Proving Invariants of Functional Programs*

Zoltán Horváth, Tamás Kozsik, Máté Tejfel

Eötvös Loránd University
Department of General Computer Science
{hz,kto,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. In this paper we present the concept of object abstraction and show how to interpret and prove temporal properties of functional programs.

1 Introduction

When proving correctness of (sequential or parallel) imperative programs, one can make use of several temporal logical operators. Some well-known such operators are e.g. "nexttime", "sometimes", "always" and "invariant". All these operators can be expressed based on the "weakest precondition" operator [6, 7]. However, temporal logical operators are less frequently used when reasoning about functional programs (among the few exceptions are e.g. [5, 8–10]). This paper aims to answer the question how to interpret and prove temporal properties of functional programs.

The temporal logical operators describe how the values of the program variables (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 postcondition according to the substitution rules defined by the statement. We will show some examples of how this is done in section 3.1.

A property P is an invariant with respect to a program if P holds initially 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. This ensures that if we execute an atomic

* Supported by OTKA T037742

statement in a state where P holds (and hence the weakest precondition of the statement with respect to P also holds), the statement will again terminate in a state for which P holds. We will formalize this in the following way. (The weakest precondition operator is denoted by wp, while a program and its atomic statements are denoted by S and s, respectively.)

$$\forall s \in S : \quad P \Rightarrow wp(s, P)$$

Invariants can manifest in many ways: we can talk about loop invariants (where the atomic statement is a loop body), type invariants (where the atomic statements are the primitive operations of a type), and invariants are an important concept in many parallel programming methodologies as well (see e.g. [4]). We believe that invariants are popular because they provide a very natural concept and a very useful abstraction for specifying and proving properties of programs. This paper focuses on invariant properties, hence we omit the description of the other afore-mentioned temporal logical operators.

When proving correctness of functional programs, the practicability of temporal operators is not obvious. 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. [11]). 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.

We have chosen Clean [13], a lazy, pure functional language for our research. An important factor in our choice was that a theorem prover, Sparkle [11] 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.

The "uniqueness type system" of Clean [2] makes destructive updates possible without violating referential transparency. The uniqueness type system guarantees that certain values are only used once in the program (they are unique), hence they can be destructively updated when computing other values. This technique is used to define I/O in Clean, furthermore it greatly increases the efficiency of Clean programs. It is interesting to see that in many cases, an abstract object of our approach corresponds to a set of unique values: values that were computed from each other by destructive updates. Hence the abstract view of objects often—but not always—coincides with the memory layout of the implementation.

The rest of the paper is organized in the following way. In Section 2 we introduce our approach through a simplistic example. Then in Section 3 we present a formal method for the calculation of weakest preconditions and for the proof of invariant properties. Next, in Section 4, we give a more realistic example of object abstraction, and show a simple invariant property of an abstract object. Section 5 presents the proof of a more interesting invariant property. Finally, in section 6, we draw the conclusions and define future work.

2 Object abstraction

In Clean the uniqueness type system makes destructive updates possible without violating referential transparency. Not only the efficiency of Clean programs can be increased by destructive updates, but also the I/O system of Clean is defined in terms of a "unique environment". (The other well-known technique to define pure functional I/O is the monadic approach, applied in the language Haskell [12].) The Object I/O library [1] is a standard API for Clean. Programs written with the Object I/O library are reactive. They create a unique state space (referred to as "process state" and "local state" in Object I/O terminology), and define initialization and state transition functions. The library supports *interactive processes*, which can be created and closed dynamically. Each interactive process may consist of an arbitrary number of *interactive objects*. To characterize the behavior of I/O processes we can use a temporal logic-based notation [4,7]. We have researched this issue in [8,9].

This paper investigates a more general approach to formulating and proving temporal properties of functional programs. In this approach we can also reason about Clean programs that do not use unique values or interactive Object I/O processes. Not only the call-back functions of Object I/O will be state transition functions: the programmer can demarcate state transitions explicitly in a more flexible way. Different values computed by a functional program and stored in variables (in the functional sense of variables) can be regarded as different states of the same object. State transitions will thus be the pieces of functional code that compute such a value from another one.

