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. This paper has two contributions. First, it gives a semantics for the iTask workflow management system. Second, it describes an effective methodology to construct such a semantics.

Semantics is a formal description of the meaning of language constructs. Just like any other formal description there are umpteen ways of introducing flaws in such a description. Even trained people are not very effective in spotting issues in formal text. In this paper we show that it is very well possible to describe semantics of programming languages using a modern functional programming as carrier of the specification. This enables automatic sanity checks by the language compiler, simulation of the described semantics to validate the specification, and automatic testing of properties of the semantics.

We illustrate this technique with the well-known example of simple imperative language as well as the iTask workflow management system. In our experience this methodology works very well. The combination of sanity checks, simulation and automatic testing of properties really helped to construct a trustworthy semantics for the iTask system.

1 Introduction

The semantics of a system is a formal description of the meaning of that system. In these notes we define the semantics of programming language constructs like expressions, assignments, conditionals and loops. The definition of all language constructs and the way to combine them specifies the meaning of programs in that language. Although the formal semantics is usually a much simpler system than the system described, the semantics itself can be quite complicated as well. Since the semantics is a piece of formal language, the semantics itself is error prone just like any other large formal system (e.g. a computer program, or a mathematical proof). 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 or type 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 semantics, e.g. the semantics of multiplication accidentally specifies addition. In these notes we show an effective way to tackle these problems and illustrate our method by a well-known as well as 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 these notes 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.

The semantics is expressed in the functional programming language Clean instead of the more common Scott Brackets, denotational semantics [22], or horizontal bar style, structural operational semantics à la Plotkin [18]. The close correspondence between semantics and functional programs goes back at least to [15]. 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 [8] it is possible to express properties about the semantics and equivalence of language constructs/tasks concisely, and to test these properties fully automatically. Although the semantics is executable, it is not a serious implementation of the described language itself. The semantics assigns a meaning to expressions represented by some data structure, a real implementation generally provides a parser and type checker to map textual representations to such a data structure. The iTask 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. Using a proof assistant like COQ [23] or SPARKLE [14] for this purpose requires 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.

We will illustrate our approach with two case studies. The first one is the semantics of the very simple imperative language While to explain the concepts. This language, or a very similar language, can be found in many text books about semantics. We compare our formulation of semantics with standard ways to specify the semantics. The second case study is the iTask WFMS (WorkFlow Management System). A WFMS is a system that supports users and machines to execute tasks. The iTask systems consists of a language to describe the task to be executed, as well as the tools to administrate and guide the execution of these tasks via a web based user interface. The iTask example is a good test case since it is still under development and hence requires agile validation. It is a complex system, hence its properties are not cut and clear, and they require exploration.

In this paper we provide a basic rewrite semantics for iTasks as well as a number of useful contributions to reason about tasks, such as *needed events* and *equivalences* of tasks. As usual we omit many details in the semantics to express the meaning of the basic iTask combinators as clearly as possible.

Section 2 gives a short introduction to the various kind of semantics that exist and shows how this can be expressed concisely in a functional language. In section 3 we show how we model iTasks. 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. 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, we discuss related work in section 6 and there is a discussion and a conclusion in section 7.

2 Formal Semantics

The semantics [15] of a programming language defines the meaning of programs. In order to reason about the semantics we 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 built 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.

These approaches to specify the semantics have in common that they pattern match on the syntactic constructs of the language. A meaning is assigned to each of the alternatives in the syntax. For this reason we will represent the language by a data structure that has one constructor for each alternative in the syntax.

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. Similar languages can be found in most textbooks about formal semantics. This language While and its semantics is taken from [15]. We 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 know the syntactical details of numbers and variables. Such details are usually omitted in the semantics. The syntax for the other constructs is:

$$\begin{aligned}
a &::= v \mid n \mid a+a \mid a-a \mid a*a \\
b &::= \texttt{TRUE} \mid \texttt{FALSE} \mid a=a \mid a<a \mid \neg b \mid b \mathbin{\&\&} b \\
S &::= x \texttt{ := } a \mid \texttt{skip} \mid S1\,;\,S2 \mid \texttt{if}\ b\ S1\ \texttt{else}\ S2 \mid \texttt{while}\ b\ S
\end{aligned}$$

The arithmetic expressions do not contain a division operator, /, since that would involve exceptions in the semantics. Division by zero is undefined, hence it cannot yield an ordinary number. It is perfectly possible to handle this in the semantical description of a language, but it makes things more complex as wanted in these notes.

2.2 The Semantics of Expressions in While

In order to specify the semantics we need an *environment*, also called *state*[1], that relates variables and their values. Usually this environment is modeled as a function from variables, modeled by the type Var, to their values, modeled by the type Int: State : Var $\rightarrow$ Int, i.e. a State is a function from variables to integers. 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 if they have a value, what is that value), or is the value of these variables undefined (and is the state a partial function Var $\hookrightarrow$ Int). For simplicity we assume that the state is a total function and that the value of all variables are initialized to 0.

In order to compare the classical mathematical way to define semantics with our functional programming based definitions, we repeat some of the classical definitions. In figure 1 we define the semantics of arithmetic expressions by a function from the expression and a state to an integer value. This definition is very similar to [15], the only difference is the resulting type of expressions. In the version of the Nielsons the result is a natural number, N, while our version produces its representation in computers, Int. Problems that are caused by a finite representation are ignored for the moment. It is perfectly possible to tackle these issues, but that complicates the definition.

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 ignored the syntactic details of numbers n, hence we also omit the definition of their semantics $\mathcal{N}$. 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. Note that the operators $+$, $-$ and $*$ occurring on the left in these equations are just syntax, while their counterparts on the right are the corresponding operations on values.

