iTasks

```
task1 =  buttonTask "default" (return 1)
         -||- editTask "done" 42
```

- will the user be able to use the default button as soon as she starts editing the value 42?
- is the expression s -||- t equivalent to t -||- s?
- is the expression s -&&- t equivalent to t -&&- s?
- what is the value of return 0 -||- return 42 ?

8

look at the definition of combinators

```
allTasksCond    :: !String !(TasksToShow a) !(FinishPred a) ![LabeledTask a] -> Task [a] | iData a
allTasksCond label chooser pred taskCollection
= mkTask "andTasksCond" (doandTasks chooser taskCollection)
where
  lengthlta
  doandTa
  doandTa
 # ((alist,a

  | finishec
  | pred al
  # selectI
  # ((selec
  # (_,tst=

  = (alist,{                                                         ])

  where
     show
     show
     # (a,                                                            ml=BT []}
     # (a:
     = ([a

     chec
     | cta
     # (taskname,task)    = taskCollection!!ctasknr
     # (a,tst=:{activated = adone,html=html}) = ..
     ..
```

```
mkTask :: !String !(Task a) -> (Task a) | iCreateAndPrint a
mkTask taskname mytask = mkTaskNoInc taskname mytask o incTask

mkTaskNoInc :: !String !(Task a) -> (Task a) | iCreateAndPri
mkTaskNoInc taskname mytask = mkTaskNoInc`
where
     mkTaskNoInc` tst=:{activated,tasknr,use
     | not activated                    = (c          ,tst)
     # (val,tst=:{activated,trace})
     # tst      = {tst & tasknr =          ons = options, userId = userId}
     | isNothing trace || tas                    = (val,tst)
     = (val,{tst & trac           ertTrace activated tasknr userId options taskname
                                  (printToString val%(0,30)) (fromJust trace))})
```

this is not the way to answer semantic questions

and many more (generic) functions

why are the combinators so complicated?

- the definition of combinators does not give a quick answer to questions about the semantics
- combinators have several *interdependent* duties
  - produce html-code for the web interface displaying the current state of the task for each worker
  - handle inputs from the web, update task expression
  - interface to files and databases
  - multi-user
  - client/server
  - ..
- due to the large number of related jobs it is inevitable that the combinators are complicated

10

a solution

- in order to answer the semantic questions we define an operational semantics for iTasks
- in addition we want:
  - understand the system we are developing
    - guide decisions in the design and development
  - explain the behaviour to other people
  - reason about iTasks

  - use it as specification in model-based testing of the implementation of the iTask system
    - this connects the semantics and the implementation

11

how to define such a semantics ?

- it is kind of standard to define such a semantics using Scott brackets or horizontal bars
- however, such a specification is
  - hard to get correct and consistent
  - hard to validate
  - not suited for model-based testing of the system
- we use a functional programming language as carrier for the specification
  - language implementation checks types
  - simulation to validate the semantics
  - model-based test of the semantics
  - suited for model-based testing of the real system

12

a semantics for iTasks

- in order to reason about iTasks we define a simplified model of iTasks
  - based on a data type instead of functions
    - sequence operator contains function to guarantee enough expressive power (monadic bind)
  - some simplifications:
    - only basic iTask combinators
    - tasks can only handle a fixed number of types
  - concentrate first on event handling
    - ignore interface generation (HTML)
    - use a model of the world rather than real i/o

the data type for iTasks

- the type ITask mimics Task a:

```
:: ITask
   = EditTask        ID String BVal
   | .||.  infixr 3  ITask  ITask
   | .&&.  infixr 4  ITask  ITask
   | >>-   infixl 1  ID ITask (Val → ITask)
   | Return          Val
:: Val     = Pair Val Val   | BVal BVal
:: BVal    = String String | Int Int | VOID

ButtonTask id name task
 :== EditTask id name VOID >>- λ x → task
```

edit only basic values

real function for sequences

a monad

advantages of a data structure for tasks

- in the semantics we can look at the structure of the task
- we can inspect the task to see what events are enabled
- we can even change the task if desired

there is one exception

- in the bind operator we use a real function to maintain the expressive power

```
:: ITask =  >>- infixl 1   ID ITask ( Val → ITask )
          | ..
```


one-to-one mapping from Task a to ITask

- we had the real iTask task

```
task1 :: Task Int
task1 =  buttonTask "default" (return 1)
         -||- editTask "done" 42
```