Our first example, though it might seem oversimplified, illustrates well our concept of *object abstraction*. Suppose we want to sort (in ascending order) a list containing five numbers. We will make use of the function `bubble`. This function searches for two elements in a list that are in wrong order. If it can find such two elements, it swaps them and returns the resulting list, otherwise it returns the original list. Obviously, it is sufficient to invoke `bubble` 25 times to sort five numbers.

```
sort_5 list = bubble (bubble (bubble (... (bubble list)...)))
```

We can write the same program in Clean using so-called let-before expressions. Both of the following function definitions are legal in Clean and have the same meaning. The one on the right uses the same name, `list`, to all 24 variables. In this case the usual static nested scoping rules apply. (The line "`# list = bubble list`" introduces a nested scope with a fresh `list` variable, thus hiding the variable with the same name appearing on the right-hand side.)

```
sort_5 list                          sort_5 list
   # ls1 = bubble list                  # list = bubble list
   # ls2 = bubble ls1                   # list = bubble list
   # ls3 = bubble ls2                   # list = bubble list
   ...                                  ...
   # ls23 = bubble ls22                 # list = bubble list
   = bubble ls23                        = bubble list
```

An important property of the `bubble` function is that the list it returns is a permutation of the list it receives. If we regard the list values computed in `sort_5` as the successive states of the same abstract object, and consider `bubble` an atomic operation, we can formulate an invariant, a temporal property of `sort_5`. If we denote our abstract list object with $\boxed{\text{list}}$, then the invariant can be written as:

$$\boxed{\text{list}} \in \texttt{perm}(list).$$

Here $list$ denotes the argument of `sort_5`.

Consider now the second definition of `sort_5`, the one that contains the variables `ls1`, `ls2`, `ls3`, etc. Our approach should allow us to declare that the abstract object $\boxed{\text{list}}$ is made up of the values `list`, `ls1`, `ls2`, ..., `ls23` and `bubble ls23`. Furthermore, it should allow us to declare that `bubble` is considered atomic and that each invocation of `bubble` in `sort_5` is a state transition.

2.1 The object abstraction operator

We introduce an operator which maps functional values to our semantic domain, the state space. This "object abstraction" operator will be used to refer to the abstract object to which a value belongs. In our sorting example $\boxed{\text{list}}$ will refer to the same abstract object as e.g. $\boxed{\text{ls1}}$. This can be expressed by the equation

$$\boxed{\text{list}} \quad = \quad \boxed{\text{ls1}}.$$

(Hence the object abstraction operator will define an equivalence relation over the values appearing in a functional program.) Moreover, the successive values of the same abstract object will define an abstract time structure denoted by the partial ordering $<_t$.

$$list <_t ls1 <_t ls2 <_t \ldots <_t ls23 <_t (bubble\ ls23)$$

If more than one object is present in a certain piece of code, then the states are compositions of the individual states of the objects. In such cases two or more objects can be involved in an atomic state transition over this compound state space.

Since evaluation is lazy in Clean, the partial ordering $<_t$ determines a branching time structure over the state space. Fortunately, we do not have to refer explicitly to the $<_t$ relation, because we use temporal logical operators instead.

2.2 Identification of state transitions

It would be useful to assign symbolic names to pieces of code which correspond to atomic state transitions. We introduce an infix binary operator for labeling, which does not have any influence on computation: the operator "`.:`" simply drops its first argument, the label. For example, we can label a state transition `"b"` in the following way:

```
# list = "b" .: (bubble list)
```

This let before definition determines the value of `list` on the left-hand side depending on the value of `list` on the right-hand side. The two `list`-s are different from each other, the `list` on the left-hand side hides (within its scope) the `list` on the right hand side. We may consider the two `list`-s (two different functional entities) belonging to the same abstract object $\boxed{\text{list}}$. According to this object abstraction the `list` on the left-hand side represents a descendant of the value of `list` on the right-hand side. The let-before definition labeled by `"b"` represents an atomic state transition.

