6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Verifying invariants of abstract functional objects—a case study*

Zoltán Horváth, Tamás Kozsik, Máté Tejfel

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest
e-mail: hz@inf.elte.hu, kto@elte.hu, matej@inf.elte.hu

Abstract

In a pure functional language like Clean the values of the functional variables are constants; variables of functional programs do not change in time. Hence it seems that temporality has no meaning in functional programs. However, in certain cases (e.g. in interactive or distributed programs, or in ones that use IO) we would like to consider a series of values computed from each other as different states of the same "abstract object". For this abstract object we can already prove temporal properties (e.g. invariants). In this paper we present a case study: our example is an interactive database with some simple operations like updating, sorting, querying records. We specify an invariant property of our program and we show how to prove this property. We utilize Sparkle, a theorem prover designed for the Clean language. Since Sparkle is not capable of handling temporal logical properties, the application of certain rules of the proof system are performed by hand. This way we simulate the behaviour of a more sophisticated theorem prover, which is currently under development.

Categories and Subject Descriptors: D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.1 [Logics and meanings of programs]: Specifying and Verifying and Reasoning about Programs - *invariants*;

Key Words and Phrases: Verification, invariant properties, abstract functional object, Clean, Sparkle

1 Introduction

Temporal logical operators (such as "nexttime", "sometimes", "always" and "invariant") are very useful for proving correctness of (sequential or parallel) imperative programs. All these operators can be expressed based on the "weakest precondition" operator [9, 10].

The temporal logical operators describe how the values of the program variables (which constitute the so-called program state) vary in time. For example, the weakest precondition of a program statement with respect to a postcondition holds for a state "*a*" if and only if the statement starting from "*a*" always terminates in a state for which the postcondition holds. The weakest precondition of a statement is possible to compute in an automated way: one has to rewrite the

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr.T037742 and by Bolyai Research Scholarship of Hungarian Academy of Sciences.

postcondition according to the substitution rules defined by the statement. The details of this technique can be found in e.g. [5].

Temporal properties are not really used when reasoning about functional programs (among the few exceptions are e.g. [8, 11, 3, 12]). In a pure functional programming language a variable is a value, like in mathematics, and not an "object" that can change its value in time, viz. during program execution. Due to referential transparency, reasoning about functional programs can be accomplished with a fairly simple mathematical machinery, using, for example, classical logic and induction (see e.g. [4]). This fact is one of the basic advantages of functional programs over imperative ones.

In our opinion, however, in certain cases it is natural to express our knowledge about the behaviour of a functional program (or, we had better say, our knowledge about the values the program computes) in terms of temporal logical operators. Moreover, in the case of parallel or distributed functional programs, temporal properties are exactly as useful as they are in the case of imperative programs. For example, those invariants which are preserved by all components of a distributed or parallel program, are also preserved by the compound program.

According to our approach, certain values computed during the evaluation of a functional program can be regarded as successive values of the same "abstract object". This corresponds directly to the view which certain object-oriented functional languages hold.

This paper aims to show a simple example how to interpret and prove temporal properties of functional programs using the object abstraction method. We inspect a special temporal property, an invariant. A property P is an invariant with respect to a program and an initial condition if P holds initially (namely it is implied by the initial condition) and all the atomic statements of the program preserve P. Note that the second part of this requirement can be expressed with the weakest precondition operator: for all atomic statements, the weakest precondition of the statement with respect to P must follow from P.

We have chosen Clean [14], a lazy, pure functional language for our research. An important factor in our choice was that a theorem prover, Sparkle [4] is already built in the integrated development environment of Clean. Sparkle supports reasoning about Clean programs almost directly. We would like to extend the first-order logic used by Sparkle with temporal operators, thus making semi-automated reasoning about parallel, interactive or distributed Clean programs easier.

2 The program to reason about

In this case study we use a very simple example program modelling a database of financial transactions. In this example a transaction is made up of two integer numbers; the first one represents the date when the transaction occured, and the other one stores the amount of money transfered in the transaction. The database contains a list of transactions and the overall sum of the amounts transfered in the transactions.