The function $\mathcal{A}$ is a mathematical entity that assigns a value to syntax. In [15] 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

[1] While we follow the approach of the Nielson's we will call the relation between variables and their values state, just as they do. In our own work we call this function environment.

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

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

functional language can be used for static sanity 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)** there are constructs in mathematics that cannot be 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 it is very well possible to reason about undefinedness and nontermination, if we are not careful this might cause run-time errors on nonterminating computations in a functional language. In this section we show that 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 instead of the mathematical formulation above. 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 append a dot to the names of 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")`). This detail is left unspecified or implicit above. If we do not want to specify those details we can simply label the infix constructors as **infix**.

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

If desired we can even reuse the the ordinary Cleaninfix symbols in right-hand sides of definitions by defining appropriate instances of the infix symbols.

```
instance - AExpr where (-) a b = a -. b
```

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

```
:: Env :== Var → Int

emptyEnv :: Env
emptyEnv = λx → 0
```

The equivalent of the mathematical semantic function $\mathcal{A}$ from figure 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 $\mathcal{A}$. Hence we can use `x` and `y` instead of a_1 and a_2. The various elements of the language are determined by the type they obtain from the context in which they are used instead of the naming conventions.

```
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 arithmetic expressions in Clean

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.

```
:: 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

According to these definitions the semantics of integers and Booleans in While inherits the semantics of `Int` and `Bool` in Clean. If this is not desired we can define tailor made data types and operations to describe the desired 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 $\mathcal{DS} :: S \rightarrow \mathsf{State} \hookrightarrow \mathsf{State}$ 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:

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

This reads that the same environment is returned that is received as argument, but the binding of variable v is mapped to the value of the arithmetic expression a. Note that v and a are meta variables, they represent all concrete variable and expressions that can occur in the language While.

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)
```

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. Again this is a direct transcription of the usual mathematical formulation of the denotational semantics using Scott brackets.

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

ds :: Stmt Env → Env
ds (v :=. a)  env = (v ↦ A a env) env
ds Skip       env = env
ds (s1 :. s2) env = ds s2 (ds s1 env)
ds (IF c t e) env = if (B c env) (ds t env) (ds e env)
ds (While c stmt) env
 = fix (λf env2 . if (B c env2) (f (ds stmt env2)) env2) 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.

Example. As an example we show how the famous Euclid algorithm to compute greatest common divisors in the language While can be expressed as a data structure of type `Stmt`. This algorithm expects the arguments in the variables a and b and leaves the result in the variable b. Since the language While has no functions nor print statement this is about the best we can do.

```
gcdStmt
 =  IF (va =. zero)
       (c :=. vb)
       (While (¬. (vb =. zero))
           (IF (vb <. va)
               (a :=. va -. vb)
               (b :=. vb -. va)
           ) :.
        c :=. va
       )
 where
    a = "a"; va = Var a
    b = "b"; vb = Var b
    c = "c"; vc = Var c
```

We can use the executability of the semantics to compute for instance the greatest common divisor of 294 and 546 by storing these values at the labels a (represented as `"a"`) and b (represented as `"b"`) in an (empty) environment (`emptyEnv`). The denotational semantics (`ds`) of the `gcdStmt` and this environment yields an environment that contains the result at `"c"`. Hence applying the environment to `"c"` yields the value given by the denotational semantics of this While program. This can be computed by the following one-liner in Clean:

```
Start = ds gcdStmt (("a" ↦ 294) (("b" ↦ 546) emptyEnv)) "c"
```

As expected this produces the value 42, which is the correct result.

2.4 Natural Semantics

The natural semantics is a big step semantics 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

```
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

of the semantics in figure 5 is identical to the denotational semantics given in figure 4.

This natural semantics (`ns`) can be executed in exactly the same way as the denotational semantics (`ds`). As required this produces the same result for the `gcdStmt` shown above. Below we discuss how the equivalency of those semantics can be investigated more thoroughly.

In the traditional representation of the operational semantics, the semantics is often specified by a 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:

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

The part within the square brackets on the left is the name of this axiom. The phrase if $\mathcal{B}[\![b]\!]\ s$ is the condition. If this condition does not hold, the rest of the axiom cannot be applied.

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 one order. In figure 5 we use one function to represent all axioms. 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
```

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 operational 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[2].

```
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] The pattern `c=:(Final s)` for function arguments allows us to use the entire argument as `c` in the right-hand side, as well as to do a pattern match on a constructor and to use sub-arguments (like `s`) in the right-hand side.

2.6 Sanity Checks

Since our semantics is just a set of types and functions in a functional programming language, we can use the language implementation (here Clean) for elementary sanity checks of the specified semantics. In our examples the Clean compiler checks that all identifiers used are defined and used in a proper type context. Also the compiler checks the type correctness of each and every subexpression. Moreover the compiler can produce a warning if the semantics is a partial function, e.g. since no semantics is given for one of the language constructs. For a semantics defined in mathematics one has to rely on humans. If the semantics is formalized in some proof tool, this tool will provide similar support.

Although this sounds simple, it appears to be very useful. The draft version of these notes contained for a very long time the rule

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

Neither the authors nor the readers discovered that there was a wrong meta variable used in this definition[3] for a very long time. This mistake was found with a lot of luck. If we would have made the corresponding error in the semantics expressed in Clean(line 9 of figure 4), the Clean compiler would have indicated a problem immediately. We do not claim that the compiler of the host language will find all issues, but in our experience it really helps to get the semantics correctly.