3 Formal calculations

In this section we explain how to calculate the weakest precondition of an atomic action with respect to a postcondition, and how to prove invariant properties of programs. Consider the following piece of code, which increases a value by one modulo 5.

```
# v = "f" .: (if (v<5) (inc v) 0)
```

Let our state space consist of a single component, the state represented by object $\boxed{\text{v}}$. Both values denoted by `v` are associated with this object. Moreover, we assume that the state transition labeled with `"f"` is atomic.

3.1 Formal calculation of weakest precondition

Let us consider the following postcondition R:

$$R(\boxed{\text{v}}) \;=\; (0 \leq \boxed{\text{v}} < 5).$$

We are interested in characterizing all states from where the atomic state transition "f" terminates in a state for which R holds, i.e. we would like to determine the weakest precondition of "f" with respect to R.

If $v \geq 5$, then the new v value will be equal to 0, hence the new value of the abstract object $\boxed{\text{v}}$ will be 0. If $v < 5$, then the new v is calculated by incrementing the old v, hence the state of $\boxed{\text{v}}$ is changed to $\boxed{\text{v}} + 1$. The postcondition holds for the new value, if the weakest precondition calculated below holds for the original state. This example illustrates the general method that we can use to calculate the weakest precondition: in the postcondition we should substitute the old value of the object with its new value.

$$\begin{aligned}&wp(f, R) = \\ &\quad (\boxed{\text{v}} < 5 \;\rightarrow\; 0 \leq inc(\boxed{\text{v}}) < 5) \quad \wedge \quad (\boxed{\text{v}} \geq 5 \;\rightarrow\; 0 \leq 0 < 5)\end{aligned}$$

The new value used in the substitution is the right-hand side of the definition of the function which function is applied on the old value of the abstract object. This way the calculation of the weakest precondition is a simple rewriting step, which fits very well into the world of functional computations.

3.2 Proving invariants

Proving that a property P is an invariant requires two things. First, one has to check whether the initial values of the objects satisfy P. Next, one has to calculate the weakest precondition for all atomic state transitions: for each such atomic state transition one has to compute the substitution of P using the corresponding state transition function. Then one should prove that all these wp-s hold, if P holds.

Now we show how to prove that the atomic step "f" preserves the truth of R, i.e. $R \Rightarrow wp(f, R)$. We have to prove by hand or by a proof assistant (e.g. Sparkle) the following theorem:

$$\begin{aligned}&0 \leq \boxed{\text{v}} < 5 \;\;\Rightarrow \\ &\quad (\boxed{\text{v}} < 5 \;\rightarrow\; 0 \leq inc(\boxed{\text{v}}) < 5) \quad \wedge \quad (\boxed{\text{v}} \geq 5 \;\rightarrow\; 0 \leq 0 < 5).\end{aligned}$$

To complete the proof we have to apply the definition of inc—which is a rewriting step again—and then use the well-known deduction rules of classical logic.

4 A more realistic example

Consider now a more complex example. The analyzed Clean function will be a binary search. It takes an array of elements of type “a” (where “a” is a type variable expressing polymorphism) and a value of type "a". It returns either `Nothing`, if the given value could not be found in the array, or `(Just h)`, if the given value was found at position `h` in the array. The array is unique (denoted by the `*` symbol in the type specification of the function), that's why `bin_search` also returns a new unique reference to it. The implementation of unique arrays is very similar to objects in imperative languages, in the sense that the new array is stored in the same memory location where the old array was stored. This is possible because uniqueness guarantees that there are no more references to the old array.

```
bin_search :: *{a} a -> (Maybe Int, *{a}) | Ord, Eq a
bin_search arr e
   # (s, arr) = usize arr
   = find_it arr 0 (s-1)
   where find_it arr u v
              | u > v
                  = (Nothing, arr)
              | otherwise
                  # h = (u+v)/2
                  # (arr_h, arr) = uselect arr h
                  | arr_h == e
                       = (Just h, arr)
                  | otherwise
                       # (u,v) = if (arr_h<e) (h+1,v) (u,h-1)
                       = find_it arr u v
```

Functions `usize` and `uselect` are from the standard library. They can be used to retrieve the size and an element of a unique array, respectively.

