Using a Functional Language as Embedding Modeling Language for Web-Based Workflow Applications

Rinus Plasmeijer[1], Jan Martin Jansen[2], Pieter Koopman[1], Peter Achten[1]

[1] Institute for Computing and Information Sciences (ICIS), Radboud University Nijmegen, the Netherlands,
[2] Faculty of Military Sciences, Netherlands Defence Academy, Den Helder, the Netherlands
jm.jansen.04@nlda.nl, {rinus, pieter, peter88}@cs.ru.nl

Abstract. Workflow management systems guide and monitor tasks performed by humans and computers. Workflow specifications are usually expressed in special purpose (graphical) formalisms. These models are concise and made rapidly. These formalisms commonly have as disadvantage that they have limited expressive power, handle only rather static workflows, do not handle intricate data dependencies, and cannot easily adapt to the current situation. Furthermore, workflow specification tools mostly generate a software infrastructure that needs to be extended with custom crafted code. To overcome these problems, we *entirely embed* a workflow modeling language in a modern general purpose *functional* language, and *generate* a complete workflow application. We have developed the iTask prototype system in the pure and lazy language Clean. An iTask workflow specification can use the expressive power and strong type facilities of Clean. Workflows can be higher-order and adapt their behavior based on the available data. The generated application is web-based and runs on both server and clients.

1 Introduction

Workflow Management Systems (WFMS) are computer applications that coordinate, generate, and monitor tasks performed by human workers and computers. Workflow *models* play a dominant role in WFMSs: the work that needs to be done to achieve a certain goal is specified in such a model as a structured and ordered collection of tasks that are assigned to available resources at run-time. In most WFMSs, a workflow model is used as input to generate a framework, i.e. a *partial* workflow *application.* Substantial coding is required to complete the workflow application. For example, most workflow models only deal with the flow of control of the application. All code with respect to manipulating the data in the workflow application has to be implemented separately. In this paper we advocate that a workflow model actually *can be* a computer program: an entire workflow application can and should be generated from a workflow model.

Contemporary WFMSs use special purpose (mostly graphical) modeling languages. Graphical formalisms are easier to communicate to domain experts than textual ones. Domain experts are knowledgeable about the work to be modeled, but often lack programming experience or formal training. Special purpose modeling languages provide workflow engineers with a concise formalism that enables the rapid development of a workflow framework. Unfortunately, these formalisms suffer from a number of disadvantages when compared with textual ones. First, *recursive definitions* are commonly inexpressible, and there are only limited ways to make *abstractions*. Second, workflow models usually only describe the *flow of control*. Data involved in the workflow is mostly maintained in databases and is extracted or inserted when needed (see the 'Data Interaction – Task to Task' workflow data pattern (8) by Russell *et al*[17]). As a consequence, workflow models cannot easily use this data to parameterize the flow of work. The workflow is more or less pre-described and cannot be dynamically adapted. Third, these dedicated languages usually offer a fixed set of *workflow patterns* [1]. However, in the real world work can be arranged in many ways. If it does not fit in a (combination of) pattern(s), then the workflow modeling language probably cannot cope with it either. Fourth, and related, is the fact that special purpose languages cannot express functionality that is not directly related to the main purpose of the language. To overcome this limitation, one either extends the special language or interfaces with code written in other formalisms. In both cases one is better off with a well designed general purpose language.

For the above reasons, we advocate to use a textual *programming language* as a workflow modeling language. This allows us to address all computational concerns in a workflow model and provides us with general recursion. We use a *functional* language, because they offer a lot of expressive power in terms of modeling domains, use of powerful types, and functional abstraction. We use the *pure* and *lazy* functional programming language Clean, which is a state-of-art language that offers fast compiler technology and *generic programming features* [2] which are paramount for generating systems from models in a type-safe way. Clean is freely available at `http://clean.cs.ru.nl/`.

To verify our claim, we have developed a prototype workflow modeling language called iTask [13, 14]. The iTask system is a *combinator library*. In the functional programming community, combinators are a proven method to embed domain specific languages within a functional host language: application patterns are captured with *combinator functions*, and the application domain is defined by means of the expressive type system, using algebraic, record, and function types. Workflows modeled in iTask result in complete workflow applications that run on the web distributed over server and client side [16].

The remainder of this paper is organized as follows. We present iTask in Sect. 2, and demonstrate the advantages of using a functional programming language as workflow modeling language. We continue with a larger example in Sect. 3. We discuss the major design decisions in Sect. 4. Related work is discussed in Sect. 5. We conclude in Sect. 6.

2 Overview of the iTask system

In this section we give an overview of the iTask system. We start with basic iTasks in Sect. 2.1. We present only a small subset of the available combinators in Sect. 2.2. In Sect. 2.3 we show how embedding iTasks in a general purpose language makes the system extensible and adaptable.

2.1 Basic iTasks

An iTask is a unit of work to be performed by a worker or computer. An iTask can be in different states. It can be: non-active (does not exist yet), active (someone is still working on it), or finished. Clean is a statically typed language, everything has a type. A type can be regarded as a model and a value of that type as an instance of that model. An iTask has the following opaque, parameterized type:

```
:: Task a
```

The type parameter `a` is the type of the value that is delivered by a task. The Clean compiler infers and checks the concrete type of any specified task.

The iTask library offers several functions for creating basic units of work: a basic task. For instance, the function `editTask` takes a `label` of type `String` and an initial `value` of some type `a` and creates a `Task a`: a form in a web page in which the worker can edit this initial `value`. Its type is:

```
editTask :: String a → Task a | iData a
```

The function `editTask` is very powerful. It creates an editor for *any* first-order concrete type and handles *all* changes. A worker can change the value as often as she likes but she cannot alter its type. When the `label` button is pressed, the `editTask` is finished and the final value is delivered as result.

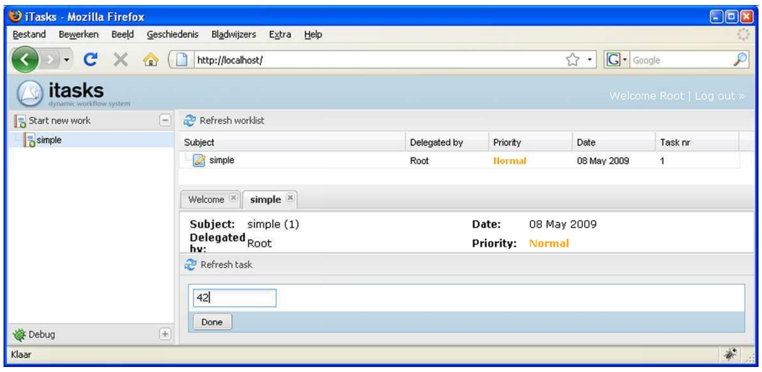
`EditTask` is not a polymorphic function (i.e. *one* function which works for *any* type), but it is overloaded. For each type it is applied on, a special version is constructed. One can regard `editTask` as a kind of *type driven* or *model driven* function. The precise working of the function depends on the concrete type (model) on which it is applied. This is *statically* determined. The type dependent behavior is inductively defined on a small number of generic, type driven functions [2], which is reflected in the type context restriction (`| iData a`) in the type definition of `editTask`. A small, but complete iTask workflow application is:

```
module SmallButCompleteExample                                          1.
import iTasks                                                           2.

Start       :: *World → *World                                          3.
Start world  = startEngine [addFlow ("simple", simpleEditor)] world     4.

simpleEditor :: Task Int                                                5.
simpleEditor = editTask "Done" createDefault                            6.
```

The application imports the iTask library (line 2). Execution begins with the `Start` function. It calls the iTask engine, and adds the workflow `simpleEditor` to the workflow list that can be invoked by workers. `SimpleEditor` (line 6) only consists of one invocation of `editTask`. Eventually, it produces an `Int` value, indicated by the type definition (line 5). The button labeled `Done` finishes the task. The library function `createDefault` is used as initial value. It creates a default value for *any* type (hence it is also a generic function). The default value for type `Int` is 0.



Subject	Delegated by	Priority	Date	Task nr
simple	Root	Normal	08 May 2009	1

Subject: simple (1)
Delegated by: Root
Date: 08 May 2009
Priority: Normal

42

Fig. 1. Generated webpage for the `simpleEditor` example.

The generated web application is shown in Fig. 1. After logging in, the application resembles a regular e-mail application. The names of the tasks that the worker needs to perform are presented in the *task list* displayed in the right upper pane. This pane can be compared with the list of incoming e-mails. When the worker clicks on a task in the task list, the current state of it is displayed in the right lower *task pane*. Tasks can be selected from the task list in any order. The iTask toolkit automatically keeps track of all progress, even if the user quits the system. When a task is finished, it is removed from the task list. Workers can start new workflows, by selecting them in the left *workflow pane*. In our example there is only one option. In general arbitrarily many workflows can be started, one can assign different workflow options for different types of workers, and the options to choose from can be controlled dynamically. The task list is updated when new tasks are generated, either on her own initiative, or because they have been delegated to her. The entire interface is generated completely and automatically from the sole specification shown above.

In the above example the worker can only enter `Integer` values. Suppose we want a similar workflow for a custom model type, say *person*. We define the necessary domain types (`Person` and `Gender`), derive framework code for these model types, and change the type of `simpleEditor` to `Task Person` (Fig. 2).

Hence, the form that is created depends on the type of the value that the editor should return. For any (user defined) type a standard form can be automatically generated in this way. The details of how a form is actually represented can be fine-tuned in a separate `CSS` file. A programmer can specialize the form generation for a certain type if a completely different view is wanted. One is

```
:: Person = { firstName   :: String
            , surName     :: String
            , dateOfBirth :: HtmlDate
            , gender      :: Gender
            }
:: Gender = Male | Female
derive iData Person, Gender
```

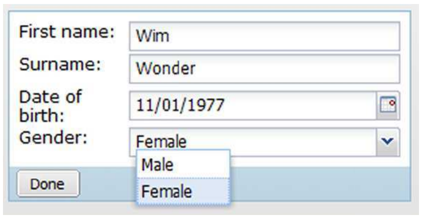

First name: Wim
Surname: Wonder
Date of birth: 11/01/1977
Gender: Female

Fig. 2. A standard form editor generated for type Person.

not restricted to use standard browser forms. It is e.g. possible to use a drawing plug-in as editor for making values of type, say `Picture` [8].

The function `editTask` is one of several basic tasks. Examples of other basic task functions are: obtaining all users of the system (if necessary grouped by their role); tasks that return at a predefined moment in time or after an amount of time; tasks that can store information in or read information from a database.

2.2 Basic iTask Combinators

New tasks can be composed out of (basic) tasks by using *combinator* functions. As said before, work can be organized in many ways. The expressive power of the host language allows us to cover all common workflow patterns listed in [1], and many more, using only relatively few combinators. Here we discuss a few of them and show their usage.

Sequential composition of tasks can be realized using *monadic* [20] combinator functions for binding (>>=), called *bind*, and emitting (`return`) values:

```
(>>=) infixl 1 :: (Task a) (a → Task b) → Task b | iData b
return         :: a                     → Task a | iData a
```

In contrast to most workflow specification languages, in the iTask system information is passed explicitly from one task to another. The first task (of type `Task a`) is activated first and when it finishes, >>= takes care that its result (of type `a`) is passed to the second argument (a *function* of type `a → Task b`). This function can inspect the result produced by the previous task and react on it. When the second task is finished it produces a value of type `b`. The result of bind is a task of type `Task b`. Note that t >>= f integrates *computation* and *sequential ordering* in a single pattern. This is hard to specify in a graphical modeling language. The `return` combinator lifts any value (of some type `a`) to a (`Task a`) that yields that value. Before we give an example of sequential composition, we introduce one of the *choice* operators: `chooseTask`.

```
chooseTask :: [HtmlTag] [(String,Task a)] → Task a | iData a
```

A worker is offered a choice out of a list of tasks with `chooseTask`. Each task is identified with a label (of type `String`) which is used to display the options to the worker. When one of the options is chosen, the corresponding task is selected and performed. The additional prompt argument (of type [HtmlTag]) can be used to explain to the worker what the intention is. Any Html-code can be used.

```
chooseUser :: Task (UserId,String)
chooseUser
=              getUsers
  >>= λusers → chooseTask  [Text "Select worker who has to do the job:"]
                            [(name,return user) \\ user=:(_,name) ← users]
```

The example task chooseUser uses several of the basic tasks and combinators introduced so far. The basic task getUsers returns the list of known users. These are passed to chooseTask to allow the worker to select one known user. The *list comprehension* in line 5 is well known functional idiom for manipulating lists.

Tasks can be *assigned* to workers. This is done with the @: combinator:

```
(@:) infix 3 :: UserId (String, Task a) → Task a | iData a

delegateTask :: String (a → Task a) a → Task a | iData a
delegateTask taskname taskf val
=   chooseUser >>= λ(user,_) → user@:(taskname, taskf val)
```

The @: operator is a basic combinator with which any task can be assigned to a worker (type UserId). The label (of type String) gives a name to the task (displayed in the task list of this specific user). DelegateTask shows its usage: first a worker is chosen with chooseUser and this worker is assigned the given task.