- this is represented as

```
task1` :: ITask
task1` = ButtonTask "default" (Return (BVal (Int 1)))
         .||. EditTask "done" (Int 42)
```

events

- inputs for the task are called ***events***
- the iTask system and representation in its semantics both require events
  - we need to link events and tasks: label the task and the event with a unique id
  - system should assign the id's
- each event contains
  - ID: unique identification of subtask
  - the event kind: edit event with new value, or button event

```
:: ID        = ID [Num]
:: Event     = Event ID EventKind | ReFresh
:: EventKind = EE BVal | BE
```


needed events

- needed events: without these events the task cannot return a value
  - pressing the done button in an editor is needed
  - changing the value is not needed

```
colNeeded (EditTask id n v) = [Event id BE]
colNeeded (t1 .&&. t2) = colNeeded t1 ++ colNeeded t2
colNeeded (Bind id t f) = colNeeded t
colNeeded t             = [ ]
```

- in the semantic representation of task we can scan the data structure to find needed events
  - the function after a bind can generate needed events
  - the needed events of the generated task often depend on the result of the first task

the semantics

- semantics is specified by a function computing the new task given the current task and the event

```
(@.) infixl 9 :: ITask Event -> ITask
(@.) ( EditTask i n v ) ( Event j (EE w) )      edit event with new value
   | i == j = EditTask (next i) n w
(@.) ( EditTask i n e ) ( Event j BE )          button event
   | i==j = Return (BVal e)
(@.) ( t .||. u ) e
      = case t @. e of                          or combinator: first try left task
          t=:(Return _) = t
          t = case u @. e of
                u=:(Return _)  = u
                u              = t .||. u
..
```


iTask semantics part 2

```
..
(@.) ( t .&&. u ) e                             and combinator
     = case ( t @. e, u @. e ) of               send event to both tasks
         (Return v, Return w)= Return (Pair v w)
         (t, u)              = t .&&. u
(@.) ( Bind id t f ) e                          bind combinator
     = case t @. e of
         Return v = normalize id (f v)
         t        = Bind id t f                 assign unique identifiers
(@.) t e = t
```

this answers our questions

- is the user able to press the button default after changing a value in the editor?

u = ButtonTask id1 "default" (Return (BVal (Int 1)))
    .||. EditTask id2 "done" (Int 42)

- equational reasoning

u @. Event id2 (EI 36)
→ ButtonTask id1 "default" (Return (BVal (Int 1)))
    .||. EditTask id2 "done" (Int 36)

- answer: user can still press the default button

equivalence of tasks

- there are various notions of equivalence for tasks

1. two tasks are equivalent if they yield the same result for any input sequence
   - abstract from event Ids, they are invisible for users
   - we can decide to look only at needed events
2. have exactly the same structure
3. ..

- we use the first notion of equivalence
- task of different shape can be equivalent
  EditTask i s 3 ≈ EditTask i s 3 >>= λ v→Return (id v)

equivalence relation

- simulation: s ⊆ t:
  user can do anything with task t that can be done with s
  - ButtonTask id "ok" (Return 3) ⊆ EditTask id "ok" 3
- checking the shape of the tasks is not good enough
  - in general we have to check if it responds identical to all input sequences
- equivalence: s ⊆ t ∧ t ⊆ s ⇔ s ≈ t
- we can approximate this property automatically
  - using shape information can help to determine equivalence, but is in general not enough

equivalence of tasks

- given the tasks

s = editTask "a" 1
t = editTask "b" 2

- equivalence?

