Acta Cybernetica **17** (2005) 419–445.

Extending the Sparkle Core language with object abstraction*

Máté Tejfel*, Zoltán Horváth*, and Tamás Kozsik†

Abstract

Sparkle is a theorem prover specially constructed for the functional programming language Clean. In a pure functional language like Clean the variables represent constant values; variables 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 I/O), a series of values computed from one another can be considered as different states of the same "abstract object". For this abstract object temporal properties can be proved. This paper presents a method to describe abstract objects and invariant properties in an extended version of the Sparkle Core language. The creation of such descriptions will be supported by a refactoring tool. The descriptions are completely machine processible, and provide a way to automatize the proof of temporal properties of Clean programs with the extended Sparkle system.

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

The temporal logical operators describe how the values of the program variables (the so-called program state) vary in time. They are very useful for proving correctness of (sequential or parallel) imperative programs. 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 [7, 13].

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

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

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

terminates in a state for which the postcondition holds. We can compute the weakest precondition of a statement in an automated way: we have to rewrite the postcondition according to the substitution rules defined by the statement.

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 [22]). This fact is one of the basic advantages of functional programs over imperative ones.

However, in certain cases it is natural to express our knowledge about the behaviour of a functional program (or rather 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.

In the authors' 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.

Clean [24], a lazy, pure functional language was chosen for this research. An important factor in our choice was that a theorem prover, Sparkle [22] is already built in the integrated development environment of Clean. Sparkle supports reasoning about Clean programs almost directly. The authors extended the basic logic used by Sparkle with temporal operators.

Earlier, correctness proofs about interactive, concurrent (interleaved) Clean programs, namely Object IO processes have been provided in [14, 15]. However, these proofs were carried out by hand. The authors argue that the extension of the theorem prover with tools supporting temporal logical operators facilitates the reasoning about interactive, concurrent or distributed (see [16]) Clean programs, since temporal logical reasoning can be performed within the theorem prover.

For formulating and proving temporal properties of a Clean program, the "abstract objects" have to be determined, that is it has to be specified which functional (mathematical) values correspond to different states of the same abstract object. Furthermore, state transitions should also be expressible. Therefore, Clean and correspondingly the Sparkle Core language have to be extended with some new syntactical elements. This paper aims to present these extensions, to show how one can describe abstract objects in these extended languages, to give the semantics of the introduced language extensions, and to illustrate by means of some simple examples that temporal reasoning is really useful for functional programs.

The rest of the paper is organized in the following way. In Section 2 the object abstraction method is presented through a simplistic example. Then Section 3 describes an extension to the Clean language. Section 4 introduces the new language constructs into the Sparkle Core language. Section 5 explains how temporal

propositions can be expressed in the extended Sparkle framework. Next, Section 6 presents some more complex and more useful examples of temporal properties and their proofs. Finally, in Section 7, the conclusions are drawn and future work is defined.

2 Object abstraction

In Clean the uniqueness type system makes destructive updates possible without violating referential transparency. In the case of unique values temporality has a similar meaning as in imperative languages: unique values encode states. Destructive updates are not merely used to increase the efficiency of Clean programs, but the I/O system of Clean is also defined in terms of state transitions over a "unique environment". (The other well-known technique to define I/O in a pure functional language is the monadic approach, applied in the language Haskell [23].)

Programs written with the Object I/O library (a standard API for Clean) [1] are reactive. They create a unique state space and define initialisation 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*. Since I/O processes may run in parallel (in an interleaved manner), their behaviour can be described with a temporal logical machinery [4, 13] (in contrast to e.g. [3, 10, 11]). The authors have researched this issue in [14, 15].

This paper exploits a more general method, discussed before in [17, 18]. In this methodology one can reason about temporal properties of Clean programs even if they do not use unique values or interactive Object I/O processes. Not only some call-back functions of Object I/O can 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. A state transition will thus be a piece of functional code that computes such a value from another one.

The following simple example introduces shortly the object abstraction method. The example program `sort3` puts three integer values in increasing order. It uses another function `sort2`, which, in turn, puts two integer values in increasing order.

```
sort3 a b c
   # (a, b) = sort2 a b
   # (b, c) = sort2 b c
   # (a, c) = sort2 a c
   = (a, b, c)
```

According to the scoping rules in Clean, this program is equivalent to the following one:

```
sort3 a1 b1 c1
   # (a2, b2) = sort2 a1 b1
   # (b3, c2) = sort2 b2 c1
   # (a3, c3) = sort2 a2 c2
   = (a3, b3, c3)
```

Here the values a_1, a_2 and a_3 may be associated to the same abstract object, e.g. obj_1. Similarly, the values b_i and c_i may be associated to obj_2 and obj_3, respectively. The let-before expressions (denoted by `#`) will hence become the state transitions (atomic actions) of this program. Clean has to be extended with new syntactical elements so that one can express this kind of "object abstraction".