Workers can be *prompted* about their progress with ?>>:

```
(?>>) infixr 5 :: [HtmlTag] (Task a) → Task a | iData a

taskToDelegate :: a → Task a | iData a
taskToDelegate value
=   [Text "Would you be so kind to fill in the following form:"]
    ?>> editTask "TaskDone" value
```

The ?>> combinator can be used to add a prompt to any task. The prompt (any Html-code can be used here again) is displayed as long as the task is not finished. This facility is used in taskToDelegate to make a polite version of editTask.

Notice that all definitions above can be applied for any task of any type (read: model). In concreteDelegate below we apply delegateTask to a general task for filling a form (taskToDelegate) given some initial value (createDefault). As was the case with a simple form editor, the type specified here completely determines the actual form to be filled in by the chosen worker:

```
concreteDelegate :: Task Person
concreteDelegate = delegateTask "person" taskToDelegate createDefault
```

Finally, iTask specifications can use all features of Clean including *recursion*.

```
recursiveDelegate:: String (a → Task a) a →  Task a  | iData a
recursiveDelegate taskname taskf val
=                 delegateTask taskname taskf val
   >>= λresult → chooseTask [Text "Result:", toHtml result, Text "Approved ?"]
```

```
                    [ ("Yes",return result)                                          5.
                    , ("No", recursiveDelegate taskname taskf result)                6.
                    ]                                                                7.

concreteRecDelegate :: Task Person                                                   8.
concreteRecDelegate = recursiveDelegate "person" taskToDelegate createDefault       9.
```

A variant of delegateTask is given in recursiveDelegate. The worker who delegates the task receives the result back (in result) for examination. It is shown in the prompt of chooseTask: toHtml (line 4) displays any result of any type. If the worker chooses not to approve the result, recursiveDelegate is called recursively and the delegation of the work starts all over, but now with the latest result as starting point (line 6). Again this recursive version can be used for any task and any type: it is turned into a Person workflow by concreteRecDelegate (line 8).

2.3 The Expressive Power of the Combinators

In [1] an inventory is made of the workflow patterns offered by commercial WFMSs. This collection is rather large. The reason for this is that the underlying workflow languages generally do not offer the right abstraction mechanism for defining new patterns. In this section we show how iTask makes use of the functional host language to capture new patterns concisely.

Two commonly available workflow patterns are orTasks and andTasks. Both work on a collection of tasks but have different termination conditions: orTasks finishes as soon as one of its subtasks finishes, andTasks finishes as soon as all subtasks are finished. We can capture this common behavior with a more general combinator, parallel, that can be used to define orTasks and andTasks:

```
parallel :: String ([a] → Bool) (Bool → [a] → b) [(String, Task a)] → Task b
          | iData a & iData b

orTasks  = parallel "orTasks"  (not o isEmpty) (const hd)
andTasks = parallel "andTasks" (const False)   (const id)
```

The parallel combinator is given a predicate (of type [a] → Bool) which determines when to stop. The predicate is applied on the list of values of all completed subtasks. As soon as the predicate holds, all unfinished subtasks are stopped (even if other workers are working on it) and the results of the finished tasks are collected and converted (by the function of type Bool → [a] → b) to the type b demanded by the application. The Bool argument is true iff the predicate was satisfied. Parallel also terminates when all subtasks have terminated, but the predicate is still invalid. With parallel, orTasks and andTasks are easily expressed, as shown above. (The const function ignores the boolean argument and applies its argument function to the list of results.) Many other imaginable and useful patterns can be defined in a similar way, e.g. when one wants to stop as soon as enough information has been produced by the finished tasks (see also Sec. 3), or as soon as someone has given up. Such user defined stop criteria cannot be defined in most commercial workflow languages.

Another useful pattern is the ability to cancel a task (even when performed by someone else), which can concisely be expressed with `orTasks` as shown in the function `cancelable`.

```
:: Maybe a = Just a | Nothing

cancelable :: (Task a) → Task (Maybe a) | iData a
cancelable task
= orTasks [("Cancel",editTask "Cancel" Void >>= λ_ → return Nothing )
          ,("Normal",task >>= λresult → return (Just result))
          ]

cancelableDelegate = cancelable concreteRecDelegate
```

The `Void` type has no visualization, hence `editTask` only shows a button labeled `Cancel`. `Nothing` is returned when `Cancel` is pressed.

The iTask library has several of these general purpose combinators, but there is no room here to discuss them all. There are combinators for workflow process creation and handling, thread handling, exception handling, and combinators which enable the change of work under execution. Compared to the set of well-known workflow patterns, we have far less combinators yet we can express a lot more different work situations. The reason is that with Swiss-army-knife combinators such as `parallel` and the expressive power of the host language we can construct not only many well-known workflow patterns, but also new variants, as explained above. Furthermore, the system is open ended: the programmer can add new combinators when the basic collection is insufficient.

3 Order Booking Example

To demonstrate the expressive power of iTask, we present an *order booking* example. The code presented below is a complete, executable, iTask workflow. The workflow has a recursive structure and monitors intermediate results in a parallel and-task. This case study is hard to express in traditional workflow systems. The overall structure contains the following steps (see `getSupplies` below): first, an inventory is made to determine the required amount of goods (`getAmount`) (e.g. vaccines for a new influenza virus); second, suppliers are asked in parallel how much they can supply (`inviteOffers`); third, as soon as sufficient goods can be ordered, these orders are booked at the respective suppliers (`placeOrders`). A `Supplier` is a pair (`UserId`,`String`), and `Amount` is a non-negative `Int`.

```
getSupplies :: Task [Void]                                          1.
getSupplies = getAmount >>= inviteOffers >>= placeOrders            2.
```

Determining the required amount of goods also proceeds in two steps: first, the institute enquires other institutes *recursively* in parallel (using the `andTasks` combinator) how many goods they need (lines 6-9). This means that each of these institutes can ask other institutes for the same thing, and so on. Note that an institute can decide to select no other institutes. In that case the recursion stops. Second, given this amount, the institute can alter this number (line 11). Also, `chooseUsers` (line 6) is a variant of `chooseUser` (Sect. 2.2) that uses *multiple choice* and yields a list of workers. The operator `<+` converts its second argument to `String` and concatenates it to its first argument.

```
getAmount :: Task Amount                                                    3.
getAmount =          [Text "Ask other institutes"] ?>> chooseUsers          4.
    >>= λinsts  →   andTasks [( "Request for " <+ name                      5.
                              , uid @: ("Amount request", getAmount)        6.
                              ) \\ (uid,name) ← insts]                      7.
    >>= λothers →   [Text "Enter the required amount"]                      8.
                    ?>> editTask "Done" (sum others)                        9.
```

Once the amount of goods is established, the workflow can continue by inviting offers from a collection of candidate suppliers. This collection is determined first (line 14). Each supplier can provide an amount (line 18). This is again done in parallel (line 15-20). The termination criterium is the `enough` predicate which is satisfied as soon as the sum of provided offers exceeds the requested amount (line 22). The canonization function `maximum` is discussed below. Hence, the result of this workflow task is a list of offers. Each offer is a pair of a supplier and the amount of goods that it offers to deliver.

```
inviteOffers :: Amount → Task [(Supplier,Amount)]                           10.
inviteOffers needed                                                         11.
=              [Text "Choose candidate suppliers"] ?>> chooseUsers          12.
  >>= λsups → parallel "Supplier_requests" enough (maximum needed)          13.
              [("Request for " <+ name                                      14.
                ,uid @: ("Order request"                                    15.
                        ,prompt ?>> editTask "Done" needed >>= λa → return (sup,a) 16.
                )       )                                                   17.
              \\ sup=:(uid,name) ← sups                                     18.
              ]                                                             19.
where enough as = sum (map snd as) >= needed                                20.
      prompt    = [Text "Request for delivery, how much can you deliver?"]  21.
```

The total number of offered goods can differ from the required number of goods. The function `maximum` makes sure that not too many goods are ordered. This is an easy exercise in functional programming:

```
maximum :: Amount Bool [(Supplier,Amount)] → [(Supplier,Amount)]            22.
maximum needed enough offers                                                23.
| not enough              = offers                                          24.
| otherwise               = [(supplier,exact) : less]                       25.
where [(supplier,_) : less] = sortBy (λ(_,a1) (_,a2) → a1 > a2) offers      26.
      exact               = needed - sum (map snd less)                     27.
```

With the correct list of offerings, we can place an order for each supplier. This can be expressed directly with `andTasks`:

```
placeOrders :: [(Supplier,Amount)] → Task [Void]                            28.
placeOrders offers                                                          29.
=   andTasks [("Order for " <+ name,                                        30.
               uid @: ("Order request for " <+ name
                      ,[Text ("Please deliver " <+ a)] ?>> editTask "Done" Void)
                      )
             \\ ((uid,name),a) ← offers ]
```

4 Modeling by Abstracting from Details

An iTask workflow specification is modeled in a functional language. This means that the host language needs to support abstraction from (almost) all annoying details. On the other hand, all information has to be provided somehow because from this single source specification a complete real working distributed web application has to be generated. How has this been realized?

Defining General Workflow Schemes. By using polymorphic, overloaded, and generic (type driven) functions, one can specify custom workflow schemes for any frequently occurring work situation that works for *any* concrete task of *any* type. The type system ensures that everything is type correct. To get a working application, a scheme somewhere has to be applied to a concrete model (i.e. type). This is used by the compiler to generate specialized functions that handle the low level details.

Abstracting from Form Views and Form Handling. One does not have to worry about the user interface as a whole. It is defined separately, handled by the client and it can be fine-tuned if desired without affecting the workflow specification. Using generic programming techniques, an editor can be generated for any concrete type specified by the programmer, the information is displayed in a web page and can be interactively modified. Any change made in the form is handled automatically by the application. The input given interactively is type checked; it is impossible to create ill-typed values. It is also possible to specify user-defined predicates over the input which consistency is checked at run-time.

Abstracting from specific Html Output. Sometimes it is necessary to show additional information to the worker, e.g. for prompting and feedback. Abstracting from this is not always completely possible. However, with the function `toHtml` any value of any type can be converted to Html-code (Sec. 2). Furthermore, it is possible to abstract from a concrete prompt or feedback by defining functions for it, which can be given as argument to `?>>` (Sec. 2).

Abstracting from Layout. Lay-out details of the output generated can be specified separately as is common in web applications. The dynamic behavior of the iTask system requires additional run-time facilities for controlling the lay-out.

```
:: TaskCombination
= TTSplit [HtmlTag] | TTVertical| TTHorizontal | TTCustom ([[HtmlTag]] → [HtmlTag])

myCancelable task = cancelable task <<@ TTVertical
```

To influence the layout of Html-code generated by subtasks, one can use the (overloaded) tuning operator `<<@`. One can place the Html-code of each subtask in a separate worktab (`TTSplit`), below each other(`TTVertical`), next to each other (`TTHorizontal`), or give a user defined recipe how to combine it (`TTCustum`). `MyCancelable` gives an example of its usage. `<<@` can be used to fine tune a system, but the application also works without it.

Abstracting from Storage. In the iTask system information is passed explicitly from one task to another. This may involve different workers. Between the events all information has to be stored somewhere. For a web-application there are many possibilities, e.g. in a file, in a database, on the client. Furthermore, we need to remember the progress of every worker. Again, by using generic programming techniques, all this information is stored fully automatically. With the overloaded tuning operator `<<@` the workflow programmer can decide *where* and in what *format* the information is stored (text, binary).

An iTask application also needs access to information stored in standard information systems. We do not have to bother workflow programmers with something low level as SQL queries. We can systematically convert an information model defined in e.g. ORM to Clean data type definitions. This enables the automatic conversion between values of these types and the corresponding values stored in a relational database [10].

Abstracting from the Web Architecture. The architecture (Fig. 3) of iTask applications is representative for Ajax [5] based web applications. Initially a web page is generated as shown in Fig. 1. Whenever a worker clicks on something, an

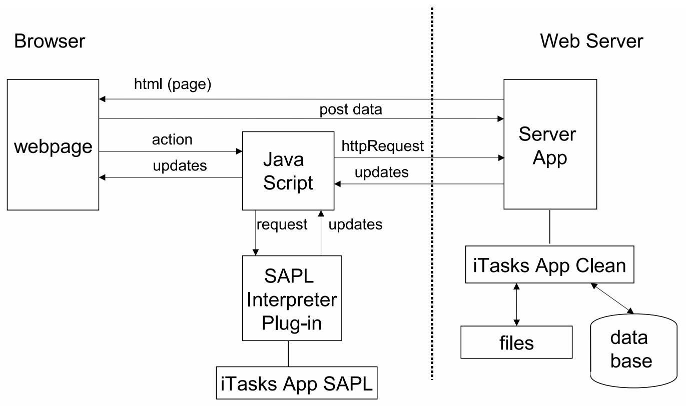


Fig. 3. The architecture of an iTask application

asynchronous Ajax request is sent to the iTask application to handle it. It reacts by sending an update of those parts of the page that have to be changed. All web handling is done automatically. Although one is defining a multi-user web application, one does not need any knowledge about the underlying architecture.

One of the most interesting aspects of the architecture is that it is possible to execute, in principle, *any* iTask on either the server or on a client [16]. For example, a function like `concreteDelegate` (Sect. 2.2) is executed on the server by default. It can be executed on the client by using the tuning operator `<<@` and the

`OnClient` pragma: `concreteDelegate <<@ OnClient`. The `<<@` operator accepts *any* task. However, not all tasks can be executed (completely) on the client (e.g. tasks that access a database). This is detected at run-time and the system automatically migrates such a task back to the server.