2.7 Simulating 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. At the push of a button the iTask system can show the trace of the reduction of such a program using the structural operational semantics defined by sos, or the value of all used variables in the environment that is yielded by execution of the semantics. Since this is a straightforward application of iTasks that has its power in interactive simulation, we do not elaborate on this here.

2.8 Testing Properties of the Semantics

Another possibility is to test properties of the semantics using our model based test system G∀st. We now give a quick overview of model-based testing with logical properties as models, see [8] for details. Next we will show some applications of model-based testing for the semantics of While.

Model-Based Testing of Logical Properties. The main difference between model-based testing with logical properties and an ordinary automated test tool

[3] Another option is that the readers discovered it, but didn't tell it to the authors. Given the other feedback, this is highly unlikely.

like JUnit[4] is that the model-based test tool generates the test case while in JUnit they are always specified by the programmer. Both kind of test tools execute the tests automatically and give a verdict.

We will illustrate this with a few simple examples. First we introduce the enumeration type `Color` and the recursive algebraic data type `Tree`.

```
:: Color  = Red | Yellow | Blue
:: Tree a = Leaf | Node (Tree a) a (Tree a)
```

The function `mirror` recursively flips the left and right subtree of nodes in the tree.

```
mirror :: (Tree a) → Tree a
mirror Leaf         = Leaf
mirror (Node l a r) = Node (mirror r) a (mirror l)
```

A desirable property of this `mirror` function is that applying it twice to a tree t yields the original tree. In logic this is:

$$\forall\, t \in \mathsf{Tree}\ \tau\,.\, \mathsf{mirror}\ (\mathsf{mirror}\ t) \;=\; t$$

In a JUnit setting the user chooses some typical values for the tree `t` and checks the property for these values. For example `equal (mirror (mirror Leaf)) Leaf` and `equal (mirror (mirror (Node Leaf Red (Node Leaf Blue Leaf)))) (Node Leaf Red (Node Leaf Blue Leaf))`.

In the model-based test tool `G∀st` we only indicate the type of arguments that need to be generated, otherwise it is a direct translation of the logical property:

```
propMirror1 :: (Tree Color) → Bool
propMirror1 t = mirror (mirror t) == t
```

Any function yielding a Boolean, or special test value of type `Property`, can be interpreted as property where the arguments are interpreted as universal quantified variables. `G∀st` test this property by generating a large number of elements of type `Tree Color` and evaluates the function `propMirror1` for these test cases. We have to indicate the type in order to allow `G∀st` to generate the test cases.

The list of test cases is generated by the generic function `ggen`. In most situations we can just derive the generation from the generic algorithm:

```
derive ggen Color, Tree
```

For any data type this generic generation algorithm [6] produces a list of all instances of that type. If the instances of the type have more than one subtype, these instances are generated in a breadth first way. To make the testers happy

[4] JUnit is a well known test tool for the Java programming language. It executes a number of predefined tests (basically expression yielding a Boolean value). If such a test yields `True` it is regarded a success, otherwise it indicates an issue. JUnit generates a report giving the number of successes and issues. For the test yielding `False` JUnit prints the arguments of the test function. Originally JUnit was associated with Java, meanwhile it is ported to most mainstream languages.

there is some pseudo random perturbation of the order of elements compared to pure breadth first traversal. As a consequences elements can appear somewhat earlier or later in the list of test values. For instance the first 10 elements generated of type `Tree Color` are:

```
[Leaf
,Node Leaf Blue Leaf
,Node (Node Leaf Red Leaf) Blue Leaf
,Node Leaf Yellow Leaf
,Node (Node (Node Leaf Red Leaf) Red Leaf) Blue Leaf
,Node (Node Leaf Red Leaf) Yellow Leaf
,Node Leaf Blue (Node (Node Leaf Red Leaf) Red Leaf)
,Node (Node Leaf Yellow Leaf) Blue Leaf
,Node (Node (Node Leaf Red Leaf) Red Leaf) Yellow Leaf
,Node (Node Leaf Red Leaf) Blue (Node (Node Leaf Red Leaf) Red Leaf)
]
```

Executing `Start = test propMirror1` yields Pass[5], no counterexamples are found in the executed tests.

As a second property we might formulate that the function `mirror` should change its argument tree:

```
propMirror2 :: (Tree Color) → Bool
propMirror2 t = mirror t =!= t
```

The test system immediately finds a large number of counterexamples like `Leaf`, `Node Leaf Red Leaf` and so on. If we think a little harder we would realize that this property holds for nonsymmetric trees. An appropriate way to improve the property is by adding a precondition; only if the tree is not symmetric mirroring should change it:

```
propMirror3 :: (Tree Color) → Property
propMirror3 t = not (symmetric t) ==> mirror t =!= t

symmetric Leaf         = True
symmetric (Node l _ r) = l === r && symmetric l && symmetric r
```

Another direction to tackle the problems with the second property is by generating only asymmetric trees. The generic generation algorithm cannot do this all by itself. A possible solution is replacing the generic generation algorithm by a specific instance that transforms lists to list like trees (all left subtrees are empty):

```
ggen{|Tree|} f n r = map listToTree (ggen{|*→*|} f n r)

listToTree :: ([t] → Tree t)
listToTree = foldr (Node Leaf) Leaf
```

[5] Only in exceptional cases G∀st is able to *prove* properties by exhaustive testing. For such tests we either have to give a finite test suite by hand, or the only universal quantified variables are of nonrecursive algebraic data types or finite primitive types (like `Bool` and `Char`).

Now G∀st uses this generator of list-like trees in any property tested. If we want specific test cases only for one test, we should use the operator `For` (see below). Executing the test shows that this does not remove the problems with `propMirror2`, if there are less than two `Nodes` in the tree, it is still symmetric.