Although this example may seem too simplistic, it illustrates well the technique used for introducing objects. For more complex examples the same technique can be used, but the propositions may become less readable and the proofs substantially longer. To address the first issue, the authors are planning to develop a tool that provides support for object abstraction on a graphical user interface. Managing the second issue requires the use of predefined libraries of domain-specific lemmas.

3 Extending Clean with object abstraction

For the demarcation of "abstract objects" two new language constructs are needed. One of the constructs will be used to define which values (functional variables) correspond to different states of the same abstract object. The other construct will mark the state transitions of the program: in each state transition, one or more objects may change their values. State transitions will be regarded as atomic actions with respect to the temporal logical operators, and will be referred to as "steps" in the forthcoming sections.

The first construct will be denoted by "`.|.`". It has two arguments: an object identifier and a value identifier, like in "` .|. object_id value_id `". This means that the value identified by `value_id` is associated to the abstract object identified by `object_id`, or, for short, `value_id` identifies a state of `object_id`. The second construct, used for marking steps, is similar to the let-before (`#`) construct of Clean, hence a similar syntax has been chosen for that: "`.#.`".

The Clean syntax has been extended with the two constructs in the following way. The original definition of `Variable` has been changed to include an alternative "`Object`".

```
Variable = LowerCaseId
         | Object

Object = .|. LowerCaseId LowerCaseId
```

Therefore, an `Object` can be used wherever a `Variable` can be used in (the original) Clean, under the following conditions. A function definition may introduce objects, only if the body of the function is made up of let-before constructs. (The current

implementation of the extended Sparkle system can handle multiple function alternatives, but does not allow objects in functions with guards and rule alternatives.) The objects are local to the function definition, and the same object name refers to the same object in this scope. (Defining multiple objects with the same name within the same function definition is disallowed.) Objects can only be used in "steps". In every binding within a step (a so called *StepBind*, see later), the same object can appear at most once on the left, and at most once on the right-hand side. Obviously, the variables constituting the states of an abstract object must be of the same type. Finally, it has to be noted that currently only objects within a single function definition are supported: the variables that make up the states of the object must be defined in the same function. (In the near future the authors plan to develop an enhanced version of extended Clean and extended Sparkle in which object abstraction is not restricted to happen within the boundaries of a single function.)

The rule for `LetBeforeExpression` has also been extended with a new alternative, `StepExpression`, to support the second introduced new construct. A `StepExpression` can be used in the extended Clean wherever a `LetBeforeExpression` can be used in Clean.

```
LetBeforeExpression = # {GraphDef}+
                    | #!{GraphDef}+
                    | StepExpression

StepExpression = .#. {GraphDef}+
```

The example program expressed in the extended Clean language is the following.

```
sort3 (.|. obj1 a1) (.|. obj2 b1) (.|. obj3 c1)
   .#. ((.|. obj1 a2), (.|. obj2 b2)) = sort2 (.|. obj1 a1) (.|. obj2 b1)
   .#. ((.|. obj2 b3), (.|. obj3 c2)) = sort2 (.|. obj2 b2) (.|. obj3 c1)
   .#. ((.|. obj1 a3), (.|. obj3 c3)) = sort2 (.|. obj1 a2) (.|. obj3 c2)
    =  ((.|. obj1 a3), (.|. obj2 b3), (.|. obj3 c3))
```

This program text is much harder to read than the original Clean program. Note, however, that the extended Clean language is just an intermediate language used between two programs. A refactoring tool [25], integrated into an interactive software development environment, will be used to produce extended Clean code. The input to the refactoring tool is the original Clean code, and *interactive* instructions from the programmer regarding which values belong to which objects. The extended Clean code will be processed by a proof system, namely an extended version of Sparkle.

The semantics of Clean can be expressed in terms of Sparkle Core [21]. The Sparkle Core language is a part of the formal framework used by the Sparkle theorem prover, which has been developed for reasoning about Clean programs. Essentially, Sparkle Core corresponds to the internal representation of Clean programs in the Clean compiler. It is possible to express the semantics of the extended Clean

language in terms of a *variant* of Sparkle Core. The next sections present how Sparkle Core was modified to support extended Clean, and how Sparkle is to be modified to enable formulating and proving invariants of abstract objects.