To implement this, the Clean compiler generates *two* executable instances from a single source. An executable iTask system runs on the server, while an interpreted version runs on every client. Both implementations are efficient. The Clean compiler is well-known for the excellent code it generates which is comparable to C. In the browser we use the Sapl-interpreter [7] which is currently one of the fastest interpreters for a functional language. One JavaScript function decides for each user action whether it is handled on the client or on the server.
Abstracting from the Evaluation Order. The iTask combinators define the order in which tasks can be executed: tasks can be performed sequentially, in parallel, distributively (on clients), and there even is strong support for exception handling. Notice that iTasks are just plain Clean applications (there is no special interpreter for iTask applications or something like that). A functional language like Clean evaluates expressions in a fixed way: lazily (normal order). Clean does not support parallel evaluation, distributed evaluation, nor exception handling. It is very remarkable that the wild scala of evaluation orders offered by the iTask library can nevertheless be embedded in the host language. This has been achieved by making clever use of generic techniques for the automatic storage of the state of the iTask application in combination with re-evaluation of (part of) the application [13]. This allows us to mimic any evaluation strategy without the need to make any change or extension in the host language.

5 Related Work

The WebWorkFlow project [6] shares our point of view that a workflow specification is regarded as a web application. WebWorkFlow is an object oriented workflow modeling language that is embedded in WebDSL [19], a domain specific language for developing web applications. In WebWorkFlow, *workflow objects* accumulate the progress made in a workflow. *Workflow procedures* define the actual workflow. Their specification is broken down into *clauses* that individually control *who* can perform *when*, what the *view* is, what should be *do*ne when the workflow procedure is applied, and what further workflow procedures should be *process*ed afterwards. Like in iTask, one can derive a GUI from a workflow object. The main difference is that iTask is embedded in a functional language, but this has significant consequences: iTask supports higher-order functions in both the data models and the workflow specifications; arbitrary recursive workflows can be defined (WebWorkFlow is restricted to tail recursion and recursion on simpler structures); reasoning about the evaluation of an iTask program is reasoning about the combinators instead of the collection of clauses.

Brambilla *et al*[4] enrich a domain model (specified as UML entities) with a workflow model (specified as BPMN) by modeling the workflow activities as additional UML entities and use OCL to capture the constraints imposed by the workflow. The similarity with iTask is to model the problem domain separately. However, in iTask a workflow is a function that can manipulate the model values in a natural way, which enables us to express functional properties seamlessly (Sect. 3). This connection is ignored in [4] and can only be done ad-hoc.

Pešić and van der Aalst [12] base an entire formalism, ConDec, on linear temporal logic (LTL) constraints. Frequently occurring constraint patterns are represented graphically. This approach has resulted in the DECLARE tool [11]. In iTask a workflow can use the rich facilities of the host language for computations and data declarations – such facilities are currently absent in DECLARE.

Andersson *et al*[3] distinguish high level *business models* (value transfers between *agents*), low level *process models* (workflows in BPMN), and medium level *activity dependency models* (activities for value transfers of business models). Activities are *value transfer*, *assigning* an agent to a value transfer, value *production*, and *coordination* of mutual value transfers and activities. Activities are modeled as nodes in a directed graph. The edges relate activities in a way similar to [4] and [12]: they capture the workflow, but now at a conceptual level. A *conformance relation* is specified between a process model and an activity dependency model. Currently, there is no tool support for their approach. The activity dependency models provide a declarative foundation to bridge the gap between business models and process models. One of the goals of the iTask project is to provide a formalism that has sufficient abstraction to accomodate both business models and process models.

Vanderfeesten *et al*[18] have been inspired by the *Bill-of-Material* concept from manufacturing, recasted as *Product Data Model* (PDM). A PDM is a directed graph. Nodes are product data items, and arcs connect at least one node to one target node, using a functional style computation to determine the value of the target. A tool can inspect which product data items are available, and hence, which arcs can be computed to produce next candidate nodes. This allows for flexible scheduling of tasks. Similarities with the iTask approach are the focus on tasks that yield a data item and the functional connection from source nodes to target node. We expect that we can handle PDM in a similar way in iTask. iTask adds to such an approach strong typing of product data items (and hence type correct assembly) as well as the functions that connect these data items.

6 Conclusions and Future Work

The iTask system demonstrates that a high level general purpose functional language such as Clean (or Haskell which is also supported by the Clean compiler) is very suited to embed a special purpose modeling language. A library provides the domain specific constructs and the specification language inherits the power and advantages of the embedding language. Because only pure functions can be used, one is forced to model in a mathematical way. Strong typing prevents modeling mistakes. The use of generic (type driven) functions enable the use of types as models. Powerful abstraction mechanisms allow to abstract from annoying details such that one can concentrate on modeling. With relatively few combinators all common workflow patterns are supported, but many more complex workflow situations can be expressed. iTask workflows are dynamic: the flow can depend on the outcome of other tasks. From the specification we are able to generate a real working, distributed evaluated, web enabled, multi-user workflow system. Any task can be shifted from the server to the client. The whole system is generated from one source: a concise iTask specification defined in Clean. Reasoning about the behavior of the system is relatively easy. The semantics of the iTask system and the properties it has are treated in [9].

We have tested the system with larger examples, such as a Conference Management System [15]. We are extending our prototype to a full system which can be applied in industrial environments. In collaboration with the Netherlands Ministry of Defense we are investigating the suitability of the iTask system for handling operational planning and crisis management scenarios. To support these kind of complex dynamic applications we are currently adding the ability to adapt and/or extend running workflows on the fly.

References

1. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. QUT technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia, 2002.
2. A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Selected Papers of the 13th International Symposium on the Implementation of Functional Languages, IFL'01*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, Sept. 2002.
3. B. Andersson, M. Bergholtz, and A. Edirisuriya. A Declarative Foundation of Process Models. In O. Pastor and J. Cunha, editors, *Proceedings 17 Int'l Conference on Advanced Information Systems Engineering, CAiSE 2005*, volume 3520 of *LNCS*, pages 233–247. Springer-Verlag, 2005.
4. M. Brambilla, J. Cabot, and S. Cornai. Automatic Generation of Workflow-Extended Domain Models. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Proceedings Model Driven Engineering Languages and Systems, 10th Intl' Symposium, MoDELS 2007*, volume 4735 of *LNCS*, pages 375–389. Springer-Verlag, 2007.
5. J. Garrett. Ajax: a new approach to web applications, 18, Feb. 2005.
6. Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS'08*, volume 5301 of *LNCS*, pages 113–127. Springer-Verlag, 2008.
7. J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.
8. J. Jansen, P. Koopman, and R. Plasmeijer. iEditors: extending iTask with interactive plug-ins. In S.-B. Scholz, editor, *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, pages 170–186, Hertfordshire, UK, 10-12, Sept. 2008. University of Hertfordshire.

9. P. Koopman, R. Plasmeijer, and P. Achten. An executable and testable semantics for iTasks. In S.-B. Scholz, editor, *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, pages 53–64, Hertfordshire, UK, 10-12, Sept. 2008. University of Hertfordshire.
10. B. Lijnse. Between types and tables - Generic mapping between relational databases and data structures in Clean. Master's thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, July 2008. Number 590.
11. M. Pešić. *Constraint-based workflow management systems: shifting control to users.* PhD thesis, Technical University Eindhoven, 8, Oct. 2008.
12. M. Pešić and W. van der Aalst. A declarative approach for flexible business processes management. In J. Eder and S. Dustdar, editors, *Proceedings of the 1st Business Process Management Workshop on Dynamic Process Management, DPM'06*, volume 4103 of *LNCS*, pages 169–180. Springer-Verlag, 2006.
13. R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP'07*, pages 141–152, Freiburg, Germany, 1-3, Oct. 2007. ACM Press.
14. R. Plasmeijer, P. Achten, and P. Koopman. An introduction to iTasks: defining interactive work flows for the web. In *Selected Lectures of the 2nd Central European Functional Programming School, CEFP'07*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, 2008. Springer-Verlag.
15. R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, and T. van Noort. An iTask case study: a conference management system. In *Selected Lectures of the 6th International Summer School on Advanced Functional Programming, AFP'08*, LNCS, Center Parcs "Het Heijderbos", The Netherlands, 19-24, May 2008. Springer-Verlag.
16. R. Plasmeijer, J. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP'08*, pages 56–66, Valencia, Spain, 15-17, July 2008.
17. N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst. Workflow resource patterns: identification, representation and tool support. Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, Australia, 2004.
18. I. Vanderfeesten, H. Reijers, and W. van der Aalst. Product based workflow support: dynamic workflow execution. In Z. Bellahsène and M. Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering, CAiSE'08*, volume 5074 of *LNCS*, pages 571–574, Montpellier, France, 2008. Springer-Verlag.
19. E. Visser. WebDSL: a case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. a. Saraiva, editors, *Selected Lectures of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE'07*, volume 5235 of *LNCS*, pages 291–373, Braga, Portugal, 2-7, July 2007.
20. P. Wadler. Comprehending monads. In *Proceedings of the 6th Conference on Lisp and Functional Programming, LFP'90*, pages 61–77, Nice, France, 1990.

An effective methodology for defining consistent semantics of complex systems

Pieter Koopman, Rinus Plasmeijer, and Peter Achten

Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{pieter, rinus, p.achten}@cs.ru.nl

Abstract. The semantics of a complex system is a formal description of the required behavior of that system. Although the formal semantics is usually a much simpler system than the original system described, the semantics itself can become quite difficult as well. Often the semantics itself is error prone just like any other large formal system (e.g. a program). There are several kinds of potential problems with such a formal system: **1)** the system is incomplete, e.g. not all notions used are defined properly; **2)** the system is inconsistent, e.g. functions are used with the wrong number of arguments; **3)** the system does not possess the required properties, e.g. addition is not associative, commutative and distributive; **4)** the system does not prescribe the right semantic, e.g. the semantics of multiplication accidentally specifies addition. In this lecture we show an effective way to tackle these problems and illustrate our method by a nontrivial example.
Problems 1) and 2) are avoided by using a high level functional programming language as carrier of the formal semantics. The burden of using such a language as formalism instead of ordinary mathematics appears to be very limited. The advantage is obvious; the compiler checks consistency and well-definedness of the semantics. Problem 4) can now be handled by simulating the specified semantics. In our experience such a validation is very effective. In this lecture we show how problem 3) can be treated by stating desired properties of the semantics and test these properties automatically with the model-based test system G∀st. During the development of the semantics one does not want the burden of formal proofs of properties for intermediate versions of the semantics; a test system yields a solid approximation of correctness within seconds. In a next step one can use this semantics to test the real implementation of the specified system with G∀st.
As an example of a complex system we will take the iTask workflow system as currently being developed in Nijmegen. Workflow systems are automated systems which coordinate the tasks that have to be done to achieve a certain goal. These tasks are executed by human beings and computers. In Clean we have defined the iTask combinator library with which dynamic workflows can be defined: the kinds of tasks to do, the assignment of tasks to workers and the order in which the tasks are executed is not fixed, but may depend on the outcome of previous tasks. The system generated is a multi-user distributed web application in which the tasks to be done are offered to the intended workers. Due

to the architecture of the web and the fact that tasks can be executed both on the server and on the client, the iTask system is becoming quite an intriguing system.
Describing the semantics of and properties of such a system and proving its correctness is therefore not an easy job. The system is complex on its own and it is on forehand not clear which properties it exactly has. Furthermore, the iTask system is still under development, such that is too early to use heavy and time consuming equipment like a proof system. Yet we would like to be sure that the system we have defined so far behaves well even though we are not sure yet in all detail what that means.

1 Introduction

The iTask system [11] is an experimental toolkit to specify data dependent dynamic workflows in a flexible and concise way by a set of combinators. The iTask system supports workers executing the specified tasks by a web-based interface. Typical elementary user tasks in this system are filling in forms and pressing buttons to make choices. The elementary tasks are implemented on top of the iData system [10]. Based on an input the iTask system determines the new task that has to be done and updates the interface in the browser. Arbitrary complex tasks are created by combining (elementary) tasks. The real power of data dependent tasks is provided by the monadic bind operator that contains a *function* to generate the next task based on the value produced by the previous task.

The iTask implementation executes the tasks, but has to cope with many other things at the same time: e.g. i/o to files and database, generation of the multi-user web interface, client/server evaluation of tasks, and exception handling. The iTask system uses generic programming to derive interfaces to files, databases and web-browsers for data types. The combination of these things makes the implementation of iTasks much too complicated to grasp the semantics. To overcome these problems we develop a high level operational semantics for iTasks in this paper. The semantics in this paper is an extended version of our earlier work published in [6]. This semantics is used to explain the behavior of the iTask system, and to reason about the desired behavior of the system. In the future we will use this semantics as model to test the real iTask implementation with our model-based test tool G∀st. A prerequisite for model-based testing is an accurate model of the desired behavior. Making a model with the desired properties is not easy. Such a model is developed, validated, and its properties are tested in this paper.