Now let us apply `bin_search` on an array `arr` and an element `e`.

```
(h,arr1) = bin_search arr e
```

Note that the binary search algorithm requires as precondition that its first argument is a sorted array. We will denote it with the following formula:

$$sorted(arr)$$

We introduce the abstract object $\boxed{\text{arr}}$ from values `arr` and `arr1`, where `arr` $<_t$ `arr1`. First we would like to prove a trivial invariant property of this object, namely that `bin_search` does not change the unique array.

$$P(\boxed{\text{arr}}) \quad = \quad (\boxed{\text{arr}} = arr), \qquad P \in inv$$

We will not consider `bin_search` an atomic state transition, hence we will dive into its definition. We identify one more state of the $\boxed{\text{arr}}$ object, namely when $\boxed{\text{arr}}$ has the value `arr` returned by `usize`. Now the second state transition of $\boxed{\text{arr}}$ changes this second occurrence of `arr` to `arr1` by applying `find_it`. The two state transitions are the following:

```
(s,arr) = usize arr
(h,arr1) = find_it arr 0 (s-1)
```

In order to prove that P is an invariant of `bin_search`, we will prove that it is also an invariant of `find_it`. Again, we will not consider the second state transition atomic, hence we will dive into the definition of `find_it`. Since `find_it` is defined as an alternative construct with two branches, we will replace our second state transition with two other state transitions: one corresponding to the $u > v$ case, the other one corresponding to the `otherwise` (that is the $\neg(u > v)$) case. This latter can be further refined, until we obtain the following six state transitions of $\boxed{\text{arr}}$, which we will not intend to further refine. (Irrelevant results of state transitions are replaced with the joker character underscore.)

s_1: `(_,arr) = usize arr`
s_2: if $u > v$, then `(_,arr1) = (Nothing, arr)`
s_3: if $\neg(u > v)$, then `(_,arr1) = uselect arr h`
s_4: if $\neg(u > v) \wedge arr\_h = e$, then `(_,arr1) = (Just h, arr)`
s_5: if $\neg(u > v) \wedge \neg(arr\_h = e)$, then `(_,arr1) = find_it arr u v`
(Notice that the variables u and v in the predicate and in the formal arguments of `find_it` represent two different values with the same name.)

Note that the last state transition is the recursive application of `find_it` and therefore can be cut. We will consider the remaining 4 state transitions atomic. To prove that ($\boxed{\text{arr}} = arr$) is an invariant of `bin_search` and `find_it` with respect to the atomicity level described above, we show that all these atomic state transitions preserve this property and that this property holds for the initial value of $\boxed{\text{arr}}$. The first part is fairly simple: s_2 and s_4 apply the identity function on the array, while s_1 and s_3 apply `usize` and `uselect`, which again do not change the value of the abstract object. (For this latter we must formulate axioms about these two standard library functions.) Finally, we should prove that the initial value of $\boxed{\text{arr}}$, namely arr satisfies ($\boxed{\text{arr}} = arr$), which is obvious, since this requires that arr should be equal to itself.

5 A more interesting invariant property of binary search

If we want to prove the partial correctness of `bin_search`—namely that `e` does not appear in `arr`, if "`bin_search e arr`" returns `Nothing`, and that `e` can be found in `arr` at position `h`, if "`bin_search e arr`" returns "`Just h`"—we can make use of some further invariants of our program. We can identify new abstract objects and express our assumptions about them in terms of invariants. New abstract objects and new invariants describing their behaviour are introduced usually when we dive into a function invocation. In our example this happens at the point where `find_it` is applied in `bin_search`.