Some other possibilities of the test system are introduced below, see [8] for a more elaborate and complete treatment.

Model-Based Testing of the Semantics of While. Now we turn to model-based testing properties of the semantics of While. Consider the factorial function in While:

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

instance one AExpr where one = Int 1
```

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"`. In logic this is:

$$\forall \mathrm{sem} \in (\mathrm{Stmt} \to \mathrm{Env} \to \mathrm{Env})\,.\, sem\ facStmt\ (x \mapsto 4\ (\lambda\, x\,.\,0))\, y = 24$$

In G∀st this property reads:

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

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])
```

Executing this program yields `Proof`. The Proof by exhaustive testing produced by the test system gives us confidence in the consistency and hence the correctness of the various semantics.

Of course one can think of many incorrect semantics that produce an incorrect environment in the example above. That is exactly the reason why we explicitly enumerate the semantics that needs to be used in testing this property, rather that let the test system generate a semantics.

Generating Terminating Statements. 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 only want 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.

We cannot directly use the generic algorithm [6] to generate statements to be used in the tests: instances of the type `Stmt`. Since the generic generation algorithm has no notion of statements nor of their semantics, it will simply generate valid instances of this type without taking care of the desired properties. In general there are two ways to solve this kind of generation issues. First we can specify the generation completely by hand instead of using the generic algorithm. In this situation we need to derive `ggen{|Stmt|}` such that it yields (an infinite) list of terminating statements. In general this can be quite tricky. An easier and more elegant alternative is to define an additional data type that represents only terminating statements `Gstmt` and the associated transformation `conv` to statements. Once we start doing this we also introduce types to control the generation of numbers, variables and so on. The only interesting point is that we replace general while-loops that might cause nontermination with for-loops with a decreasing counter and a small counter value.

```
:: Gstmt
    = Assign Gvar GAExpr
    | Comp Gstmt Gstmt
    | Gskip
    | GIF GBExpr Gstmt Gstmt
    | GFor (Maybe GBExpr) Gvar GNat Gstmt

:: GNat = GNat Int
:: Gvar = Gvar String

:: GAExpr
    = GInt Int
    | GVar Gvar
    | GPlus GAExpr GAExpr
    | GMinus GAExpr GAExpr
    | GTimes GAExpr GAExpr
```

Instances of the generic generation function `ggen` for the natural numbers and variables are defined by hand: positive numbers and single letter variables.

```
ggen{|GNat|} n r = map GNat [0..]
ggen{|Gvar|} n r = map (Gvar o toString) ['a'..'z']
```

For all other and more complex types, we use the generic algorithm incorporated in G∀st. Since those types have many combinators and (double) recursion it would be rather cumbersome to define generation for those types manually.

```
derive ggen Gstmt, GAExpr, Maybe
```

For convenience we introduce a class `conv` to do the transformation from the generation types to the types used in the semantics.

```
class conv a b :: a → b
```

The transformation of expressions is a very simple one to one mapping:

```
instance conv GAExpr AExpr
where
    conv (GInt i)        = Int i
    conv (GVar (Gvar v))= Var v
    conv (GPlus x y)     = conv x +. conv y
    conv (GMinus x y)    = conv x -. conv y
    conv (GTimes x y)    = conv x *. conv y
```

The transformation of most statements is equally simple. Only in the transformation of for-loops to while-loops we need to do some work: we introduce a semi-fresh counter, initialize it with the given integer value and enter a while loop. The condition is the combination of the given boolean value (if the maybe type provides it) and the boolean expression that checks if the value of the counter is still positive. Since the introduced variable is only fresh in the body of the loop, introducing this variable can alter the semantics of the existing statement in `Gstmt` form. Since our only purpose is to generate valid and terminating statements, this is no problem whatsoever.

```
instance conv Gstmt Stmt
where
    conv (Assign (Gvar v) e)     = v :=. conv e
    conv (Comp x y)              = conv x :. conv y
    conv Gskip                   = Skip
    conv (GIF c t e)             = IF (conv c) (conv t) (conv e)
    conv (GFor b (Gvar v) (GNat n) s)
     =  counter :=. (Int n) :.
        While cond
            (body :.
             counter :=. Var counter -. one
            )
    where
        counter = fresh v (allvars body)
        cond    = case b of
                        Nothing = c0
                        Just b  = c0 &&. conv b
        body    = conv s
        c0      = Int 0 <. Var counter
```

The function `allvars` that yields all variables that occur on the left-hand side of an assignment in a program in the language While is defined as

```
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)
```

Having the generic generation of the new data types and the conversion of these data types to the data types used for statements in the semantics, we can define the generation of statements as a simple combination of these items.

```
ggen{|Stmt|}  n r = map conv stmts
where stmts :: [Gstmt]
      stmts = ggen{|*|} n r
```

Since `Gstmt` only contains quickly terminating statements, the statements of type `Stmt` generated in this way will also terminate.

The first 10 elements of the list of statements generated are:

```
[ a :=. a
, a0 :=. 0 :.
  While (0 <. a0 &&. -2147483648 <. a+.1+.0)
    (Skip :.
     a  :=. 0 :.
     a0 :=. a0 -. 1
    )
, b :=. a
, a :=. (0 *. (1 -. 0)) +. 2147483647
, Skip
, a :=. -2147483648 :.
  a :=. -2147483648 :.
  a :=. -1 :.
  a :=. -1
, IF TRUE
    (a :=. -1 -. a)
    (a :=. 2147483647)
, IF FALSE
    (a :=. -1 -. a)
    (a :=. 2147483647)
, c :=. a
, Skip :.
  a :=. 1
]
```

This generation of terminating statements is used in each and every property below that quantifies over statements.