In this paper we provide a basic rewrite semantics for iTasks as well as a number of useful notions to reason about tasks, such as *needed events* and *equivalence* of tasks. The semantics of many other workflow systems is based on Petri-nets [12], actor-oriented directed graphs (including some simple higher order constructs) [8], or abstract state machines (ASM) [7]. Neither of these alternatives is capable to express the flexibility covered by the dynamic generation of tasks of the monadic bind operation of the iTask system. As usual we omit many details in the semantics to express the meaning of the basic iTask combinators as clearly as possible. The semantics is expressed in the functional programming language Clean instead of the more common Scott Brackets, denotational semantics, or horizontal bar style, structural operational semantics a la Plotkin. The close correspondence between semantics and functional programs goes back at least to [9]. Expressing the operational semantics in a FPL is as concise as in Scott Brackets style. Using a functional programming language as carrier of the specification of the semantics has a number of advantages: **1)** the type system performs basic consistency checks on the semantics; **2)** the semantics is executable; **3)** using the iTask system it is easy to validate the semantics by interactive simulation; **4)** using the model-based test tool G∀st [5] it is possible to express properties about the semantics and equivalence of task concisely, and to test these properties fully automatically. Although the semantics is executable, it is not an iTask system itself. The semantics is a model of the real system, it lacks for instance a frontend (user interface) as well as a backend (e.g. interface to a database).

Especially the ability to express properties of the specified semantics and to test them automatically appears to be extremely convenient in the development of the semantics and associated notions described in this paper. An alternative, more traditional, approach would be to define a semantics in a common mathematical style, state properties in logic, and formally prove these properties. It would be wise to use a proof assistant like COQ [14] or SPARKLE [3] in proving the properties, this would require a transformation of the semantics to the language of the proof assistant. In the past we have used this approach for the iData system [1]. In such a mathematically based approach it is much harder to experiment with different formulations of the semantics and to get the system consistent after a change. Proving a property of a new version of the semantics typical takes some days of human effort where testing the same property is done in seconds by a computer. When we have a final version of the semantics obtained in this way, we can decide to prove (some of) the tested properties in order to obtain absolute confidence in their correctness.

Section 2 gives a short introduction to the various kind of semantics that exists and shows how this can be expressed concisely in a functional language. In section 3 we show how we model iTasks and the effect of applying an input to a task. In this section we also define useful notions about subtasks, such as when they are enabled or needed. In section 4 we define the equivalence of tasks and how the equivalence of tasks can be determined in two different ways. Some important properties of the semantics of iTasks are given in section 5, we also show how these properties can be tested fully automatically. Testing properties of the semantics increases the confidence that we have specified the semantics of iTasks right. In the future we will use this semantics for model-based testing of the real iTask implementation, this will increase the confidence that the system obeys the semantics. Finally there is a discussion.

2 Formal Semantics

The semantics [9] of a programming language defines the meaning of programs. In order to reason about the semantics we usually need a formal definition of the semantics, rather than an informal textual description. There is a considerable amount of research done in this field. As a result there are many different ways to describe the semantics of some programming language. The three main approaches are:

Operational semantics This approach assigns a meaning to the syntactical constructs in a programming language. The meaning of each language construct is specified by the computations it induces when the construct is executed. This approach focusses on how the effect of each language construct is build in some kind of mathematical interpretation of the programming language.
Within the operational semantics we distinguish two approaches. The *structural operational semantics* (or *small-step* semantics) focusses on details of individual execution steps. The semantics is a mathematical interpreter of the programming language that hides details like storage allocation, but otherwise specifies the effect of each language construct in detail. The *natural semantics* (or *big-step* semantics) hides even more details. There is no emphasis on specifying the meaning of individual execution steps, but one specifies the meaning of a language constructs in larger steps than in the structural operational semantics.

Denotational semantics The focus in this approach is on the value that a language construct denotes, rather than how this value is obtained. For simple language constructs the denotational description resembles the operational semantics. For more complicated constructs like loops and recursive function applications one often uses a fixed point description. For instance, for a loop the denotational semantics specifies the state of the program after termination of the loop, while the operational semantics describes the state change by executing the loop once.

Axiomatic semantics In this approach specific properties of executing language constructs are expressed as assertions. These properties specify the effect of language constructs, but there can be aspects of the execution that are ignored in an axiomatic semantics.

2.1 The Imperative Language While

In order to illustrate these approaches to specify the semantics and our own approach based on functional programming, we specify the semantics of a very simple imperative language called While in this section. We will start with the classical formal semantical description. Since this approach assigns a meaning to the syntactical constructs in a programming language, we need a specification of the syntax. For the language While we use the following meta-variables and catagories:

n ranges over integer numbers;
v ranges over variables;
a ranges over arithmetical expressions;
b ranges over Boolean expressions;
S ranges over statements.

These meta-variables can be primed or subscripted if we need more than one instance, e.g. v and v_1 are also variables. We do not need to known the syntactical details of numbers and variables. The syntax for the other constructs is:

$$\begin{aligned}
a &::= v \mid n \mid a + a \mid a - a \mid a * a \\
b &::= \texttt{TRUE} \mid \texttt{FALSE} \mid a = a \mid a < a \mid \neg\, b \mid b \;\&\&\; b \\
S &::= x \texttt{ := } a \mid \texttt{skip} \mid S\,;\,S \mid \texttt{if}\ b\ S\ \texttt{else}\ S \mid \texttt{while}\ b\ S
\end{aligned}$$

2.2 The Semantics of Expressions in While

In order to specify the semantics we need a *state* that relates variables and their values. Usually this state is modeled as a function from variables to their values: $\mathsf{State} = \mathsf{Var} \rightarrow \mathsf{Int}$. Here we have to make our first semantic choice: do variables that did not occur on the left-hand side of an assignment have a value (and they have a value, what is that value), or is the value of these variables undefined (and is the state a partial function $\mathsf{Var} \hookrightarrow \mathsf{Int}$). For simplicity we assume that the value of all variables is initialized to 0. In figure 1 we define the semantics of arithmetic expressions by a function from the expression and a state to an integer value. In this function we use the Scott brackets $[\![\ ]\!]$ to denote a match on syntax using the syntactical categories defined above. This function specifies that the semantics of a number denotation n is the value of that denotation generated by $\mathcal{N}[\![n]\!]$. For brevity we ignore the definition of $\mathcal{N}$. The value of a variable v is obtained by looking it up in the state s. For operators we determine the values of the operands recursively and apply the mathematical operator corresponding to the operator indicated in the syntax.

$$\begin{aligned}
&\mathcal{A} :: \mathsf{A} \rightarrow \mathsf{State} \rightarrow \mathsf{Int} \\
&\mathcal{A}[\![n]\!]\ s &&= \mathcal{N}[\![n]\!] \\
&\mathcal{A}[\![v]\!]\ s &&= s\ v \\
&\mathcal{A}[\![a_1 + a_2]\!]\ s &&= \mathcal{A}\ [\![a_1]\!]\ s + \mathcal{A}\ [\![a_2]\!]\ s \\
&\mathcal{A}[\![a_1 - a_2]\!]\ s &&= \mathcal{A}\ [\![a_1]\!]\ s - \mathcal{A}\ [\![a_2]\!]\ s \\
&\mathcal{A}[\![a_1 * a_2]\!]\ s &&= \mathcal{A}\ [\![a_1]\!]\ s * \mathcal{A}\ [\![a_2]\!]\ s
\end{aligned}$$

Fig. 1. Classical definition of the semantics of arithmetics expressions

The function $\mathcal{A}$ is a mathematical entity that assigns a value to syntax. In [9] the Nielsons already indicate that there is a direct mapping from this mathematical function to a function in a functional programming language. In a modern functional programming language like Clean this can be further improved for instance by user defined infix operators and constructors. The advantages of using a functional language instead of a mathematics are **1)** the compiler of the functional language can be used for statics checks of the specification (e.g. that all identifiers are defined and the specification is correctly typed); **2)** the specification can be executed. The execution of the semantics can for instance be used to validate the definition by interactive simulation, or to test properties of the semantics automatically. Possible drawbacks of using a functional language instead of mathematics are **1)** that there are many constructs in mathematics that cannot be used expressed directly in a functional programming language; **2)** in general it is easier to add ad-hoc notation and new constructs to mathematics than in a functional language; **3)** in mathematics is is very well possible to reason about undefinedness and nontermination, if we are not careful this might cause runtime errors on nonterminating computations in a functional language. In this section we show that the a modern functional language is very suited to express the semantics in a concise way.

As a first example we restate the semantics of arithmetic expressions in Clean. In order to mimic the syntactic match we model the arithmetic expressions by the datatype `AExpr`. This data type directly mimics the syntax of allowed expressions given above. In this datatype we add a dot to the names infix operators (like `+`) that are defined with a different purpose in Clean. The priority of the operators is chosen such that expressions like `Var "x" +. Int 2 *. Var "y"` have the usual binding of operators (here `Var "x" +. (Int 2 *. Var "y")`).

```
:: AExpr
    = Int Int
    | Var Var
    | (+.) infixl 6 IExpr IExpr
    | (-.) infixl 6 IExpr IExpr
    | (*.) infixl 7 IExpr IExpr
:: Var := String
```

The environment used in the semantics is a function from variables (`Var`) to integers.

```
:: Env := Var → Int

emptyEnv :: Env
emptyEnv = λx → 0
```

The equivalent of the mathematical semantic function $\mathcal{A}$ from 1 is the function `A` given in figure 2. Note that these functions have exactly the same length and structure. In the function `A` we are not bound to the naming conventions of $\mathcal{A}$.

In exactly the same way we define a data type for Boolean expressions and the associated semantics in figure 3. This data type directly follows the syntax for B given above. Just like above we added priorities to the infix operators in order to assign the usual binding to Boolean expressions without parentheses.

According to these definitions the semantics of integers and Booleans in While inherits the semantics of `Int` and `Bool` in Clean. If this would be not what is desired we define tailor made data types and operations to mimic the desired semantics.

```
A :: AExpr Env → Int
A (Int i)  env = i
A (Var v)  env = env v
A (x +. y) env = A x env + A y env
A (x -. y) env = A x env - A y env
A (x *. y) env = A x env * A y env
```

Fig. 2. Semantics of arithmetics expressions in Clean

```
:: BExpr
    = TRUE
    | FALSE
    | (=.) infix 4 AExpr AExpr
    | (<.) infix 4 AExpr AExpr
    | ¬. BExpr
    | (&&.) infixr 3 BExpr BExpr

B :: BExpr Env → Bool
B TRUE      env = True
B FALSE     env = False
B (x =. y)  env = A x env == A y env
B (x <. y)  env = A x env ≤ A y env
B (¬. exp)  env = not (B exp env)
B (x &&. y) env = B x env && B y env
```

Fig. 3. Boolean expressions and their semantics

2.3 Denotational Semantics

The goal of denotational semantics is to show the effect of executing a program. In the traditional formulation this is done by a function $\mathcal{DS} :: S \to \mathsf{State} \hookrightarrow \mathsf{State}$ using a syntactic pattern match on statements. The state of a program is just its environment. The effect of an assignment is a change in the state or environment:

$$\mathcal{DS}\,[\![\, v := a \,]\!]\; s \;=\; s\,[\,x \mapsto \mathcal{A}\,[\![\, a \,]\!]\; s\,]$$

The updated state function $s\,[\,x \mapsto i]$ associates the same value to every argument as s except the argument x is mapped to i:

$$(s\,[\,x \mapsto i])\; x = i$$
$$(s\,[\,x \mapsto i])\; y = s\; y,\; \text{if } x \neq y$$

In Clean we define the operator $\mapsto$ (written in plain text as `|->`) to achieve this effect, e.g. `("x" ↦ 1) emptyEnv`.

```
(↦) infix :: Var Int → Env → Env
(↦) v i = λenv x.if (x==v) i (env x)
```

Note that the environment to be changed is here written after the new mapping and we use ordinary parentheses instead of square parentheses.

We use the functional language approach where the statements are represented by a data structure and an associated interpretation function as specified in figure 4.

```
:: Stmt
    = (:=.) infix 2 Var AExpr
    | (:.) infixl 1 Stmt Stmt
    | Skip
    | IF BExpr Stmt Stmt
    | While BExpr Stmt

ds :: Stmt Env → Env
ds (v :=. a) s = (v ↦ A a s) s
ds Skip      s = s
ds (s1 :. s2) s = ds s2 (ds s1 s)
ds (IF c t e)   env
    | B c env
        = ds t env
        = ds e env
ds (While c stmt) env = fix f env
where
    f ff env
        | B c env
            = ff (ds stmt env)
            = env

fix :: (a → a) → a
fix f = f (fix f)
```

Fig. 4. Statements and their denotational semantics

The denotational semantics focusses on the meaning of programs and not on their detailed execution. For this reason the while-statement is not evaluated step by step in the semantics. The semantics simply states that the state produced by the semantics of the while statement (if any) is the fixed point of the given function. Of course there are statements (like `While TRUE Skip`) that have no fixed point. Technically the semantics is a partial function that assigns a meaning, state transformation, to correct terminating statements.