The types involved in this example can be specified in a functional language (e.g. in Clean) in the following way.

```
:: Transaction :== (Int, Int)          // a pair of date and amount
:: DB :== (Int, [Transaction])         // sum and list of transactions
```

Strict data structures and strict function arguments make reasoning simpler. Furthermore, for some irrelevant technical reasons, in this example we have not used the built-in polymorphic

list datatype, and we have defined the list of transactions without the using the synonym type `Transaction`. Hence our type definitions are as follows:

```
:: Transaction :== (!Int, !Int)             // date and amount
:: List = Nil | Cons !(!Int,!Int) !List     // list of transactions
:: DB :== (!Int, !List)                     // sum and transactions
```

One can develop some basic operations for manipulating the database. A new (empty) database can be created by invoking the function `newDB`. Functions `insertDB`, `remove-First` and `sortDB` can be regarded as state transition functions, which describe how the state of a database will change. In the FP terminology, these functions compute the "new value" of the database from the "old value". Function `insertDB` extends the database with a new transaction, `removeFirst` removes the first transaction from the list of transactions, and `sortDB` sorts the transactions by date. (In this simple example program we might, but not obliged to, assume that the date is a primary key.)

```
newDB:: -> !DB
newDB = (0, Nil)

insertDB:: !(!Int,!Int) !DB -> DB
insertDB t=:(date,amount) (sum,list) = (sum+amount, Cons t list)

removeFirst:: !DB -> DB
removeFirst (sum,Nil) = (sum,Nil)       // nothing to remove, skipping
removeFirst (sum, Cons (date,amount) list) = (sum-amount, list)

sortDB:: !DB -> DB
sortDB (sum,list) = (sum, sort_ins list)
```

The definition of `sortDB` applies function `sort_ins`, which implements an insertion sort algorithm. Function `insrt` (invoked by `sort_ins`) requires that the operation < be defined for type `Transaction`. In our example < compares the first field of transactions, namely date.

```
sort_ins:: !List -> List
sort_ins Nil = Nil
sort_ins (Cons x xs) = insrt x (sort_ins xs)

insrt:: !Transaction !List -> List
insrt e Nil = Cons e Nil
insrt e ls=:(Cons x xs) = if (x<e) (Cons x (insrt e xs)) (Cons e ls)

instance < Transaction where (<) a b = (fst a) < (fst b)
```

Now we can develop a simple "scenario" application, which is built upon the basic operations. One can imagine that this scenario simulates an interactive session between a database management application and an end-user. The input to this scenario is a database and a transaction. First we insert the transaction into the database, then we sort the resulting database, finally we remove the first transaction stored in the (sorted) database.

```
scenario :: !DB !(!Int,!Int) -> DB
```

```
scenario db t
   # db = insertDB t db
   # db = sortDB db
   # db = removeFirst db
   = db
```

3 Object abstraction

Before formulating and proving temporal properties of our program, we have to define "abstract objects", that is we have to specify which functional (mathematical) values correspond to different states of the same abstract object. Here we will introduce a single abstract object, a database, whose consecutive states will be the value given as argument to `scenario`, and the values computed by functions `insertDB`, `sortDB` and `removeFirst`, respectively. Note that the program text has also suggested the very same abstraction: the programmer chose the same name, viz. `db`, to different functional values. This was possible due to the scoping rules of Clean with respect to the "let-before" (`#`) construct. The `scenario` function could have been equivalently defined without hiding of variables in this way:

```
scenario :: !DB !(!Int,!Int) -> DB
scenario db t
   # db1 = insertDB t db
   # db2 = sortDB db1
   # db3 = removeFirst db2
   = db3
```

The names of the functional variables that constitute the object abstraction are `db`, `db1`, `db2` and `db3`. Note that the choice of the values for the states of the abstract object also determines the state transitions we have to deal with during the proof of temporal properties.