Hence we extend the state space with $\boxed{\text{u}}$ and $\boxed{\text{v}}$: these objects specify the interval where `find_it` looks for the value `e` in $\boxed{\text{arr}}$. We are about to formulate invariants expressing that `e` cannot be found in $\boxed{\text{arr}}$ at a position outside the interval $[\boxed{\text{u}}..\boxed{\text{v}}]$.

First of all let us describe more precisely the object abstraction for $\boxed{\mathrm{u}}$ and $\boxed{\mathrm{v}}$. Their initial values are 0 and $s-1$, respectively, according to the application of `find_it` within `bin_search`. (Note that the value s comes from `usize`$\boxed{\texttt{arr}}$, thus it is the length of the array.) Furthermore, all occurrences of the second argument of `find_it` correspond to the $\boxed{\mathrm{u}}$ object, and all occurrences of the third argument of `find_it` correspond to the $\boxed{\mathrm{v}}$ object.

The extension of the state space—and the introduction of new abstract objects—is often followed by the refinement of the time structure $<_t$. For example, in `bin_search` we can cut the state transition s_5 (previously considered atomic) into two steps. The two new state transitions replacing s_5 will be considered atomic steps from now on. They are the following:

s_{5a}: if $\neg(u > v) \wedge \neg(arr\_h = e)$, then
    `(u,v) = if (arr_h<e) (h+1,v) (u,h-1)`
s_{5b}: if $\neg(u > v) \wedge \neg(arr\_h = e)$, then `(_,arr1) = find_it arr u v`

Our refined invariant, P' will be the conjunction of four parts. The first part is the original P, which states that the initial value of the array object is preserved. The second part specifies that the array remains sorted. The third part states that the interval identified by $\boxed{\mathrm{u}}$ and $\boxed{\mathrm{v}}$ is either empty or part of the domain of the array. Finally, the fourth part claims that the element we are looking for is not outside the $[\boxed{\mathrm{u}}, \boxed{\mathrm{v}}]$ interval. The free variables x, y and i in the following formulas are implicitly universally quantified.

$$P'(\boxed{\mathrm{arr}}, \boxed{\mathrm{u}}, \boxed{\mathrm{v}}) \quad = $$
$$P(\boxed{\mathrm{arr}}) \;\wedge\; P_0(\boxed{\mathrm{arr}}) \;\wedge\; P_1(\boxed{\mathrm{arr}}, \boxed{\mathrm{u}}, \boxed{\mathrm{v}}) \;\wedge\; P_2(\boxed{\mathrm{arr}}, \boxed{\mathrm{u}}, \boxed{\mathrm{v}})$$

where

$$P_0(\boxed{\mathrm{arr}}) \quad = \quad sorted(\boxed{\mathrm{arr}})$$

$$P_1(\boxed{\mathrm{arr}}, \boxed{\mathrm{u}}, \boxed{\mathrm{v}}) \quad =$$
$$\Big(\big((x,y) = \mathrm{usize}\,\boxed{\mathrm{arr}}\big) \wedge \big(\boxed{\mathrm{u}} \le \boxed{\mathrm{v}}\big)\Big) \quad \rightarrow \quad 0 \le \boxed{\mathrm{u}} \wedge \boxed{\mathrm{v}} < x$$

$$P_2(\boxed{\mathrm{arr}}, \boxed{\mathrm{u}}, \boxed{\mathrm{v}}) \quad =$$
$$\Big(\big((x,y) = \mathrm{uselect}\,\boxed{\mathrm{arr}}\ \mathrm{i}\big) \wedge \big(i < \boxed{\mathrm{u}} \vee i > \boxed{\mathrm{v}}\big)\Big) \quad \rightarrow \quad \neg(x = e)$$