2.4 Natural Semantics

The natural semantics is a big-step semantic that focuses on the effect of the individual language constructs. Since this is a big-step semantics it constructs the final state in one go by applying the semantic function recursively to intermediate results, just like the denotational semantics. For the while-statement it will evaluate the body once if the condition holds and continue with the semantics of the same loop in the new state. For all other statements the formulation of the semantics is identical to the denotational semantics.

```
ns :: Stmt Env → Env
ns (v :=. e)   env = (v ↦ A e env) env
ns (s1 :. s2)  env = ns s2 (ns s1 env)
ns Skip        env = env
ns (IF c t e)  env |   B c env  = ns t env
ns (IF c t e)  env | ¬(B c env) = ns e env
ns (While c s) env |   B c env  = ns (While c s) (ns s env)
ns (While c s) env | ¬(B c env) = env
```

Fig. 5. The natural semantics of statements

In the traditional representation of the operational semantics the semantics is often specified by transition system. This system has two kind of configurations: a tuple $< S, s >$ connecting a statement S and a state s, or a final state s. Transitions are given in the form of axioms. If the premisses above the line and the condition to the right of the line holds, the conclusion below the line holds. For instance the natural semantics for the while-statement if the condition of the statement holds is expressed as:

$$[\mathsf{while}_{\mathsf{TRUE}}] \qquad \frac{< S, s > \rightarrow s_1, \ < \mathsf{while}\ b\ S, s_1 > \rightarrow s_2}{< \mathsf{while}\ b\ S, s > \rightarrow s_2} \ \mathsf{if}\ \mathcal{B}[\![b]\!]\ s$$

The advantage of this approach is that it is not necessary to give an order in the reduction rules, nor to be complete (as in an axiomatic semantics). However, in order to obtain a deterministic semantics we have to prove that the result of semantics is independent of the order of applying these axioms, or we have to show that there is only on order. It is easy to see that the alternatives of the function `ns` all cover different cases and hence these alternatives can be written in any order.

If necessary the natural semantics of While in Clean can be formulated in closer correspondence to the axiom by writing:

```
ns (While b s) env | B b env = env2
where env1 = ns s env; env2 = ns (While b s) env1
```

for this alternative. We prefer the equivalent formulation in figure 5 since it is shorter and shows the difference and similarity with the denotational semantics clearer.

2.5 Structural Operational Semantics

The structural operational semantics is a small-step semantics. It specifies the result of individual reduction steps. Hence the semantics does not always yield the final state, it can also yield an intermediate configuration consisting of a statement and the associated environment:

```
:: Config = Final Env | Inter Stmt Env
```

The structural operations semantics of our example language While is given in figure 6.

```
sosStep :: Stmt Env → Config
sosStep (v :=. e)  s = Final ((v ↦ A e s) s)
sosStep Skip       s = Final s
sosStep (s1 :. s2) s
 = case sosStep s1 s of
    Final s‘ = Inter s2 s‘
    Inter s1‘ s‘ = Inter (s1‘ :. s2) s‘
sosStep (IF c t e) s |   B c s  = Inter t s
sosStep (IF c t e) s | ¬(B c s) = Inter e s
sosStep (While c body) s = Inter (IF c (body :. While c body) Skip) s
```

Fig. 6. The structural operational semantics of statements

By applying this function repeatedly until we reach a Final configuration we obtain a trace of the reduction. For non terminating programs this trace will be an infinite list of configurations.

```
sosTrace :: Config → [Config]
sosTrace c=:(Final _) = [c]
sosTrace c=:(Inter ss s) = [c: sosTrace (sosStep ss s)]
```

Using this trace we can construct a function sos with the same type as the other functions specifying the semantics. This function yields the state in the final state that can be found in the last configuration of the trace.

```
sos :: Stmt Env → Env
sos s env = env1 where (Final env1) = last (sosTrace (Inter s env))
```

There is much more to be said about semantics. For instance the language While can be extended by language constructs like pure functions and procedures. In a similar style we can also define the semantics of functional programming languages. Due to space limitations we will not elaborate on this here.

2.6 Testing Properties of the Semantics

Having the statements available as a data type and semantics available as functions in Clean we can use the fact that the semantics is executable. For instance we can make an editor for statements using iTasks and show the trace of the reduction of such a program using the structural operational semantics defined by `sos`.

Another possibility is to test properties of the semantics using our model based test system G∀st. We show some examples.

Consider the factorial function in While:

```
facStmt
 =  y :=. one :.
    While (one <. vx)
    (
        y :=. vy *. vx :.
        x :=. vx -. one
    )
where
    x = "x"; vx = Var x
    y = "y"; vy = Var y
```

Given any semantics `sem`, the semantics of the factorial function used with an environment that assigns 4 to the variable `"x"`, should produce an environment that associates 24 to the variable `"y"`.

```
propFac :: (Stmt Env → Env) → Bool
propFac sem = sem facStmt (("x" ↦ 4) emptyEnv) "y" == 24
```

The argument of this property is the universal quantified variable

We can test this property for our three versions of the semantics (`ds`, `ns`, and `sos`) by executing:

```
Start = test (propFac For [ds,ns,sos])
```

Fortunately the test system produces Proof as test result which gives us some confidence in the correctness of the various semantics.

More general, the semantics of any statement should be equal for the natural semantics (`ns`) and the denotational semantics (`ds`). In order to guarantee that the test terminates we want only to consider statements that are known to terminate. We ensure this by generating only while-statements that correspond to for-loops with a limited number of iterations. The details will be included in the final version of this paper.

The semantic functions yield the final environment which is a function. In general it is very hard to compare functions. Here it is sufficient to check that the environment produces the same value for all variables that occur in the statement. For this purpose we introduce the function `allvars` that yields all variables that occur on the left-hand side of an assignment in a program in the language While.

```
allvars :: Stmt → [Var]
allvars (x :=. e)   = [x]
allvars (s :. t)     = allvars s + allvars t
allvars Skip         = []
allvars (IF c t e)   = allvars t + allvars e
allvars (While c b)  = allvars b

instance + [x] | Eq x where (+) x y = removeDup (x++y)
```

The function that determines the equivalence of environments is straightforward.

```
eqEnv :: Env Env [Var] → Bool
eqEnv f g vars = and [f v == g v \\ v←vars]
```

After these preparations the property to test equivalent of natural semantics and denotational semantics becomes:

```
prop1 :: (Stmt Env → Env) Stmt → Bool
prop1 stmt = eqEnv (ns stmt emptyEnv) (ds stmt emptyEnv) (allvars stmt)
```

Similar properties can be state for other combinations of the various versions of the semantics. Executing these tests yield Pass, which further increases the confidence in our definitions.

Our next property says that for all semantics and statements `stmt`, the semantics of that statement is equal to the semantics of `Skip :. stmt`.

```
propSkip :: (Stmt Env → Env) Stmt → Bool
propSkip sem stmt
 = eqEnv (sem stmt emptyEnv) (sem (Skip :. stmt) emptyEnv) (allvars stmt)
```

Another property used in the test is that the semantics for all terminating while-statements `While b s` is equivalent to the semantics of `IF b (s :. While b s) Skip`. This property needs some tweaking to generate terminating while statements corresponding to for loops.

```
propWhile :: (Stmt Env → Env) (Maybe GBExpr) Gvar GNat Gstmt → Bool
propWhile sem b v n s
 = eqEnv (sem stmt emptyEnv)
         (sem (decl :. IF cond loop Skip) emptyEnv) (allvars stmt)
where stmt = conv (GWhile b v n s); (decl :. loop=:(While cond body)) = stmt
```

Also this property passes the tests.

This completes our introduction to semantics and testing properties of such a semantics. In the next section we apply these techniques to give a semantics of iTasks.

3 A Semantics for iTasks

In the original iTask system a task is a state transformer of the strict and unique Task State `TSt`. The required uniqueness of the task state (to guarantee single threaded use of the state in a pure functional language) is in Clean indicated by the type annotation `*`. The type parameter `a` indicates the type of the result. This result is returned by the task when it is completely finished.

```
:: Task a :== *TSt → *(a,*TSt)     // an iTask is state transition of type TSt
```

Hence, a `Task` of type `a` is a function that takes a unique task state `TSt` as argument and produces a unique tuple with a value of type `a` and a new unique task state. In this paper we consider only one basic task: the edit task.

```
editTask :: String a → Task a | iData a
```

The function `editTask` takes a string and a value of type `a` as arguments and produces a task `a` under the context restriction that the type `a` is in the type class `iData`. The class `iData` is used to create a web based editor for values of this type. Here we assume that the desired instances are defined.

The `editTask` function creates a task editor to modify a value of the given type, and adds a button with the given name to finish the task. A user can change the value as often as she wants. The task is not finished until the button is pressed. There are predefined editors for all basic data types. For other data types an editor can be derived using Clean's generic programming mechanism [2], or a tailor-made editor can be defined for that type.

In this paper we focus on the following basic iTask combinators to compose tasks.

```
return           :: a                        → Task a     | iData a
(>>=) infixl 1   :: (Task a) (a→Task b)      → Task b     | iData b
(-||-) infixr 3  :: (Task a) (Task a)        → Task a     | iData a
(-&&-) infixr 4  :: (Task a) (Task b)        → Task (a,b) | iData a & iData b
```

The combinators `return` and `>>=` are the usual monadic *return* and *bind*. The return combinator transforms a value in a task yielding that value immediately. The bind combinator is used to indicate a sequence of tasks. The expression `t >>= u` indicates that first task `t` must be done completely. When this is done, its result is given to `u` in order to create a new task that is executed subsequently.

The expression `t -||- u` indicates that both iTasks can be executed in *any* order and *interleaved*, the combined task is completed *as soon as* any subtask is done. The result is the result of the task that completes first, the other task is removed from the system. The expression `t -&&- u` states that both iTasks must be done in any order (interleaved), the combined task is completed when *both* tasks are done. The result is a tuple containing the results of both tasks.

All these combinators are higher order functions manipulating the complex task state `TSt`. This higher order function based approach is excellent for constructing such a library in a flexible and type safe way. However, if we want to construct a program with which we can reason about iTasks, higher order functions are rather inconvenient. In a functional programming language like Haskell or Clean it is not possible to inspect which function is given as argument to a higher order function. The only thing we can do with such a function given as argument is applying it to arguments. In a programming context this is exactly what one wants to do with such a function. In order to specify the semantics of the various iTask combinators however, we need to know which operator we are currently dealing with. This implies that we need to replace the higher order functions by a *representation* that can be handled instead. We replace the higher order functions and the task state `TSt` by the algebraic data type `ITask`. We use infix constructors for the or-combinator, `.||.`, and the and-combinator, `.&&.`, in order to make the representation similar to the corresponding infix combinators `-||-` and `-&&-` from the original `iTask` library.

```
:: ITask
    = EditTask        ID      String  BVal              // an editor
    | .||. infixr 3   ITask   ITask                     // OR-combinator
    | .&&. infixr 4   ITask   ITask                     // AND-combinator
    | Bind            ID      ITask   (Val→ITask)       // sequencing-combinator
    | Return          Val                               // return the value

:: Val  = Pair Val Val   | BVal BVal
:: BVal = String String | Int Int   | VOID
```

Instances of this type `ITask` are called *task trees*. Without loss of generality we assume here that all editors return a value of a basic type (`BVal`). In the real `iTask` system editors can be used with every (user defined) data type. Using only these basic values in the semantics makes it easier to construct a type preserver simulator (see section 6). Since the right-hand side of the sequencing operator `Bind` is a normal function, this model has here the same rich expressibility as the real `iTask` system.

In order to write `ITasks` conveniently we introduce two abbreviations. For the monadic `Bind` operator we define an infix version. This operator takes a task and a function producing a new task as arguments and adds a default `id` to the `Bind` constructor.

```
(⇛) infixl 1 :: ITask (Val→ITask) → ITask
(⇛) t f = Bind id1 t f
```

For convenience we introduce also the notion of a button task. It executes the given `iTask` after the button with the given label is pressed. A button task is composed of a `VOID` editor and a `Bind` operator ignoring the result of this editor.

```
ButtonTask i s t = EditTask i s VOID ⇛ λ_ → t
```

Any executable form of `iTasks` will show only the button of a `VOID` editor. Since the `Type` of the edited value must be preserved, it cannot be changed.

3.1 Task Identification

The task to be executed is composed of elementary subtasks. These subtasks can be changed by events in the generated web-interface, like entering a value in a text-box or pushing a button. In order to link these events to the correct subtask we need an identification mechanism for subtasks. We use an automatic system for the identification of subtasks. Neither the worker, nor the developer of task specification has to worry about these identifications. The fact that the `iTask` system is in principle a multi-user system implies that there are multiple views on the task. Each worker can generate events independently of the other workers. The update of the task tree can generate new subtasks as well as remove subtasks of other workers. This implies that the id's of subtasks must be persistent and that newly generated subtasks cannot reuse old id's. For these reasons the numbering system has to be more advanced than just a numbering of the nodes. The semantics in this paper ignores the multi-user aspect of the semantics, but the numbering system is able to handle this (just as the real iTask system).