How can a programmer supply this information? In our framework an integrated programming environment (similar to the one currently available for Clean, [16]) will help the programmer develop, and reason about, programs. This IDE will store various data about a program in a relational data base. This data base contains different sorts of compile time information (e.g. symbol table, syntax tree) which enable Model-View-Controller based refactoring tools to operate on these data [17]. One of the views is the "source code view", another view may contain the properties of the program and the proof of these properties. Certain views will be human readable, other views will be processed by different programs. For example, it will be possible to create a view in a format in which Sparkle internally represents Clean programs.

Among the controllers there will be one which allows the programmer to select—e.g. in a graphical user interface, by mouse-clicking on the source code—the variables, or expressions that belong to the state transition tree of an abstract object. Another controller will make it possible to enter temporal logical properties of the abstract object. Then these temporal logical properties can be proven by an appropriate theorem prover. To accomplish this goal, the current theorem prover for Clean (Sparkle) has to be altered: we will extend it with tactics (elementary proof steps) to handle certain temporal logical operators. Since this enhanced theorem prover is not yet implemented, in this case study will use Sparkle where possible, and simulate the application of the lacking (temporal logical) tactics by hand.

4 An invariant of the abstract object

The invariant property of the abstract object chosen in the previous section will be the following: if we sum up the amounts of money appearing in the transactions stored in the second part of the database, we get the first ("sum") part of the database. In the current version of Sparkle, the definitions (functions and predicates) required to formulate a theorem should be given in Clean. We will make use of a function that sums up the amounts appearing in a list of transactions.

```
sumUp:: !List -> Int
sumUp Nil = 0
sumUp (Cons (date,amount) list) = amount + sumUp list
```

Let us denote the abstract object "database" by $\boxed{\text{db}}$. At first glance, we could formulate an invariant property of $\boxed{\text{db}}$ in the following way:

$$\text{fst}\ \boxed{\text{db}} \quad = \quad \text{sumUp}\ \left(\text{snd}\ \boxed{\text{db}}\right)$$

When reasoning about programs, one must always take undefined results into account. For example, in a functional language like Clean it is possible to apply partial functions, or to work with lazy or even infinite data structures. Certain use of partial functions and lazy data structures can lead to run-time errors or infinite computations. In a theorem prover such "undefined results" can be modelled by a special $\perp$ (undefined) value. In Sparkle the predefined predicate `eval` is used to express that an expression does not contain undefined parts, that is the "complete evaluation" of the expression is possible. For the user-defined data types the programmer should provide the appropriate definitions of `eval`.

```
instance eval Transaction
where eval (date,amount) = eval date && eval amount

instance eval List
where eval Nil = True
      eval (Cons t ts) = eval t && eval ts

instance eval DB
where eval (sum,list) = eval sum && eval list
```

Now it is possible to put down the required invariant property of $\boxed{\text{db}}$:

$$I\left(\boxed{\text{db}}\right): \qquad \text{fst}\ \boxed{\text{db}} \quad = \quad \text{sumUp}\ \left(\text{snd}\ \boxed{\text{db}}\right) \quad \wedge \quad \text{eval}\ \boxed{\text{db}} \tag{1}$$

Next we will formulate an initial property $Q(db,t)$ of `scenario`.

$$Q(db,t): \qquad \text{eval}\ db \quad \wedge \quad \text{eval}\ t \quad \wedge \quad \text{fst}\ db \quad = \quad \text{sumUp}\ (\text{snd}\ db) \tag{2}$$

Finally we will formulate that $I\left(\boxed{\text{db}}\right)$ is an invariant property of `scenario` with respect to the initial property $Q(db,t)$.

$$I\left(\boxed{\text{db}}\right) \quad \in \quad inv_{scenario}\left(Q(db,t)\right) \tag{3}$$

Besides the Clean definitions and the description of the abstract object $\boxed{\text{db}}$, definitions (1,2) and statement (3) will be the input to the theorem prover.

5 The proof of the invariant property

The theorem prover for Clean which contains temporal logical tactics is not yet implemented. Hence the first step of the proof discussed in this section was performed by hand. This first step is the application of the not yet implemented tactic "inv" on our goal, namely statement (3), which results in subgoals (4–7). For the sake of readability we will use that definition of `scenario` which does not contain hiding of variables. Subgoal (4) describes that $I\Big(\boxed{\text{db}}\Big)$ holds initially, and subgoals (5–7) describe that $I\Big(\boxed{\text{db}}\Big)$ is preserved by each state transitions of $\boxed{\text{db}}$. None of the four subgoals contain temporal logical operators, so we can prove all of them with Sparkle. The details of these proofs can be found in [15].