s -||- t ≈ t -||- s

- when are tasks equivalent x ≈ y ?
  - user can do the same with x and y
  - produce the same result for any sequence of events
  - ignore differences in layout and event id's
- simulation x ⊆ y :
  - user can do everything with y that can be done with x
- x ≈ y ⇔ x ⊆ y ∧ y ⊆ x

equivalence of tasks 2

- given the tasks

s = editTask "a" 1

t = editTask "b" 5

- equivalence s -||- t ≈ t -||- s ?
  - yes
- simulation s -||- t ⊆ t -||- s,
  s -||- t ⊆ t -||- s ?
  - yes

CEFP 2009: iTask semantics

equivalence of tasks 3

- given the tasks

s = editTask "a" 1

t = editTask "b" 5

- equivalence s ≈ t ?
  - no, BE
- simulation s ⊆ t,
  t ⊆ s ?
  - no, idem

CEFP 2009: iTask semantics

equivalence of tasks 4

- given the tasks

s = editTask "a" 1

t = buttonTask "b" (return 1)

- equivalence s ≈ t ?
  - no: EE 7, BE
- simulation s ⊆ t ?
  - no: EE 7, BE
- simulation t ⊆ s ?
  - yes: t only allows BE
- s ≈ t ⇔ s ⊆ t ∧ t ⊆ s ?

CEFP 2009: iTask semantics

equivalence of tasks 5

- given the tasks

t = editTask "a" 1

- equivalence t ≈ t -||- t ?
  - no: EE 7, BE
- simulation t ⊆ t -||- t ?
  - yes: use only 1 of the tasks in t -||- t
- simulation t -||- t ⊆ t ?
  - no: EE 7, BE
- be believed for a long time that t ≈ t -||- t !
  - testing the semantics showed that we were wrong

equivalence of tasks 6

- given the tasks

s = editTask "a" 1

t = editTask "b" 1 >>= return

- equivalence s ≈ t ?
  - yes, but shape of tasks is different
- simulation s ⊆ t, t ⊆ s ?
  - yes

equivalence of tasks 7

- given the tasks

s = editTask "a" 1

t = editTask "b" "Hi"

- equivalence s -&&- t ≈ t -&&- s ?
  - no the types are different
    Task (Int, String) and Task(String, Int)
- simulation s -&&- t ⊆ t -&&- s, s -&&- t ⊆ t -&&- s ?
  - no, idem

equivalence of tasks 8

- given the tasks

s = editTask "a" 1 >>= f

t = editTask "b" 1 >>= g

- equivalence s ≈ t ?
  - depends on the functions f and g
  - in general this is not decidable
  - we can approximate it by supplying sequences of events (like testing)
- simulation s ⊆ t, t ⊆ s ?
  - idem

approximation of equivalence

- if tasks are both of the form return v compare the values
- compare the needed events and the enabled events of tasks s and t, these should be equal
- apply different sequences of events and try again
- possible results:
  Proof equal for all possible sequences of events,
  Pass equal the used sequences of events,
  CounterExample we found a difference,
  Undefined no results found, e.g. infinite tasks

semantic properties

- what we want:
  - ∀ s t .(s .||. t) ≈ (t .||. s)
  - ∀ s t .(s .||. t) .||. u ≈ s .||. (t .||. u)
- some consequences:
  - task remain equivalent after applying a needed event
    s ≈ t ⇒ ∀ i ∈ neededEvents s . s @. i ≈ t @. i
  - ∀ i ∈ neededEvents u . u @. i ≈ (t .||. s) @. i
    where u = s .||. t
  - we can apply needed events in any order
    is = neededEvents t . t @. i ≈ t @. permutation is

checking the system

- having defined reduction and equivalence there are interesting questions
  - do the required properties hold
  - is the system consistent
  - ..
- checking this manually is tedious and error prone
- a proof system requires a significant amount of human guidance
- use our test system G∀st !!
  - express the properties using the logical combinators
  - generation of test data is done by the generic system, a little guidance by an additional data type is necessary

testing a property

- what we want:
  - ∀ s t .(s .||. t) ≈ (t .||. s)
- in Gast:

∀ s t .

pOr1 :: ITask ITask -> Property
pOr1 s t = (s .||. t) ~~ (t .||. s)

- the test result

"pOr1" Counterexample 1 found after 23 tests:
(Return (BVal (Int 0)))
(Return (Pair (BVal (Int 0)) (BVal (Int 0))))

- a correct property
  - ∀ s t . ¬ NF s ∧ ¬ NF t ⇒ (s .||. t) ≈ (t .||. s)

pOr s t = notNF [t, u] ==> (s .||. t) ~~ (t .||. s)

examples of automatic testing

- first test our property of the or-operator:

pOr :: ITask ITask -> Property
pOr s t = notNF [t, u] ==> (s .||. t) ~~ (t .||. s)

∀ s t .

Start = test pOr

- test execution yields: Passed after 1000 tests

pAnd s t = (s .&&. t) ~~ (t .&&. s)

- Counterexample: (GButtonT "b" (GReturn (BVal (Int 0))))
  (GReturn (BVal (String "a")))
- the types of the subtasks are different !

automatic generation of tasks for tests

- systematic automatic generation of test cases is desired
  - easy to do more tests
  - we do not forget tasks
- the data type ITask also allows bad test tasks
  - nonterminating
  - badly typed
- use an additional data type for desired test tasks
  - generate instance by the generic algorithm
  - transform these tasks to proper test talks
  - very simular as we did for terminating While-programs

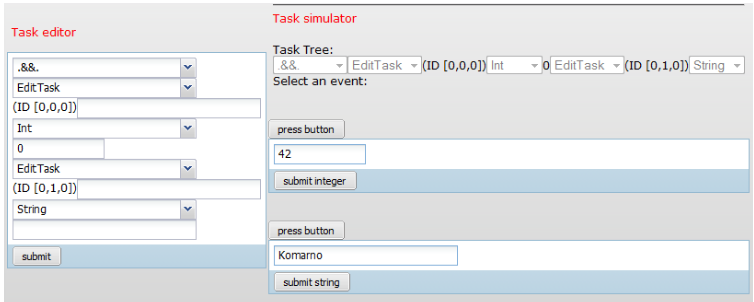
how to obtain properties

- 'obvious' properties arise during the development of the semantics
  - associativety of -||-, ..
- learn from your semantical mistakes
  - turn each example of undesired behaviour into a test
  - try to generalize these mistakes to general properties
  - (s .||. t) ≈ (t .||. s) only holds iff s and t are can consume events
- learn from your test results
  - turn each counterexample into a test (is it still correct)
  - try to generalize these errors to general properties

validating the semantics

- semantics with properties can be consistent, but wrong
  - testing does not reveal all problems
- validation by simulating and human inspection
  - since our tasks are data structures we can edit them with a standard iTask for this data type !
  - we can simulate the task using the executable semantics

a screen shot of the simulator



a methodology for defining consistent semantics

- semantics is a formal artefact (like a program)
  - we need tool support to get it correct
- use a functional programming language as carrier of the semantics
  - Haskell, Clean, Erlang, F#, ..
  - compiler spots mistakes with types and identifiers
- simulate the semantics to validate it using iTasks
- use model-based testing for regression tests: Gast
  - learn from your mistakes
- [optional] prove the tested properties
  - proving is much more work than model-based testing

conclusions

- the iTask system needs a semantics
  - we need to know what we are building
  - explain the system
  - model-based test of the implementation
- we defined an operation semantics and equivalence
- a functional programming language is very suited to construct such a semantics
  - concise
  - compiler checks types
  - automatic testing of desired properties of the semantics
    - we have results in seconds
  - validation by simulation
  - edit tasks in the simulator

future work

- extend the semantics to cover:
  - multi-user tasks
  - access to databases and files
  - exceptions
  - dynamic changes of the running tasks
  - ..
- test if the real iTask system behaves as specified
- prove properties that are tested successfully

the end

- Thanks for your attention
- questions ??

itasks
dynamic workflow system