Tasks are identified by a list of integers. These task identifications are used similar to the sections in a book. On top level the tasks are assigned integer numbers starting at 0. In contrast to sections, the least significant numbers are on the head of the list rather than on the tail. The data type used to represent these task identifiers, `ID`, is just a list of integers.

```
:: ID = ID [Int]

next :: ID → ID
next (ID [a:x]) = ID [a+1:x]
```

Whenever a task is replaced by its successor the `id` is incremented by the function `next`. For every `id`, `i`, we have that `next i` $\neq$ `i`. In this way we distinguish inputs for a task from inputs to its successor. The function `splitID` generates a list of task identifiers for subtasks of a task with the given id. This function adds two numbers to the identifier, one number for the subtask and one number for the version of this subtask. If we would use the same number for both purposes, one application of the function `next` would incorrectly transform the identification of the current subtask to that of the next subtask.

```
splitID :: ID → [ID]
splitID (ID i) = [ID [0,j:i] \\ j ← [0..]]
```

These identifiers of subtasks are used to relate inputs to the subtasks they belong to. The function `nmbr` is used to assign fresh and unique identifiers to a task tree.

```
nmbr :: ID ITask → ITask
nmbr i (EditTask _ s v) = EditTask i s v
nmbr i (t .||. u)       = nmbr j t .||. nmbr k u where [j,k:_] = splitID i
nmbr i (t .&&. u)       = nmbr j t .&&. nmbr k u where [j,k:_] = splitID i
nmbr i (Bind _ t f)     = Bind k (nmbr j t) f    where [j,k:_] = splitID i
nmbr i t=:(Return _)    = t
```

By convention we start numbering with `id1 = ID [0]` in this paper.

3.2 Events

The inputs for a task are called *events*. This implies that the values of input devices are not considered as values that change in time, as in FRP (Functional Reactive Programming). Instead changing the value of an input device generates an event that is passed as an argument to the event handling function. This function will generate a new state and a new user interface.

An event is either altering the current value of an editor task or pressing the button of such an editor. At every stage of running an iTask application, several editor tasks can be available. Hence many inputs are possible. Each event contains the id of the task to which it belongs as well as additional information about the event, the EventKind.

```
:: Event     = Event ID EventKind | Refresh
:: EventKind = EE BVal | BE
```

The event kind EE (*E*ditor *E*vent) indicates a new basic value for an editor. A *B*utton *E*vent BE signals pressing the button in an editor indicating that the user finished editing.

Apart from these events there is a Refresh event. In the actual system it is generated by each refresh of the user-interface. In the real iTask system this event has two effects: 1) the task is normalized; and 2) an interface corresponding to the normalized task is generated. In the semantics we only care about the normalization effect. *Normalization* of a task is done by applying the Refresh event to the task. Although this event is ignored by all elementary subtasks it has effects on subtasks that can be rewritten without user events. For instance, the task `editTask "ok" 1 -||- return 5` is replaced by `return 5`. Similarly the task `return 7 >>= editTask "ok"` is replaced by `editTask "ok" 7` We elaborate on normalization in the next section.

3.3 Rewriting Tasks given an Event

In this section we define a rewrite semantics for iTasks by defining how a task tree changes if we apply an event to the task. Because we want an executable semantics rewriting is defined by an operator @., pronounced as *apply*. We define a class for @. in order to be able to overload it, for instance with the application of a list of events to a task.

```
class (@.) infixl 9 a b :: a b → a
```

Given a task tree and an event, we can compute the new task tree representing the task after handling the current input. This is handled by the most important instance of the operator @. for ITask and Event listed in figure 7. It is assumed that the task is properly numbered and normalized, and that the edit events have the correct type for the editor.

This semantics shows that the ids play a dominant role in the rewriting of task trees. An event only has an effect on a task with the same id. Edit tasks can react on button events (line 4) as well as edit events (line 3). Line 14 shows why the Bind operator has an id. Events are never addressed to this operator, but the id is used to normalize (and hence number) the new subtask that is dynamically generated by `f v` if the left-hand side task is finished. All events that are not enabled are ignored (line 16). All other constructs pass the events to their subtasks and check if the root of the task tree can be rewritten after the reduction of the subtasks. The recursive call with `@. e` on line 13 can only have

```
instance @. ITask Event
where
  (@.) (EditTask i n v) (Event j (EE w)) | i==j = EditTask (next i) n w
  (@.) (EditTask i n v) (Event j BE)     | i==j = Return (BVal v)
  (@.) (t .||. u) e  = case t @. e of
                          t=:(Return _) = t
                          t = case u @. e of
                                u=:(Return _)  = u
                                u              = t .||. u
  (@.) (t .&&. u) e  = case (t @. e, u @. e) of
                          (Return v, Return w) = Return (Pair v w)
                          (t, u)               = t .&&. u
  (@.) (Bind i t f) e = case t @. e of
                          Return v = normalize i (f v)
                          t        = Bind i t f
  (@.) t e = t
```

Fig. 7. The basic semantics of iTasks.

an effect when the task was not yet normalized, in all other situations applying the event has no effect.

A properly numbered task tree remains correctly numbered after reduction. Editors that receive a new value get a new number by applying the function `next` to the task identification number. The numbering scheme used guarantees that this number cannot occur in any other subtask. If the left left-hand task of the bind-operator is rewritten to a normal form a new task tree is generated by `f v`. The application of `normalize (next i)` to this tree guarantees that this tree is well formed and properly numbered within the surrounding tree. This implies that applying an event repeatedly to a task has at most once an effect.

The handling of events for a task tree is somewhat similar to reduction of combinator systems or in the λ-calculus. An essential difference of such a reduction system with the task trees considered here is that all needed information is available inside a λ-expression. The evaluation of task trees needs the event as additional information.

Event sequences are handled by the following instance of the apply operator:

```
instance @. t [e] | @. t e where (@.) t es = foldl (@.) t es
```

Normalization A task `t` is *normalized* (or *well formed*) iff `t @. Refresh = t`. The idea is that all reductions in the task tree that can be done without a new input should have been done. In addition we require that each task tree considered is properly numbered (using the algorithm `nmbr` in section 3.1). In the definition of the operator `@.` we assume that the task tree given as argument is already normalized. Each task can be normalized and properly numbered by applying the function `normalize1` to that task.

```
normalize :: ID ITask → ITask
normalize i t = nmbr i (t @. Refresh)

normalize1 :: ITask → ITask
normalize1 t = normalize id1 t
```

Enabled Subtasks All editor tasks that are currently part of the task tree are *enabled*, which implies that they can be rewritten if the right events are supplied. The subtasks that are generated by the function on the right-hand side of a `Bind` construct are **not** enabled, even if we can predict exactly what subtasks will be generated. Events accepted by the enabled subtasks are called *enabled events*, this is the set of events that have an effect on the task when it is applied to such an event. Consider the following tasks:

```
t1 = EditTask id1 "b" (Int 1) .&&. EditTask id2 "c" (Int 2)
t2 = EditTask id1 "b" (Int 1) .||. EditTask id2 "c" (Int 2)
t3 = ButtonTask id1 "b" (EditTask id2 "c" (Int 3))
t4 = ButtonTask id1 "b" t4
t5 = EditTask id1 "b" (Int 5) ⇛ λv.ButtonTask id2 "c" (Return (Pair v v))
t6 = EditTask id1 "b" (Int 6) ⇛ λv.t6
t7 v p = EditTask id1 "ok" v  ⇛ λr=:(BVal w).if (p w) (Return r) (t7 w p)
```

In `t1` and `t2` all integer and button events with identifier `id1` and `id2` are enabled. In `t3` and `t4` only the event `Event id1 BE` is enabled. In `t5`, `t6` and `t7` all integer and button events with identifier `id1` are enabled. All other events can only be processed after the button event for the task with `id1` on the left-hand side of the bind operator.

Task `t4` rewrites to itself after a button event. In `t6` the same effect is reached by a bind operator. The automatic numbering system guarantees that the tasks obtain another `id` after applying the enabled button events. Task `t7` is parameterized with a basic value and a predicate on such a value, and terminates only when the worker enters a value satisfying the predicate. This simple example shows that the bind operator is more powerful than just sequencing fixed tasks. In fact any function of type `Val→ITask` can be used there.

Normal Form A task is in *normal form* if it has the form `Return v` for some value `v`. A task in normal form is not changed by applying any event. The function `isNF :: ITask → Bool` checks if a task is in normal form. In general a task tree does not have a unique normal form. The normal form obtained depends on the events applied to that task. For task `t2` above the normal form of `t2 @. Event id1 BE` is `Return (BVal (Int 1))` while `t2 @. Event id2 BE` is `Return (BVal (Int 2))`. The recursive tasks `t4` and `t6` do not have a normal form at all.

Needed Events An event is *needed* in task `t` if the subtask to which the event belongs is enabled and the top node of the task tree `t` cannot be rewritten without that event.

In task `t1` above the events `Event id1 BE` and `Event id2 BE` are needed. Task `t2` has no needed event. This task can evaluate to a normal form by applying either `Event id1 BE` or `Event id2 BE`. As soon as one of these events is applied, the other task disappears. In `t3` only `Event id1 BE` is needed, the event `Event id2 BE` is not enabled. Similarly, in `t4`, `t5` and `t6` (only) the event `Event id1 BE` is needed.

For an edit-task the button-event is needed. Any number of edit-events can be applied to an edit-task, but they are not needed. For the task `t1 .&&. t2` the needed events is the sum of the needed events of `t1` and the needed events of `t2`. For a monadic bind the only needed events are the needed events of the left hand task. The needed events of a task `t` are obtained by `collectNeeded`. To ensure that needed events are collected in a normalized task we apply `normalize1` before scanning the task tree. In the actual `iTask` system the task is normalized by the initial refresh event and needs no new normalization ever after. In the task `t1 .||. t2` non of the events is needed, the task can is finished as soon as the task `t1` or the task `t2` is finished. Normalization is only include here to ensure that the task is normalized in every application of this function.

```
collectNeeded :: ITask → [Event]
collectNeeded t = col (normalize1 t)
where
  col (EditTask id n v) = [Event id BE]
  col (t1 .&&. t2)      = col t1 ++ col t2
  col (Bind id t f)     = col t             // no events from f
  col _                 = []                // Return and the OR-combinator
```

In exactly the same spirit `collectButtons` collects all enabled button events in a task tree, and `collect` yields all enabled button events plus the enabled edit events containing the current value of the editors. The list of events is needed for the simulation of the task discussed in section 6.

An event is *accepted* if it causes a rewrite in the task tree, i.e. the corresponding subtask is enabled. A sequence of events is accepted if each of the events causes a rewrite when the events are applied in the given order. This implies that an accepted sequence of events can contain events that are not needed, or even not enabled in the original tree. In task `t2` the button event with `id1` and `id2` are accepted, also the editor event `Event id1 (EE (Int 42))` is accepted. All these events are enabled, but neither of them is needed. The task `t5` accepts the sequence `[Event id1 BE, Event id2 BE]`. The second event is not enabled in `t5`, but applying `Event id1 BE` to `t5` enables it.

Value The *value* of a task is the value returned by the task if we repeatedly press the left most button in the task until it returns a value. This implies that the value of task `t1` is `Pair (Int 1) (Int 2)`, the value of `t2` is `Int 1` since buttons are pressed from left to right. The value of `t3` is `Int 3` and the value of `t5` is `Pair (Int 5) (Int 5)`. The value of `t4` and `t6` is undefined. Since a task cannot produce a value before all needed events are supplied, we can apply all needed events in one go (there is no need to do this from left to right).

For terminating tasks the value can be computed by inspection of the task tree, there is no need to do the actual rewrite steps as defined by the `@.` operator. For nonterminating tasks the value is undefined, these tasks will never return a value. The class `val` determines the value by inspection of the data structure.

```
class val a :: a → Val

instance val BVal where val v = BVal v
instance val Val  where val v = v
instance val ITask
where
    val (EditTask i n e)    = val e
    val (Return v)          = val v
    val (t .||. u)          = val t    // priority for the left subtask
    val (t .&&. u)          = Pair (val t) (val u)
    val (Bind i t f)        = val (f (val t))
```

The value produced is always equal to the value returned by the task if the user presses all needed buttons and the leftmost button if there is no needed button. The property `pVal` in section 5 states this and testing does not reveal any problems with this property.

The value of a task can change after applying an edit event. For instance the value of task `EditTask id1 "ok" (BVal (Int 2))` is `BVal (Int 2)`. After applying `Event id1 (BVal (Int 7))` to this task the value is changed to `BVal (Int 7)`.

Type Although all values that can be returned by a task are represented by the type `Val`, we occasionally want to distinguish several families of values within this type. This type is not the data type `Val` used in the representation of tasks, but the type that the corresponding tasks in the real `iTask` system would have. We assign the type *Int* to all values of the form `Int i`. All values of the form `String s` have type *String*. If value `v` has type v and value `w` has type w then the value `Pair v w` has type *Pair v w*. The types allowed are:

$$\mathit{Type} = \mathit{Int} \mid \mathit{String} \mid \mathit{VOID} \mid \mathit{Pair}\ \mathit{Type}\ \mathit{Type}$$

To prevent the introduction of yet another data type we represent the types yielded by tasks in this paper as instance of `Val`. The type *Int* is represented by `Int 0` and the type *String* is represented as `String ""`. We define a class `type` to determine types of tasks.

```
:: Type :== Val
class type a :: a → Type
```

Instances of this class for `Val` and `ITask` are identical to the instances of `val` defined in section 3.3. Only the instance for `BVal` is slightly different:

```
instance type BVal
where
    type (Int i)    = BVal (Int 0)
    type (String s) = BVal (String "")
    type VOID       = BVal VOID
```

4 Equivalence of Tasks

Given the semantics of iTasks we can define equivalence of tasks. Informally we want to consider two tasks equivalent if they have the same semantics. Since we can apply infinitely many update events to each task that contains an editor we cannot determine equivalence by applying all possible input sequences. Moreover, tasks containing a bind operator also contain a function and the equivalence of functions is in general undecidable. iTasks are obviously Turing complete and hence equivalence is also for this reason known to be undecidable. It is even possible to use more general notions of equivalence, like tasks are equivalent if they can be used to do the same job. Hence, developing a useful notion of equivalence for tasks is nontrivial.