The first resulting subgoal, subgoal (4), describes that $I\Big(\boxed{\text{db}}\Big)$ holds initially: precondition $Q(db, p)$ guarantees that the first (initial) state of $\boxed{\text{db}}$, namely db, satisfies I. Formally, we require that $Q(db, p) \Rightarrow I(db)$. If we unfold the definitions for Q and I, we obtain the following theorem.

$$(\text{eval } db) \wedge (\text{eval } t) \wedge \Big((\text{fst } db) = \big(\text{sumUp } (\text{snd } db)\big)\Big) \;\Rightarrow$$
$$(\text{eval } db) \wedge \Big((\text{fst } db) = \big(\text{sumUp } (\text{snd } db)\big)\Big) \tag{4}$$

This theorem is easy to prove. In Sparkle it takes 5 proof steps to complete this proof.

The state transitions for our abstract database object $\boxed{\text{db}}$ correspond to the application of functions `insertDB`, `sortDB` and `removeFirst`. Subgoals (5–7) describe that $I\Big(\boxed{\text{db}}\Big)$ is preserved by each of the three state transitions. We can express this with the help of the weakest precondition operator [9], *wp*. For each state transition *s*, the following should hold:

$$I\Big(\boxed{\text{db}}\Big) \;\Rightarrow\; wp\Big(s,\, I\Big(\boxed{\text{db}}\Big)\Big)$$

Refer to [5] to see how *wp* can be interpreted in this context. In subgoals (5–7) the temporal logical operator *wp* has already been eliminated.

The first state transition of $\boxed{\text{db}}$ corresponds to the following computation of the value `db1` from `db`: `db1 = insertDB db t`. Note that this computation is parametrized by another value, `t`, which is an argument of function `scenario`. In such a case we can introduce a hypothesis about `t`, based on the initial condition $Q(db,t)$ of the invariant property we are currently proving. This hypothesis tells us that *t* does not contain undefined parts: `eval t`. If `t` were not an argument of `scenario`, but a value defined inside `scenario`, we could introduce a hypothesis based on its definition. This case study, however, is lacking such an example.

$$(\text{eval } t) \wedge I(db) \wedge (db1 = \mathit{insertDB}\ db\ t) \quad \Rightarrow \quad I(db1) \tag{5}$$
$$I(db1) \wedge (db2 = \mathit{sortDB}\ db1) \quad \Rightarrow \quad I(db2) \tag{6}$$
$$I(db2) \wedge (db3 = \mathit{removeFirst}\ db2) \quad \Rightarrow \quad I(db3) \tag{7}$$

The proofs of goals (5–7) requires about 4000 proof steps in Sparkle.

6 Conclusions and future work

In this paper we have studied a method that allows the definition and proof of temporal properties (namely invariants) in pure functional languages. We have presented the concept of object abstraction, which is based on contracting functional variables into objects with dynamic (temporal) behaviour. We have introduced a notion of state transitions and we have illustrated on a very simple example how an invariant of an abstract object over a set of atomic state transitions can be proved. The proof was constructed in a theorem prover (Sparkle) except for a single proof step. The missing step applied a tactic (inv) that is not yet implemented in the theorem prover. The case study identified the method of producing subgoals during the application of this tactic. The implementation of this tactic will be based on this result.

The proof we have constructed in this case study is represented in a completely machine processable form. As a consequence, not only the program, but also its proved invariant property and the proof itself can be stored, transmitted or checked by a computer.

The case study in this paper does not address some issues which may be important for more sophisticated examples. We have considered a situation where the atomic state transitions of an abstract object were not distributed in more than one function definition. We will have to develop further case studies in which the analyzed function (like `scenario` in our example) contains "compound state transitions", namely it invokes other functions that contain more than one atomic state transitions themselves. The solution to this problem will be useful to handle recursive functions.