To prove that P' is an invariant of `find_it` we have to check whether the initial values of the objects satisfy P', and we have to show that P' guarantees the weakest precondition of P' for all the atomic actions s_2, s_3, s_4, s_{5a} and s_{5b} of `find_it`. Note that s_1 need not be considered, since this state transition is outside of `find_it`, and hence the scope of P'. However, we still have to prove for this state transition (and also for the others, s_2, s_3, s_4, s_{5a} and s_{5b})

the relevant, weaker invariant $P \wedge P_0$, because this invariant will be used as a precondition during the proof of "$(P' \in inv_{find\_it})$". We omit the proof for "$(P \wedge P_0 \in inv_{bin\_search})$" here, since it is fairly simple; we will focus on `find_it` and P'.

The precondition Q for `find_it` is the following: $P \wedge P_0 \wedge \boxed{\text{u}} = 0 \wedge \boxed{\text{v}} = s-1$. This should guarantee P', that is $Q \Rightarrow P \wedge P_0 \wedge P_1 \wedge P_2$. The interesting part is that $Q \Rightarrow P_1$ and $Q \Rightarrow P_2$. If we rewrite P_1 and P_2 according to the equalities found in Q, we obtain the following formulas:

$$
\begin{aligned}
&P_1(arr, 0, s-1) \quad = \\
&\quad \Big(\big((x,y) = \text{usize arr}\big) \wedge \big(0 \leq s-1\big) \Big) \quad \rightarrow \quad 0 \leq 0 \wedge s - 1 < x
\end{aligned}
$$

$$
\begin{aligned}
&P_2(arr, 0, s-1) \quad = \\
&\quad \Big(\big((x,y) = \text{uselect arr i}\big) \wedge \big(i < 0 \vee i > s-1\big) \Big) \quad \rightarrow \quad \neg(x = e)
\end{aligned}
$$

The first formula is valid, because the value of x appearing in the formula must be equal to `s`, the size of `arr` obtained in s_1. (Here we have to use the definition of the functional variable `s`, namely `(s,_) = usize arr`.) The second formula is also valid, since the left-hand side of the implication cannot hold. (The standard library function `uselect` is undefined when applied to an array `arr` and an index `i` outside of the domain of `arr`. This can be expressed by the following axiom: $(p,q) = usize\ r \quad \rightarrow \quad (x,y) = uselect\ q\ i \quad \rightarrow \quad 0 \leq i < p$.)

Now let us prove that $P' \Rightarrow wp(f, P')$ for all state transitions "f" from s_2, s_3, s_4, s_{5a} and s_{5b}. If we have already proved $P \wedge P_0 \Rightarrow wp(f, P \wedge P_0)$, as mentioned earlier, then we only need to show that $P' \Rightarrow wp(f, P_1 \wedge P_2)$. (This is a nice property of refined invariants.) The proofs for s_2, s_3 and s_4 are trivial, hence we omit these cases. Moreover, s_{5b} is a recursive application of `find_it`, therefore this case can be cut. Hence we focus on the single interesting part: $P' \Rightarrow wp(s_{5a}, P_1 \wedge P_2)$. According to the well-known conjunctivity property of wp, we can perform the proof separately for the weakest precondition of P_1 and for that of P_2. Furthermore, since s_{5a} contains an `if` construct, the proofs for the two branches of the `if` can be separately given. Hence we obtain four goals to prove. Here we give the ones for P_1; the other two are similar, just replace P_1 with P_2. By calculation of the weakest precondition we get:

$$
\begin{aligned}
&P'(\boxed{\text{arr}}, \boxed{\text{u}}, \boxed{\text{v}}) \wedge \neg(\boxed{\text{u}} > \boxed{\text{v}}) \wedge \neg(arr\_h = e) \wedge (arr\_h < e) \Rightarrow \\
&\quad P_1(\boxed{\text{arr}}, h+1, \boxed{\text{v}})
\end{aligned}
$$

$$
\begin{aligned}
&P'(\boxed{\text{arr}}, \boxed{\text{u}}, \boxed{\text{v}}) \wedge \neg(\boxed{\text{u}} > \boxed{\text{v}}) \wedge \neg(arr\_h = e) \wedge \neg(arr\_h < e) \Rightarrow \\
&\quad P_1(\boxed{\text{arr}}, \boxed{\text{u}}, h-1).
\end{aligned}
$$