In this paper we will develop a rather strict notion of equivalence of tasks: tasks t and u are equivalent if they have an equal value after all possible sequences of events and at each intermediate state the same events are enabled. Since the identifications of events are invisible for the workers using the iTask system, we allow that the lists of events applied to t and u differ in the event identifications. The strings that label the buttons in t and u do not occur in the events, hence it is allowed that these labels are different for equivalent tasks.

First we introduce the notion of *simulation*. Informally a task u can simulate a task t if a worker can do everything with u that can be done with t. It is very well possible that a worker can do more with u than with t. The notation $t \preccurlyeq u$ denotes that u can simulate t. Technically we require that: **1)** for each sequence of accepted events of t there is a corresponding sequence of events accepted by u; **2)** the values of the tasks after applying these events is equal; and **3)** after applying the events, all enabled events of t have a matching event in u. Two events are equivalent, $e_1 \cong e_2$, if they differ at most in their identification.

$$\begin{aligned} t \preccurlyeq u \equiv\; & \forall\, i \in \text{accept}(t).\exists j \in \text{accept}(u). i \cong j \wedge \mathit{val}(t\,@.\,i) = \mathit{val}(u\,@.\,j) \\ & \wedge \mathit{collect}(t\,@.\,i) \subseteq \mathit{collect}(u\,@.\,j) \end{aligned}$$

The notion $t \preccurlyeq u$ is not symmetrical, it is very well possible that u can do much more than t. As an example we have that for all tasks t and u that are not in normal form $t \preccurlyeq t\,.||.\,u$, and $t \preccurlyeq u\,.||.\,t$. If one of the tasks is in normal form it has shape `Return v`, after normalization the task tree $u\,.||.\,t$ will have the value `Return v` too. Any task can simulate itself $t \preccurlyeq t$, and an edit task of any basic value `v` can simulate a button task that returns that value: `ButtonTask id1 "b" (Return (BVal v))` $\preccurlyeq$ `EditTask id2 "ok" v`. In general we have $t\,.||.\,t \not\preccurlyeq t$: for instance if t is an edit task, in $t\,.||.\,t$ we can put a new value in one of the editors and produce the original result by pressing the `ok` button in the other editor, the task t cannot simulate this. The third requirement in the definition above is included to ensure that $t\,.||.\,t \not\preccurlyeq t$ also holds for tasks with only one button `ButtonTask id1 "b1" (BVal (Int 36))`.

Two tasks t and u are considered to be *equivalent* iff t simulates u and u simulates t.

$$t \cong u \;\equiv\; t \preccurlyeq u \;\wedge\; u \preccurlyeq t$$

This notion of equivalence is weaker then the usual definition of bisimulation [13] since we do not require equality of events, but just equivalency. Two editors containing a different value are not equivalent. There exist infinitely many event sequences such that these editors produce the same value. But for the input sequence consisting only of the button event, they produce a different value.

Since each task can simulate itself ($t \preccurlyeq t$), any task is equivalent to itself: $t \cong t$. If t and u are tasks that are not in normal form we have $t\,.||.\,u \cong u\,.||.\,t$. Consider the following tasks:

```
u1 = ButtonTask id1 "b1" (Return (BVal (Int 1)))
u2 = EditTask id2 "b2" (Int 1)
u3 = EditTask id2 "b3" (Int 2)
u4 = EditTask id2 "b4" (String "Hi")
u5 = u1 .||. u2
u6 = u2 .||. u1
u7 = u2 .&&. u4
u8 = u4 .&&. u2
u9 = u2 ⇛ λv.Return (BVal (Int 1))
u10 = u2 ⇛ λx.u4 ⇛ λy.Return (Pair x y)
```

The trivial relations between these tasks are $u_i \preccurlyeq u_i$ and $u_i \cong u_i$ for all u_i. The nontrivial relations between these tasks are: `u1` $\preccurlyeq$ `u2`, `u1` $\preccurlyeq$ `u5`, `u1` $\preccurlyeq$ `u6`, `u1` $\preccurlyeq$ `u9`, `u2` $\preccurlyeq$ `u5`, `u2` $\preccurlyeq$ `u6`, `u5` $\preccurlyeq$ `u6`, `u6` $\preccurlyeq$ `u5`, `u10` $\preccurlyeq$ `u7`, `u10` $\preccurlyeq$ `u8`, and `u2` $\cong$ `u9`, `u5` $\cong$ `u6`. Note that `u7` $\ncong$ `u8` since the tasks yield another value.

Due to the presence of functions in the task expressions it is in general undecidable if one task simulates another or if they are equivalent. However, in many situations we can decide these relations between tasks by inspection of the task trees that determine the behavior of the tasks.

4.1 Determining the Equivalence of Task Trees

The equivalence of tasks requires an equal result for all possible sequences of accepted events. Even for a simple integer edit task there are infinitely many sequences of events. This implies that checking equivalence of tasks by applying all possible sequences of events is in general impossible.

In this section we introduce two algorithms to approximate the equivalence of tasks. The first algorithm, section 4.2, is rather straightforward and uses only the enabled events of a task tree and the application of some of these events to approximate equivalence. The second algorithm, section 4.3 is somewhat more advanced and uses the structure of the task trees to determine equivalence whenever possible.

We will use a four valued logic as for the result:

```
:: Result = Proof | Pass | CE | Undef
```

The result `Proof` corresponds to `True` and indicates that the relation is known to hold. The result `CE` (for *C*ounter*E*xample) is equivalent to `False`, the relation does not hold. The result `Pass` indicates that functions are encountered during the scanning of the trees. For the values tried the properties holds. The property might hold for all other values, but it is also possible that there exists inputs to the tasks such that the property does not hold. The value `Undef` is used as result of an existential quantified property ($\exists\, w. P\ x$) where no proof is found in the given number of test cases; the value of this property is undefined [5]. This type `Result` is a subset of the possible test results handled by the test system `G∀st`. For these results we define disjunction ('or', ⩔), conjunction ('and', ⩓), and negation ('not', ¬) with the usual binding power and associativity. In addition we define the type conversion from Boolean to results and the weakening of a result which turns `Proof` in `Pass` and leaves the other values unchanged.

```
class (⩔) infixr 2 a b :: a b → Result     // a OR b
class (⩓) infixr 3 a b :: a b → Result     // a AND b

instance ¬ Result                           // negation

toResult :: Bool → Result                   // type conversion
toResult b = if b Proof CE

pass :: Result → Result                     // weakens result to at most Pass
pass r = r ⩓ Pass
```

For ⩔ and ⩓ we define instances for all combinations of `Bool` and `Result` as a straightforward extension of the corresponding operation on Booleans.

4.2 Determining Equivalence by Applying Events

In order to compare `ITasks` we first ensure that they are normalized and supply an integer argument to indicate the maximum number of reduction steps. The value of this argument `N` is usually not very critical. In our tests 100 and 1000 steps usually gives identical (and correct) results. The function `equivalent` first checks if the tasks are returning currently the same value. If both tasks need inputs we first check **1)** if the tasks have the same type, **2)** if the tasks currently offer the same number of buttons to the worker, **3)** if the tasks have the same number of needed buttons, and **4)** if the tasks offer equivalent editors. Whenever either of these conditions does not hold the tasks `t` and `u` cannot be equivalent. When these conditions hold we check equivalence recursively after applying events. If there are needed events we apply them all in one go, without these events the tasks cannot produce a normal form. If the tasks have no needed events we apply all combinations of button events and check if one of these combinations makes the tasks equivalent. We need to apply all combinations of events since all button events are equivalent. All needed events can be applied in one go since they are needed in order to reach a normal form and the order of applying needed events is always irrelevant. If there are edit tasks enabled, `length et>0`, in the task the result is at most `Pass`. This is achieved by applying the functions `pass` or `id`.

```
equivOper :: ITask ITask → Result
equivOper t u = equivalent N (normalize1 t) (normalize1 u)
```

```
equivalent :: Int ITask ITask → Result
equivalent n (Return v) (Return w) = v == w
equivalent n (Return v) _ = CE
equivalent n _ (Return w) = CE
equivalent n t u
 | n≤0
    = Pass
    = if (length et>0) pass id
      (type t == type u ∧ lbt == lbu  ∧ lnt == lnu ∧ sort et == sort eu
      ∧ if (lnt>0)
            (equivalent (n-lnt) (t @. nt) (u @. nu))
            (exists N [equivalent n (t @. i) (u @. j)\\(i,j)←diag2 bt bu]))
where
    bt  = collectButtons t; nt  = collectNeeded t
    bu  = collectButtons u; nu  = collectNeeded u
    et  = collectEdit t;    eu  = collectEdit u
    lnt = length nt; lnu = length nu; lbt = length bt; lbu = length bu
```

The function `exists` checks if one of the first `N` values is `Pass` or `Proof`.

```
exists :: Int [Result] → Result
exists n []    = CE
exists 0 l     = Undef
exists n [a:x] = a ∨ exists (n-1) x
```

In this approach we do not apply any edit events. It is easy to design examples of tasks where the current approximation yields `Pass`, but applying some edit events reveals that the tasks are actually not equivalent (e.g. `t = EditTask id1 (BVal (Int 5))` and `t ⇛ Return (BVal (Int 5)))`. We obtain a better approximation of the equivalence relation by including some edit events in the function `equivalent`. Due to space limitations and to keep the presentation as simple as possible we have not done this here.

4.3 Determining Equivalence of Tasks by Comparing Task Trees

Since the shape of the task tree determines the behavior of the task corresponding to that task tree, it is tempting to try to determine properties like $t \preccurlyeq u$ and $t \cong u$ by comparing the shapes of the trees for u and t. For most constructs in the trees this works very well. For instance it is much easier to look at the structure of the tasks `EditTask id1 "ok" (BVal (Int 5))` and `EditTask id2 "done" (BVal (Int 5))` to see that they are equivalent, than approximating equivalence of these tasks by applying events to these tasks and comparing the returned values. In this section we use the comparison of task trees to determine equivalence of tasks. The function `eqStruct` implements this algorithm.

There are a number of constructions that allow different task trees for equivalent tasks. These constructs require special attention in the structural comparison of task trees:

1. The tasks `ButtonTask id1 "b" (Return v) .&&. Return w` and `ButtonTask id1 "b" (Return (Pair v w))` are equivalent for all basic values `v` and `w`. This kind

   of equivalent tasks with a different task tree can only occur if one of the branches of `.&&.` is in normal form and the other is not. On lines 9, 16 and 17 of the function `eqStruct` there are special cases handling this. The problem is handled by switching to a comparison by applying events, very similar to the `equivalent` algorithm in the previous section. The function `equ` takes care of applying events and further comparison.
2. The choice operator `.||.` should be commutative, (`t.||.u ≃ u.||.t`), and associative (`(t.||.u).||.v ≃ t.||.(u.||.v)`). In order to guarantee this, `eqStruct` collects all adjacent or-tasks in a list and checks if there is a unique mapping between the elements of those list such that the corresponding subtasks are equivalent (using `eqStruct` recursively). The implementation of the auxiliary functions is straightforward.
3. The `Bind` construct contains real functions, hence there are many ways to construct equivalent tasks with a different structure. For instance, we have that any task `t` is equivalent to the task `t ⇛ Return`, or slightly more advanced: `s.&&.t` is equivalent `(t .&&. s) ⇛ λ(Pair x y)→Return (Pair y x)` for all tasks `s` and `t`.
   The function `eqStruct` checks if the left-hand sides and the obtained right-hand sides of two bind operators are equivalent. If they are not equivalent the tasks are checked for equivalence by applying inputs, see line 13-15.