Comparing Environments. 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 use the function `allvars` defined above. The function `eqEnv` that determines the equivalence of environments is straightforward.

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

Testing Semantic Properties of While Part 2. After these preparations, the property of testing equivalences between natural semantics and denotational semantics becomes:

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

Similar properties can be stated 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 =: (decl :. loop =: (While cond body))) = conv (GFor b v n s)
```

Of course we do not require that this property holds for each and every semantics one can imagine. It is sufficient if this property holds for the three semantics introduced above. We achieve this by executing `test (propWhile For [ds,ns,sos])` instead of `test propWhile`. Also this property passes the tests.

In the same spirit we can test whether for all of our three versions of the semantics the binding direction of the semi colon is irrelevant. In mathematics this reads $\forall\, s,\ t,\ u\ \in\ \mathit{Stmt}\,.\,\mathit{sem}\ ((s\ ;\ t)\ ;\ u)\ \mathit{env}\ =\ \mathit{sem}\ (s\ ;\ (t\ ;\ u))\ \mathit{env}$. As a property in G∀st this reads:

```
propSemiColon :: (Stmt Env → Env) Stmt Stmt Stmt → Property
propSemiColon sem s t u
 = name ";"
    λsem .
    (eqEnv  (sem (s :. (t :. u)) e)
            (sem ((s :. t) :. u) e)
            (allvars (s :. t :. u))
    )
where e = emptyEnv
```

Obviously we have chosen to test this with an empty environment. If desired we can of course extend the property, and hence the test to various environments. Fortunately also this properties passes the tests.

In our final example we extend the first test we used for Euler's algorithm to all our semantics and all integer arguments. we compute the desired result by the function `gcd` provided by `Clean`'s standard libraries.

```
propGCD :: (Stmt Env → Env) (Int,Int) → Property
propGCD sem (a,b)
    = sem gcdStmt (("a" ↦ a) (("b" ↦ b) emptyEnv)) "c" == gcd a b
```

We can test this property for our three versions of the semantics and all arguments between 0 and 100 by executing:

```
Start = test ((λsem.propGCD sem For [(a,b) \\ a←[0..100], b←[0..100]])
                                   For [ds,ns,sos])
```

If we allow sufficient test instances, G∀st is able to prove this property by exhaustive testing the property for the given values after executing 30603 test cases.

These examples illustrates how one can formulate properties about the semantics in G∀st and test these properties fully automatically. This leaves us with the problem how to find properties to be tested. We handle this question in the next paragraph.

Developing Properties to be Tested. There is always human intelligence necessary to define properties of a semantics that can be tested in order to gain confidence in the quality of specification. Nevertheless, it is not all black magic. A few simple guidelines provide helpful hints to construct properties.

1. First there are the general properties that are known to hold in the semantics of the language at hand. Examples of such properties are `propSkip` and `propWhile` above. It is straight forward to transform the known mathematical properties to G∀st and to execute the associated tests. This is an easy check after the basic sanity checks provided by the Clean compiler.
2. We can often easily indicate properties of a semantics or structures used in such a semantics.

   For instance for the environment used here: the empty environment should yield the value `0` for all variables, and after storing a result for some variable, looking up the value of that variable should produce the stored value.

   ```
   pEnv1 :: Var → Bool
   pEnv1 v = emptyEnv v == 0

   pEnv2 :: Var Res → Bool
   pEnv2 v i = (v ↦ i) emptyEnv v == i
   ```

   In the same spirit, but slightly more advanced, storing a value with a different name in an environment does not change the value of the original variable in the environment.

   ```
   pEnv3 :: Var Res Var Res → Property
   pEnv3 v i w j = w ≠ v ==> (w ↦ j) ((v ↦ i) emptyEnv) v == i
   ```

The property `pEnv3` produces a `Property` instead of a Boolean as result since we used the logical combinator `=>`. This operator mimics implication, $\Rightarrow$, from logic and is used here as additional constraint on the values used in the tests.

3. Finally, the issues[6] found during the development of the semantics are a valuable source of testable properties. If there was ever an issue with the semantics of a specific construct and value, we can state the property that it should produce the correct value. It is even better if we can generalize such a property for all values and introduce a universal quantified variable. We will encounter some examples in the iTask semantics given below.

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 for iTask workflow management system.

3 A Semantics for iTasks

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 [16]. 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 [9]. 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.

[6] Issue is the notion used in model based testing to indicate any failing test. This includes all sources of failure. A failing test indicates not always an error in the tested software. Failing test can also be caused by, for instance, erroneous properties or invalid test data.

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 a 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 these notes 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 GUI 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, or a tailor-made editor can be defined for that type.

In these notes 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 to 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 the constructor .||. for the or-combinator, -||-, and the combinator .&&. for and-combinator, -&&-, 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 simulator that preserves types (see section 7). 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
```

No implementation of the iTask system will show an editor for the type VOID, the only value of this type cannot be changed. As a consequence the GUI of the ButtonTask will be only the button with the label s. This is exactly what is required.

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 (the user executing a task), nor the workflow developer has to worry about these identifications. The fact that the iTask system is a multi-user system implies that there are multiple views on the workflow. Each worker can generate events, input for the workflow, 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 ids of subtasks must be persistent. Hence, the numbering system has to be more advanced than just a numbering of the nodes. The semantics in these notes ignore 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 with the function `next`. For every `id`, `i`, we have that `next i` $\neq$ `i`. In this way we distinguish inputs for a specific 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 uniquely identifies the subtask and one number serves as version for this subtask. This version number is increased each time when the task accepts an edit event. This implies that applying an event repeatedly to a task has at most once an effect. 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 these notes.