Our approach defines an alternative semantics of Clean programs. According to this alternative semantics, some evaluation steps correspond to state transitions over an abstract state space. The abstract state space is created by the object abstraction, where series of pure functional values are associated to an abstract objects. This methodology will be supported by an extension to Sparkle [4], the theorem prover tool for Clean, to make reasoning about temporal properties of interactive, parallel or distributed Clean programs possible. We plan to integrate the extended theorem prover into a program development and code manipulation/refactoring environment. Programs containing abstract objects will be presented to the theorem prover in a format similar to the one Sparkle currently uses to represent Clean programs and proofs, but extended with a representation of abstract object.

Our model is straightforward to extend to a full temporal logic. We can prove all temporal properties which are based on the "nexttime" operation, i.e. on the calculation of the weakest precondition [2, 10]. We intend to prove general safety properties (unless), and progress properties (leads-to, ensures) for Clean programs in the future.

References

[1] Achten, P., Plasmeijer, R.: Interactive Objects in Clean. *Proceedings of Implementation of Functional Languages, 9th International Workshop, IFL'97* (K. Hammond et al (eds)), St Andrews, Scotland, UK, September 1997, LNCS 1467, pp. 304–321.

[2] Chandy, K. M., Misra, J.: *Parallel program design: a foundation*. Addison-Wesley, 1989.

[3] Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Verification of the Temporal Properties of Dynamic Clean Processes. *Proceedings of Implementation of Functional Languages, IFL'99*, Lochem, The Netherlands, Sept. 7–10, 1999. pp. 203–218.

[4] de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem Proving for Functional Programmers, Sparkle: A Functional Theorem Prover. Springer Verlag, LNCS 2312, p. 55 ff., 2001.

[5] Horváth Z., Kozsik T., Tejfel M.: Proving Invariants of Functional Programs. *Proceedings of Eighth Symposium on Programming Languages and Software Tools*, Kuopio, Finland, June 17–18, 2003., pp. 115–126.

[6] Kozsik T.: Reasoning with Sparkle: a case study. *Technical Report*, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. (in preparation)

[7] Butterfield, A., Dowse, M., Strong, G.: Proving Make Correct: IO Proofs in Haskell and Clean. *Proceedings of Implementation of Functional Programming Languages*, Madrid, 2002. pp. 330–339.

[8] Dam, M., Fredlund, L., Gurov, D.: Toward Parametric Verification of Open Distributed Systems. *Compositionality: The Significant Difference* (H. Langmaack, A. Pnueli, W.-P. De Roever (eds)), Springer-Verlag 1998.

[9] Dijkstra, E. W.: *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs (N.Y.), 1976.

[10] Horváth Z.: The Formal Specification of a Problem Solved by a Parallel Program—a Relational Model. *Annales Uni. Sci. Bp. de R. Eötvös Nom. Sectio Computatorica*, Tom. XVII. (1998) pp. 173–191.

[11] Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Proving the Temporal Properties of the Unique World. *Proceedings of the Sixth Symposium on Programming Languages and Software Tools*, Tallin, Estonia, August 1999. pp. 113–125.

[12] Kozsik T., van Arkel, D., Plasmeijer, R.: Subtyping with Strengthening Type Invariants. *Proceedings of the 12th International Workshop on Implementation of Functional Languages* (M. Mohnen, P. Koopman (eds)), Aachener Informatik-Berichte, Aachen, Germany, September 2000. pp. 315–330.

[13] Peyton Jones, S., Hughes, J., et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, February 1999.

[14] Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001. http://www.cs.kun.nl/~clean/Manuals/manuals.html

[15] Tejfel M.: The Problem of Proof Reuse in Sparkle: a case study. *Technical Report*, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. (in preparation)

[16] Home of Clean. http://www.cs.kun.nl/~clean/

[17] Diviánszky P., Szabó-Nacsa R., Horváth Z.: Refactoring via Database Representation. *Proceedings of 6th International Conference on Applied Informatics, ICAI 2004*, January 27–31, Eger, Hungary, 2004.