4 Extending the Sparkle Core language

In order to adapt Sparkle for reasoning about temporal properties of abstract objects, the constructs `.|.` and `.#.` of the extended Clean language have been introduced into Sparkle Core. Here an incomplete description of this extended version of Sparkle Core is provided for the interested Reader, focusing only on the new elements added to Sparkle Core. An informal explanation of these new elements are also given, but for further details on the formal syntax and semantics of Sparkle Core the Reader is referred to [21]. (For the sake of readability, at certain points the notations of Sparkle Core have been simplified.)

The expressions ($\mathcal{E}$) of Sparkle Core are made up of variables, basic values, (function, delta-rule and constructor) symbols, applications, case-expressions, (lazy and strict) let-expressions, and (typed) "undefined" expressions. Two more alternatives have been introduced: object definitions and steps. Object definitions associate object identifiers (from $\mathcal{O}$) to expression-variable identifiers (from $\mathcal{V}_e$). Steps are similar to let expressions, they contain bindings of local expression variables ($\mathcal{V}_e^\bullet$) and objects to expressions. The resulting definition of expressions, $\mathcal{E}^{ext}$, is as follows. (The new elements are framed.)

$$
\begin{aligned}
\mathcal{E}^{ext} = \{\ & \texttt{var}\ x \mid x \in \mathcal{V}_e\} \\
\cup\ & \boxed{\mathcal{O}^{\texttt{def}}} \\
\cup\ & \{\texttt{basic}\ b \mid b \in \mathcal{B}_v\} \\
\cup\ & \{\texttt{symbol}\ s\ \sigma s\ es \mid s \in \mathcal{S}^e,\ \sigma s \in \langle \mathcal{T} \rangle,\ es \in \langle \mathcal{E}^{ext} \rangle \\
& \qquad\qquad \mid |\sigma s| = \mathit{Arity}^1(s) \wedge |es| \leq \mathit{Arity}^2(s)\} \\
\cup\ & \{\texttt{apply}\ e_1\ \texttt{to}\ e_2 \mid e_1 \in \mathcal{E}^{ext}, e_2 \in \mathcal{E}^{ext}\} \\
\cup\ & \{\texttt{case}\ e\ \texttt{of}\ \mathit{alts} \mid e \in \mathcal{E}^{ext}, \mathit{alts} \in \langle \mathit{Alt} \rangle\} \\
\cup\ & \{\texttt{let}\ \mathit{binds}\ \texttt{in}\ e \mid \mathit{binds} \in \langle \mathit{LetBind} \rangle,\ e \in \mathcal{E}^{ext}\} \\
\cup\ & \boxed{\{\texttt{step}\ \mathit{stepbinds}\ \texttt{in}\ e \mid \mathit{stepbinds} \in \langle \mathit{StepBind} \rangle,\ e \in \mathcal{E}^{ext}\}} \\
\cup\ & \{\texttt{let!}\ x_1 = e_1\ \texttt{in}\ e_2 \mid x \in \mathcal{V}_e^\bullet,\ e_1 \in \mathcal{E}^{ext},\ e_2 \in \mathcal{E}^{ext}\} \\
\cup\ & \{\bot_\sigma \mid \sigma \in \mathcal{T}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{StepBind} &= \{x\ \texttt{binds}\ e \mid x \in \mathcal{V}_e^\bullet \cup \mathcal{O}^{def},\ e \in \mathcal{E}^{ext}\} \\
\mathcal{O}^{\texttt{def}} &= \{\texttt{obj}\ o\ x \mid o \in \mathcal{O},\ x \in \mathcal{V}_e\} \\
\mathcal{O} &= \{\texttt{objid}\ z \mid z \in \mathbf{Z}\}
\end{aligned}
$$

Note that in *StepBind* not only object states can be bound, but (local) variables as well ($x \in \mathcal{V}_e^\bullet$). There is a technical reason for that: this is how the "current state" of an object can be retrieved and used in an expression.

The changes to $\mathcal{E}$ induce further modifications, e.g. in the definition of functions ($\mathcal{F}^{\texttt{def}}$) in Sparkle Core. The modifications make it possible to use objects both in the formal parameter list and in the body of functions, granted that the same object identifier does not occur more than once in the formal parameter list.