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 generates 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 tree 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 tree has an effect on all subtasks that can be rewritten without user events. For instance, the task `editTask "ok" 1 -||- return 5` is normalized to `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 tree. 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 main 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 same type as stored currently in 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 edit events (line 3) as well as button events (line 4). 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 other constructs pass

```
instance @. ITask Event                                                   1
where                                                                     2
  (@.) (EditTask i n v) (Event j (EE w)) | i==j = EditTask (next i) n w   3
  (@.) (EditTask i n v) (Event j BE)     | i==j = Return (BVal v)         4
  (@.) (t .||. u) e  = case t @. e of                                     5
                         t=:(Return _) = t                                6
                         t = case u @. e of                               7
                               u=:(Return _)  = u                         8
                               u              = t .||. u                  9
  (@.) (t .&&. u) e  = case (t @. e, u @. e) of                           10
                         (Return v, Return w) = Return (Pair v w)         11
                         (t, u)               = t .&&. u                  12
  (@.) (Bind i t f) e = case t @. e of                                    13
                         Return v = normalize i (f v)                     14
                         t        = Bind i t f                            15
  (@.) t e = t                                                            16
```

Fig. 7. The basic semantics of iTasks

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

An event is enabled if there is a task in the task tree that is rewritten when the event is applied. For an edit task the enabled events are the edit and button event with the corresponding id. Also the edit tasks that are composed with the combinators `.||.` and `.&&.` and on the left-hand side of the `Bind` operator in an enabled subtask are enabled. All events that belong to the right-hand side of a bins operator are not enabeled. All events that are not enabled are ignored (line 16).

A properly numbered task tree remains correctly numbered after reduction. Editors that receive a new value get a new unique 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 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 normalized and properly numbered within the surrounding tree.

The handling of events for a task tree is somewhat similar to the reduction in 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. The subtasks in the right-hand side of a bind do not exists until the `Bind` operator is rewritten. Until this rewrite takes place there is just a function that will produce the new task. Sometime this is rather confusing for human reader since the subtasks in the function seem to be present. Textually these subtasks are there, but operationally they are there only after the evaluation of the function. For instance, in task `t5` defined below the subtask `ButtonTask id2 "c" ..` is not enabled until `EditTask id1 "b" (Int 5)` is finished.

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))`. However, for any given scenario that produces a normal form the obtained normal form is unique. 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 are 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` none of the events is needed, the task is finished as soon as the task `t1` or the task `t2` is finished. Normalization is only included 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 function `collectEdit` yields all enabled edit events in the given task tree with the current value of the editors in the tree. All edit events with the same id and another value of the same type will also be accepted when we apply them to the task tree. The list of events is needed for the simulation of the task discussed in section 7.

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, because these tasks 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:

$$Type = Int \mid String \mid VOID \mid Pair\ Type\ Type$$

To prevent the introduction of yet another data type, we represent the types yielded by tasks in these notes 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
```

This reuse of the type `Val` to represent the type of instances of this type appears to be very handy in the generation of values needed to test properties of iTasks.

In the next section we use to semantics and notions introduces in this section to define equivalence of tasks.

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 these notes we 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.

This notion of equivalence is based on observational equivalence for workers. At any moment during the execution of a task the worker should have the same options for entering input (generating events), and both tasks should yield the same normal form for any sequence of inputs that produces a normal form in one of the 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 [21] 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, a result of type `Pair Int String` can never be equal to a result of type `Pair String Int`. When we swap the elements in the resulting pair of either u7 or u8 these tasks are equivalent: for example u7 $\Rrightarrow$ λ(Pair a b) → Return (Pair b a) $\cong$ u8.

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. This implies that an testing approach needs to approximate this equivalence relation in some, preferably safe, way. However, in many situations we can decide these relations between tasks by inspection of the task trees that determine the behavior of the tasks. The next sections show how equivalence can be approximated and used in test of the semantics of iTasks.

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 tried values the properties holds. The property might hold for all other values, but it is also possible that there exist 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 [8]. 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', $\vee$), conjunction ('and', $\wedge$), 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 `ITask`s 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 are `Pass` or `Proof`.

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

The edit events are sorted before we compare them in order to get rid of possible different ids. We only compare the values of edit events in the comparison of events. Button events are considered to be smaller than edit events.

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` $\simeq$ `u.||.t`), and associative ((`t.||.u).||.v` $\simeq$ `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                      15
eqStruc n s=:(a .&&. b)   t               = (inNF a||inNF b) ∧ equ n s t   16
eqStruc n s               t=:(x .&&. y)   = (inNF x||inNF y) ∧ equ n s t   17
eqStruc n s               t               = CE                             18
```

This uses instances of ≃ for basic values (`BVal`) and values (`Val`). For these instances no approximations are needed. 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 permutations.

```
eqORn :: Int [ITask] [ITask] → Result
eqORn n xs ys
    = coversUnique (eqStruc n) xs ys ∧ coversUnique (eqStruc n) ys xs
```

The definitions are a direct generalization of the ordinary equality =.