The `eqStruct` algorithm expects normalized task trees. The operator ≃ takes care of this normalisation.

```
class (≃) infix 4 a :: a a → Result       // is arg1 equivalent to arg2?

instance ≃ ITask where (≃) t u = eqStruc N (normalize1 t) (normalize1 u)
```

If the structures are not equal, but the task might be event equal we switch to applying inputs using the function `equ`. This function is very similar to the function `equivalent` in the previous section. The main difference is that the function `equ` always switches to `eqStruct` instead of using a recursive call. If a structural comparison is not possible after applying an event, the function `eqStruct` will switch to `equ` again.

```
eqStruc :: Int ITask ITask → Result                                           1
eqStruc 0 t u = Pass                                                          2
eqStruc n (Return v)       (Return w)       = v ≃ w                           3
eqStruc n (Return v)       _                = CE                              4
eqStruc n _                (Return w)       = CE                              5
eqStruc n (EditTask _ _ e) (EditTask _ _ f) = e≃f                             6
eqStruc n s=:(a .&&. b)    t=:(x .&&. y)                                      7
 = eqStruc (n-1) a x ∧ eqStruc (n-1) b y ∨                                    8
   ((inNF a || inNF b || inNF x || inNF y) ∧ equ n s t)                       9
eqStruc n s=:(a .||. b)    t=:(x .||. y)                                      10
 = eqORn n (collectOR s) (collectOR t)                                        11
eqStruc n s=:(Bind i a f) t=:(Bind j b g)                                     12
 = eqStruc (n-1) a b ∧ eqStruc (n-2) (f (val a)) (g (val b)) ∨ equ n s t      13
eqStruc n s=:(Bind _ _ _) t                 = equ n s t                       14
eqStruc n s              t=:(Bind _ _ _) = equ n s t
eqStruc n s=:(a .&&. b)  t               = (inNF a||inNF b) ∧ equ n s t
eqStruc n s              t=:(x .&&. y)   = (inNF x||inNF y) ∧ equ n s t
eqStruc n s              t               = CE
```

This uses instances of $\simeq$ for basic values (`BVal`) and values (`Val`). For these instances no approximations are needed. The line 10 and 11 implements the commutativity of the operator `.||.`: `collectOR` produces a list of all subtasks glued together with this operator, and `eqORn` determines if these lists of subtasks are equivalent in some permutation. The definitions are a direct generalization of the ordinary equality $=$.

A similar approach can be used to approximate the simulation relation $\preccurlyeq$.

Property `pEquiv` in the next section states that both notions of equivalence yield equivalent results, even if we include edit events. Executing the associated tests indicate no problems with this property. This test result increases the confidence in the correct implementation of the operator $\simeq$. Since $\simeq$ uses the structure of the tasks whenever possible, it is more efficient than `equivOper` that applies events until the tasks are in normal form. The efficiency gain is completely determined by the size and contents of the task tree, but can be significant. It is easy to construct examples with an efficiency gain of one order of magnitude or more.

5 Testing Properties of iTasks

Above we mentioned a number of properties of iTasks and their equivalency like $\forall\, s, t \in$ iTask . `s.||.t`$\simeq$`t.||.s`. Although we designed the system such that these properties should hold, it is good to verify that the properties do hold indeed. Especially during the development of the system many versions are created in order to find a concise formulation of the semantics and an effective check for equivalence.

Creating formal proofs for all properties for all those versions of the semantics during its development is completely infeasible. Assuming that all well-typed versions of the semantics are correct is much to optimistic. We used the automatic test system G∀st to check the semantic functions presented here with a set of desirable properties. For instance the above property can be stated in G∀st as:

```
pOr :: GITask GITask → Property
pOr x y = normalize1 (t.||.u) ≃ normalize1 (u.||.t)
where t = toITask x; u = toITaskT (type t) y
```

The arguments of such a property are treated by the test system as universal quantified variables over the given types. The test system generates test values of these types using the generic class `ggen`. Since some `ITask` constructs contain a function, we use an additional data type, `GITask`, to generate the desired instances. We follow exactly the approach as outlined in [4]. The type `GITask` contains cases corresponding to the constructors in `ITask`, for button tasks, for tasks of the form `t` $\Rrightarrow$ `Return`, and for some simple recursive terminating tasks.

For `pOr` we need to make sure the tasks `t` and `u` have the same type since we combine them with an or-operator. The conversion by `toITask` from the additional type `GITasks` used for the generation to `ITasks` takes care of that.

After executing 23 tests G∀st produces the first counterexample that shows that this property does not hold for `t = Return (BVal (Int 0))` and `u = Return (Pair (BVal (Int 0)) (BVal (Int 0)))`. Using the semantics from figure 7 it is clear that G∀st is right, our property is too general. A correct property imposes the condition that `t` and `u` are not in normal form:

```
pOr2 x y = notNF [t,u] ==> normalize1 (t.||.u) ≃ normalize1 (u.||.t)
where t = toITask x; u = toITaskT (type t) y
```

In the same way we can show that `t.||.t ≄ t` for tasks that are not in normal form (`p2`) and test the associativity of the `.||.` or operator (`p3`).

```
p2 :: GITask GITask → Property
p2 x y = notNF [s,t] ==> (s.||.t)≄t
where s = toITask x; t = toITaskT (type s) y

p3 :: GITask GITask GITask → Property
p3 x y z = (s .||. (t .||. u))≃((s .||. t) .||. u)
where s = toITask x; t = toITaskT (type s) y; u = toITaskT (type s) z
```

In total we have defined over 70 properties to test the consistency of the definitions given in this paper. We list some representative properties here. The first property states that needed events can be applied in any order. Since there are no type restrictions on the type `t` we can quantify over `ITask`s directly.

```
pNeeded :: ITask → Property
pNeeded t = (λj. t @. i ≃ t @. j) For perms i where i = collectNeeded t
```

In this test the fragment `For perms i` indicates an additional quantification over all `j` in `perms i`. The function `perms :: [x] → [[x]]` generates all permutations of the given list. In logic this property would have been written as $\forall t \in \mathsf{ITask}$, $\forall j \in \mathsf{perms}\ (\mathsf{collectNeeded}\ t).\ t\ @.\ (\mathsf{collectNeeded}\ t) \simeq t\ @.\ j$.

The next property states that both approximations of equivalence discussed in the previous section produce equivalent results.

```
pEquiv :: ITask ITask → Property
pEquiv t u = (equivOper t u) ≃ (t≃u)
```

The type of a task should be preserved under reduction. In the property `pType` also events that are not well typed will be tested. Since we assume that all events are well typed (the edit events have the same type as the edit task they belong to), it is better to use `pType2` where the events are derived from the task `t`.

```
pType :: ITask → Property
pType t = (λi.type t == type (t @. i)) For collect t

pType2 :: ITask → Property
pType2 t = pType t For collect t
```

The phrase `For collect t` indicates that for testing these properties the events are collect from the task tree rather then generated systematically by G∀st. However the tasks to be used in the test are generated systematically by G∀st.

The property `pVal` states that the value of a task obtained by the optimized function `val` is equal to the value of the task obtained by applying events obtained by `collectVal` until it returns a value. The function `collectVal` returns all needed events and the leftmost events if these are no needed events.

```
pVal :: ITask → Property
pVal t = val t == nf t
where
    nf (Return v) = v
    nf t = nf (t @. collectVal t)
```

The definitions presented in this paper pass all stated properties. On a normal laptop (Intel core2 Duo (using only one of the cores), 1.8 GHz) it takes about 7 seconds to check all defined properties with 1000 test cases for each property. This is orders of magnitude faster and more reliable then human inspection, which is on its turn much faster than a formal proof (even if it is supported by a state of the art tool). Most of these properties are very general properties, like the properties shown here. Some properties however check specific test cases that are known to be tricky, or revealed problems in the past. If there are problems with one of the properties, they are usually spotted within the first 50 test cases generated. It appears to be extremely hard to introduce flaws in the system that are not revealed by executing these tests. For instance omitting one of the special cases in the function `eqStruct` is spotted quickly. Hence testing the consistency of the system in this way is an effective and efficient way to improve the confidence in its consistency.

6 Discussion

In this paper we give a rewrite semantics for iTasks. Such a semantics is necessary to reason about iTasks and their properties, it is also well suited to explain their behavior. In addition we defined useful notions about iTasks and stated properties related to them. The most important notion is the *equivalence* of tasks.

Usually the semantics of workflow systems is based on Petri nets, abstract state machines, or actor-oriented directed graphs. Since the iTask system allows arbitrary functions to generate the continuation in a sequence of tasks (the monadic bind operator), such an approach is not flexible enough. To cope with the rich possibilities of iTasks our semantics incorporates also a function to determine the continuation of the task after a `Bind` operator.

We use the functional programming language Clean as carrier for the semantical definitions. The tasks are represented by a data structure. The effect of supplying an input to such a task is given by an operator modifying the task tree. Since we have the tasks available as data structure we can easily extract information from the task, like the events needed or accepted by the task. A typical case of the operator `@.` (apply) that specifies the semantics is:

```
(@.) (EditTask i n e) (Event j BE) | i==j = Return (BVal e)
```

In the more traditional Scott Brackets style this alternative is written as:

$$\mathcal{A}\ [\![\ EditTask\ i\ n\ e\ ]\!]\ (Event\ j\ BE) = Return\ (BVal\ e),\ \text{if}\ i = j$$

Our representation has the same level of abstraction and has as advantages that it can be checked by the type system and executed (and hence simulated and tested).

Having the task as a data structure it is easy to create an editor and simulator for tasks using the iTask library. Editing and simulating tasks is helpful to validate the semantics. Although simulating iTasks provides a way to interpret the given task, the executable semantics is not intended as an interpreter for iTasks. In an interpreter we would have focused on a nice interface and efficiency, the semantics focusses on clearness and simplicity.

Compared with the real iTask system there are a number of important simplifications in our `ITask` representation. **1)** Instead of arbitrary types, the `ITasks` can only yield elements of type `Val`. The type system of the host language is not able to prevent type errors within the `ITasks`. For instance it is possible to combine a task that yields an integer, `BVal (Int i)`, with a task yielding a string, `BVal (String s)`, using an `.||.` operator. In ordinary iTasks it is type technically not possible (and semantically not desirable) to combine tasks of type `Task Int` with `Task String` using a `-||-` operator. Probably GADTs would have helped us to enforce this condition in our semantical representation. **2)** The application of a task to an event does not yield an HTML-page that can be used as GUI for the iTask system. In fact there is no notion at all of HTML output in the `ITask` system. **3)** There is no way to access files or databases in the `ITask` system. **4)** There is no notion of workers and assigning subtasks to them. **5)** There is no difference between client site and server site evaluation of tasks. **6)** There is only one workflow process which is implicit. In the real iTask system additional processes can be created dynamically. **7)** The exception handling from the real iTask system is missing in this semantics.

Adding these aspects would make the semantics more complicated. We have deliberately chosen to define a concise system that is as clear as possible.

Using the model-based test system it is possible to test the stated properties fully automatically. We maintain a collection of over 70 properties and test them with one push of a button. Within seconds we do known if the current version of the system obeys all properties stated. This is extremely useful during the development and changes of the system. Although the defined notions of equivalence are in general undecidable, the given approximation works very well in practice. Issues in the semantics or properties are found very quickly (usually within the first 100 test cases). We attempted to insert deliberately small errors in the semantics that are not detected by the automatic tests, but we failed miserably.

In the near future we want to test with G∀st if the real iTask system obeys the semantics given in this paper. In addition we want to extend the semantics in order to cover some of the important notions omitted in the current semantics, for instance task execution in a multi-user workflow system. When we are convinced about the quality and suitability of the extended system we plan to prove some of the tested properties. Although proving properties gives more confidence in the correctness, it is much more work then testing. Testing with a large number of properties has shown to be an extremely powerful way to reveal inconsistencies in the system.

References

1. P. Achten, M. van Eekelen, M. de Mol, and R. Plasmeijer. An Arrow based semantics for interactive applications. In M. Morazán, editor, *Preliminary Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07*, New York, NY, USA, 2-4, Apr. 2007.
2. A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Selected Papers of the 13th International Symposium on the Implementation of Functional Languages, IFL'01*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, Sept. 2002.
3. M. de Mol, M. van Eekelen, and R. Plasmeijer. The mathematical foundation of the proof assistant Sparkle. Technical Report ICIS-R07025, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, Nov. 2007.
4. P. Koopman and R. Plasmeijer. Automatic testing of higher order functions. In N. Kobayashi, editor, *Proceedings of the 4th Asian Symposium on Programming Languages and Systems, APLAS'06*, volume 4279 of *LNCS*, pages 148–164, Sydney, Australia, 8-10, Nov. 2006. Springer-Verlag.
5. P. Koopman and R. Plasmeijer. *Fully automatic testing with functions as specifications*, volume 4164 of *LNCS*, pages 35–61. Springer-Verlag, Budapest, Hungary, 4-16, July 2006.
6. P. Koopman, R. Plasmeijer, and P. Achten. An executable and testable semantics for iTasks. In S.-B. Scholz, editor, *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, pages 53–64, Hertfordshire, UK, 10-12, Sept. 2008. University of Hertfordshire.
7. S.-Y. Lee, Y.-H. Lee, J.-G. Kim, and D. C. Lee. Workflow system modeling in the mobile healthcare B2B using semantic information. In *Proceedings of the 5th'05*, pages 762–770. LNCS, 2005.
8. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18:2006, 2005.
9. H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., 1992.
10. R. Plasmeijer and P. Achten. iData for the world wide web - Programming interconnected web forms. In *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS'06*, volume 3945 of *LNCS*, Fuji Susone, Japan, 24-26, Apr. 2006.
11. R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP'07*, pages 141–152, Freiburg, Germany, 1-3, Oct. 2007. ACM Press.

12. N. Russell, A. ter Hofstede, and W. van der Aalst. newYAWL: specifying a workflow reference language using coloured Petri nets. In *Proceedings of the 8th'07*, 2007.
13. C. Stirling. The joys of bisimulation. In *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 142–151, London, UK, 1998. Springer-Verlag.
14. T. C. D. Team. *The Coq proof assistant reference manual (version 7.0)*, 1998. `http://pauillac.inria.fr/coq/doc/main.html`.