As an illustration, we present the proof for the first goal for P_1. First let us expand the formula $P_1(\boxed{\text{arr}}, h+1, \boxed{\text{v}})$, which we have obtained after the substitution of $\boxed{\text{u}}$ by $h + 1$:

$$P_1(\boxed{\text{arr}}, h+1, \boxed{\text{v}}) \quad = $$
$$\Big(\big((x,y) = \text{usize}\,\boxed{\text{arr}}\big) \wedge \big(h+1 \leq \boxed{\text{v}}\big)\Big) \quad \rightarrow \quad 0 \leq h+1 \wedge \boxed{\text{v}} < x.$$

Notice that the only non-trivial part of the proof is to show that under the appropriate hypotheses $0 \leq h+1$. Remember that the functional variable h denotes $(\boxed{\text{u}} + \boxed{\text{v}})/2$, where the symbol / is the division operator on integer numbers (e.g. $3/2 = \lfloor \frac{3}{2} \rfloor = 1$ and $(-3)/2 = \lceil \frac{-3}{2} \rceil = -1$). Hence the proof can be easily accomplished by applying the following three lemmas on $\boxed{\text{u}}$ and $\boxed{\text{v}}$:

$$\left\lceil \frac{x+y}{2} \right\rceil + 1 \leq y \;\Rightarrow\; \left\lfloor \frac{x+y}{2} \right\rfloor + 1 \leq y \;\Rightarrow\; x \leq \left\lfloor \frac{x+y}{2} \right\rfloor + 1 \;\Rightarrow\; x \leq \left\lceil \frac{x+y}{2} \right\rceil + 1$$

The proof of the second goal for P_1 is symmetrical to this proof. Finally, the proofs of the two goals for P_2 make use of the hypothesis $P_0(\boxed{\text{arr}})$, which formulates that $\boxed{\text{arr}}$ is sorted. These proofs are left to the reader as an exercise.

6 Conclusions and future work

In this paper we have presented a method that allows the definition and proof of temporal properties (namely invariants) in pure functional languages. We have introduced the concept of object abstraction by contracting functional (that is mathematical) variables, which represent static values, into objects with dynamic (temporal) behaviour. According to this concept we could define an abstract time structure in programs, representing the computational dependencies of values (object states) on other values (other object states). We have also introduced the notion of state transitions. We have illustrated how invariants over a set of atomic state transitions can be computed and how this process can be automatized.

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. The evaluation order is non-deterministic in case of lazy evaluation, so the transition steps determine a branching time structure over the elements of the state space.

The mapping of values to objects and the labeling of state transitions can be performed by using annotations and/or supported by an appropriate user interface integrated into the proof assistant. Furthermore, objects and state transitions can be extracted from a functional program written in an appropriate style in an automated way.

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 [4, 7]. We intend to prove general safety properties (unless), and progress properties (leads-to, ensures) for Clean programs in the future. This methodology could be supported by an extension to Sparkle [11], the theorem prover tool for Clean, to make reasoning about temporal properties of interactive, parallel or distributed Clean programs possible.

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. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Comp. Sci.* 6, pp. 579–612. 1996.
3. 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.
4. Chandy, K. M., Misra, J.: *Parallel program design: a foundation*. Addison-Wesley, 1989.
5. 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.
6. Dijkstra, E. W.: *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs (N.Y.), 1976.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Peyton Jones, S., Hughes, J., et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, February 1999.
13. Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001. http://www.cs.kun.nl/~clean/Manuals/manuals.html