The function `coversUnique` checks if there is an unique mapping between elements of the given lists using the comparison operator. As a simple example `coversUnique (ab.toResult (a==b)) [1,1,2] [1,2,1]` will produce `OK`, but comparing the list [1,1,2] and [1,2,2] as well as any list different length will produce `CE`.

```
coversUnique :: (a a→Result) [a] [a] → Result
coversUnique f xs ys = eq xs ys []
where
    eq [] ys zs = Proof
    eq [x:xs] [] zs = CE
    eq [x:xs] [y:ys] zs
        | f x y ≠ CE
            = eq xs (zs++ys) [] ∨ eq [x:xs] ys [y:zs]
            = eq [x:xs] ys [y:zs]
```

If the elements of the list can be sorted it is much easier to sort the lists of values and compare the elements one by one. The given algorithm works also if the lists cannot be sorted. We needed this to compare tasks, we have no less-then operator for tasks.

As indicated above the function `equ` takes care of applying events and further comparison of task trees. This function first checks if there are approximation steps to be done (n≤0). If no steps can be done the result is `Pass`. Otherwise we check compare the current value of the task trees, the number of button events, the number of needed events, and the edit events. If there are needed events we apply them to the tasks. Otherwise we try if any pairs of button events for the tasks is structurally equivalent.

```
equ :: Int ITask ITask → Result
equ n t u
 | n≤0
    = Pass
    =     val t == val u ∧ lbt == lbu ∧ lnt == lnu
      ∧ sort (collectEdit t) == sort (collectEdit u)
      ∧ if (lnt>0)
           (eqStruc (n-lnt) (t @. nt) (u @. nu))
          (if (lbt>0)
           (exists N [ eqStruc (n-1) (t @. i) (u @. j)
                     \\ (i,j) ← diag2 bt bu
                     ]
           )
           (case (t,u) of
               (Return x,Return y) = x≃y
               _ = CE)
           )
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
```

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 semantic description many versions have been created in order to find a concise formulation of the semantics and an effective check for equivalence.

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
```

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 [7]. 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 $\not\simeq$ t for tasks that are not in normal form (p2) and test the associativity of the .||. 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 these notes. 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 collected from the task tree rather than 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)
```

When two tasks are equivalent it is not required that their buttons are exactly equivalent, some differences in layout and hence button labeling are allowed.

However, there should be an unique coverage of the buttons of those tasks. This is tested by property `p50`.

```
p50 :: ITask ITask→ Property
p50 s t = (s ≃ t) => (coversUnique (≃) (collect s) (collect t))
```

In general it is not enough to check that tasks are currently structural equivalent, this equivalence should be preserved after processing events. Details of this equivalence are beyond the scope of this paper. For our iTasksemantics we defined a notion of equivalent that covers this requirement. Many properties and hence use these notion of equivalence.

The definitions presented in these notes 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.

For iTasks it might be tempting to state properties in temporal logic. Currently temporal logic is not yet supported by G∀st, it is restricted to a slightly extended version of first order logic. G∀st does support also specifications by extended state machines and has a notion of behavioral equivalence of state machines. These state based specifications can be used to test state based properties of iTaskseffectively. However, this way of testing is outside the scope of this paper.

6 Related Work

This is certainly not the first paper on tool support for semantics, neither will it be the last one. There are several classes of related work.

The first class of papers aims to derive efficient implementations from the semantical description. Lakin and Pitts [10] propose a meta language for structural operational semantics. Their treated meta language is able to animate the described language, the long term goal is to execute it efficiently. They spend much effort in the implementation of a variable binding mechanism that is aware of α–equivalence, known as the *Barendregt variable convention* [3]. To that extend they implement *nominal unification* [24] to judge equivalence of environments (binding of variables to values). Cheney's scrap your nameplate [4] provides similar binding tools using generic programming techniques. We provide a more basic notion of binding without silent α–equivalence, nor do we aim to provide an efficient implementation. For the semantics given in this paper such an advanced notion of equivalence of environments is not needed. For specifying the semantics of, for instance, the λ-calculus, such a notion of equivalence would be a valuable addition. If necessary, such a notion of equivalence can be used instead of the simple environments used here.

A second class of related papers is about support for mechanical proofs of properties of the given semantics. These papers recognize that it is next to impossible to get a large semantical specification without an automatic sanity checking. To conquer this problem Sewell et al. [20] define the metalanguage Ott, specifications in this language are checked and can be compiled to LaTeX as well as code for proof assistants. Although proofs using a proof assistant are extremely valuable for semantical specifications and encouraging progress has been made with proof assistants in recent years, constructing such proofs still requires usually much human guidance [2]. If any detail of the semantics is changed the proofs need to be redone typically with new human guidance. For this reason proofs are typically delayed until the semantics is assumed to be correct, or omitted. Our test based approach gives quickly a fairly good approximation of the correctness of the properties stated. This appears to be very useful during the development of the semantics. We plan to investigate the combination of testing and proving properties as future research.

We neither aim to animate the semantics efficiently, nor do we directly target proof support. The possibilities for simulating the semantics using the iTask system as described in this paper are purely intended as an interactive way to create task and perform reductions. Efficiency is not an issue at all, the only purpose is the human validation of the observed reduction behavior.

Beauty, clearness and correctness of the semantics is our business. Our experience is that fast feedback from a test system is more valuable during the development of the semantics than support for proofs. Once the semantics is stable, off–line proofs do provide more confidence than tests. In our experience it is very hard to construct faithful properties for a semantics that does not hold and where the test system does not produce a counterexample quickly. In other words: if one of the stated properties does not hold, the test system G∀st is usually able to find a counterexample quickly. And if we have sufficient properties stated, issues in the semantics usually are indicated by counterexamples for one or more of the properties. Of course it would be nice to have a set of properties that completely fixes the desired semantics, but ad-hoc sets of properties appear to be effective as well.

Compared with these two classes of tools our approach has the advantage that no special purpose language needs to be designed, implemented and mastered. An existing and well-known language is reused for a new goal. We reuse all existing features of the language as well as all existing tooling (IDE, compiler, test-tool, ..), only a small number of tailor made features have to be provided to obtain a powerful embedded modeling language for semantics. A potential pitfall is that the semantics of the embedding functional language is silently inherited in the given semantics. The semantics of high level functional languages is usually not complete and rigourously defined. For Clean the basic semantics is given as a term graph rewrite system [17], de Mol [13] gives a precise definition of large parts of Clean. Since the semantics definitions directly corresponds to their counterparts in a mathematical formulation of the semantics the meaning of semantic formulation is never an issue. Moreover, only very basic rewrite semantics of the embedding language is used. All places where the semantics of the functional embedding language might be a little unclear are easily avoided.

The ability to test properties of the described semantics is an important contribution of our work.

In [5] Danvy describes various semantics in ML for simple systems and their transformation.

A third class of related papers is about semantics of workflow systems. The semantics of many other workflow systems is based on Petri-nets, e.g. [19], actor-oriented directed graphs (including some simple higher order constructs) [12], or abstract state machines (ASM) [11]. 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.

7 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:

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

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 the ordinary iTasks the type system does not allow to combine (which indeed is semantically not desirable) 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 for the iTask semantics 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. Many of these incorrect versions look very plausible for humans, without the test system one might believe that this version of the semantics is actually correct. This does give us confidence in the power of automatic testing of semantical properties. Nevertheless, a successful test is in general not a proof. Proving properties remains necessary to obtain maximum certainty about the properties. Using automatic testing as a first indication of correctness will reduce the proving effort significantly, there is a strongly reduced change that we try to prove incorrect versions of the semantics.

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. Achten, P., van Eekelen, M., de Mol, M., Plasmeijer, R.: An Arrow based semantics for interactive applications. In: Morazán, M. (ed.) Proceedings of the 8th Symposium on Trends in Functional Programming, TFP 2007, New York, NY, USA, April 2-4 (2007)
2. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The poplmark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
3. Barendregt, H.: The lambda calculus, its syntax and semantics (revised edition). Studies in Logic, vol. 103. North-Holland, Amsterdam (1984)
4. Cheney, J.: Scrap your nameplate (functional pearl). SIGPLAN Not. 40(9), 180–191 (2005)
5. Danvy, O.: From reduction-based to reduction-free normalization. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 66–164. Springer, Heidelberg (2009)
6. Koopman, P., Plasmeijer, R.: Generic generation of elements of types. In: Proceedings of the 6th Symposium on Trends in Functional Programming, TFP 2005, Tallin, Estonia, Septmeber 23-24, pp. 163–178. Intellect Books, Bristol (2005) ISBN 978-1-84150-176-5
7. Koopman, P., Plasmeijer, R.: Automatic testing of higher order functions. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 148–164. Springer, Heidelberg (2006)
8. Koopman, P., Plasmeijer, R.: Fully automatic testing with functions as specifications. In: Horváth, Z. (ed.) CEFP 2005. LNCS, vol. 4164, pp. 35–61. Springer, Heidelberg (2006)
9. Koopman, P., Plasmeijer, R., Achten, P.: An executable and testable semantics for iTasks. In: Scholz, S.-B. (ed.) Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL 2008, pp. 53–64. University of Hertfordshire, UK (2008)
10. Lakin, M.R., Pitts, A.M.: A metalanguage for structural operational semantics. In: Morazán, M. (ed.) Trends in Functional Programming, vol. 8, pp. 19–35. Intellect (2008)
11. Lee, S.-Y., Lee, Y.-H., Kim, J.-G., Lee, D.C.: Workflow system modeling in the mobile healthcare B2B using semantic information. In: Gervasi, O., Gavrilova, M.L., Kumar, V., Laganá, A., Lee, H.P., Mun, Y., Taniar, D., Tan, C.J.K. (eds.) ICCSA 2005, Part II. LNCS, vol. 3481, pp. 762–770. Springer, Heidelberg (2005)
12. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice & Experience 18, 2006 (2005)

13. de Mol, M.: Reasoning About Functional Programs - Sparkle: a proof assistant for Clean. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen,(2009) ISBN 978-90-9023885-2
14. de Mol, M., van Eekelen, M., Plasmeijer, R.: The mathematical foundation of the proof assistant Sparkle. Technical Report ICIS-R07025, Institute for Computing and Information Sciences, Radboud University Nijmegen, (November 2007)
15. Nielson, H., Nielson, F.: Semantics with applications: a formal introduction. John Wiley & Sons, Chichester (1992)
16. Plasmeijer, R., Achten, P.: iData for the world wide web - Programming interconnected web forms. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 242–258. Springer, Heidelberg (2006)
17. Plasmeijer, R., van Eekelen, M.: Functional programming and parallel graph rewriting. Addison-Wesley Publishing Company, Reading (1993) ISBN 0-201-41663-8
18. Plotkin, G.D.: The origins of structural operational semantics. Journal of Logic and Algebraic Programming 60-61, 3–15 (2004)
19. Russell, N., ter Hofstede, A., van der Aalst, W.: newYAWL: specifying a workflow reference language using coloured Petri nets. In: Proceedings of the 8th 2007 (2007)
20. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: effective tool support for the working semanticist. SIGPLAN Not. 42(9), 1–12 (2007)
21. Stirling, C.: The joys of bisimulation. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 142–151. Springer, Heidelberg (1998)
22. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge (1977)
23. Team, T.C.D.: The Coq proof assistant reference manual, (version 7.0) (1998), `http://pauillac.inria.fr/coq/doc/main.html`
24. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theoretical Computer Science 323, 473–497 (2004)