As an illustration, the two definitions of the function `sort3` (expressed in Clean, in Section 2 and in extended Clean, in Section 3) turned into Sparkle Core and extended Core are provided. It is instructive to see the differences between the two definitions, without trying to understand all their details. First, let us have a look at the one without objects. (Almost the same information is available in Sparkle – using the appropriate options offered by its user interface – as in the Sparkle Core code below.)

```
fundef sort3 αs  σs1 τ ⟨(var (exprvar 1)),
                        (var (exprvar 2)),
                            (var (exprvar 3))⟩
  let ⟨ (exprvar 4)
          binds (symbol sort2  σs2 ⟨(var (exprvar 1)) (var (exprvar 2))⟩ ),
     (exprvar 5) binds (symbol _tupleselect_2_1  σs3 ⟨ (var (exprvar 4)) ⟩ ),
     (exprvar 6) binds (symbol _tupleselect_2_2  σs3 ⟨ (var (exprvar 4)) ⟩ ),
     (exprvar 7)
          binds (symbol sort2  σs2 ⟨ (var (exprvar 6)), (var (exprvar 3)) ⟩ ),
     (exprvar 8)
          binds (symbol _tupleselect_2_1  σs3 ⟨ (var (exprvar 7)) ⟩ ),
     (exprvar 9)
          binds (symbol _tupleselect_2_2  σs3 ⟨ (var (exprvar 7)) ⟩ ),
     (exprvar 10)
          binds (symbol sort2  σs2 ⟨ (var (exprvar 5)), (var (exprvar 9)) ⟩ ),
     (exprvar 11)
          binds (symbol _tupleselect_2_1  σs3 ⟨ (var (exprvar 10)) ⟩ ),
     (exprvar 12)
          binds (symbol _tupleselect_2_2  σs3 ⟨ (var (exprvar 10)) ⟩ ) ⟩
  in symbol tuple3  σs4 ⟨(var(exprvar 11)),
                         (var(exprvar 8)),
                             (var (exprvar 12))⟩
```

Compare the above definition with the following *extended* Sparkle Core code:

```
fundef sort3 αs  σs1 τ ⟨(obj (objid 1) (exprvar 1)),
                        (obj (objid 2) (exprvar 2)),
                            (obj (objid 3) (exprvar 3))⟩
  step ⟨ (exprvar 4)
           binds (symbol sort2  σs2 ⟨ (obj (objid 1) (exprvar 1)),
                                      (obj (objid 2) (exprvar 2)) ⟩ ),
         ( obj (objid 1) (exprvar 5))
           binds (symbol _tupleselect_2_1  σs3 ⟨ (var (exprvar 4)) ⟩ ),
          ( obj (objid 2) (exprvar 6))
```

```
          binds (symbol _tupleselect_2_2  σs3 ⟨ (var (exprvar 4)) ⟩ ) ⟩
in step ⟨ (exprvar 7)
            binds (symbol sort2  σs2 ⟨ (obj (objid 2) (exprvar 6)),
                                       (obj (objid 3) (exprvar 3)) ⟩ ),
           ( obj (objid 2) (exprvar 8))
            binds (symbol _tupleselect_2_1  σs3 ⟨ (var (exprvar 7)) ⟩ ),
           ( obj (objid 3) (exprvar 9))
            binds (symbol _tupleselect_2_2  σs3 ⟨ (var (exprvar 7)) ⟩ ) ⟩
in step ⟨ (exprvar 10)
            binds (symbol sort2  σs2 ⟨ (obj (objid 1) (exprvar 5)),
                                       (obj (objid 3) (exprvar 9)) ⟩ ),
           ( obj (objid 1) (exprvar 11))
            binds (symbol _tupleselect_2_1  σs3 ⟨ (var (exprvar 10)) ⟩ ),
           ( obj (objid 3) (exprvar 12))
            binds (symbol _tupleselect_2_2  σs3 ⟨ (var (exprvar 10))⟩ ) ⟩
in symbol tuple3  σs3 ⟨(obj (objid 1) (exprvar 11))
                       (obj (objid 2) (exprvar 8))
                           (obj (objid 3) (exprvar 12))⟩
```

The semantics of Sparkle Core has been changed in such a way that object definitions and steps are only used during the formulation and proof of temporal properties. Otherwise objects can be reduced to variables and steps to let-expressions. The reduction rules expressing this are the following.

$$\frac{\texttt{obj}\ o\ x}{x} \qquad \frac{\texttt{step}\ letbinds\ \texttt{in}\ e}{\texttt{let}\ letbinds\ \texttt{in}\ e}$$

Note that the "stepbinds" part of a step have to be reduced to a "letbind" (by reducing all objects of a stepbind to variables) before reducing it to a let-definition.

5 Temporal propositions

For formulating temporal properties of abstract objects, the logical framework of Sparkle has to be made capable to manage temporal propositions. This paper shows how safety properties, namely *invariants* and *unless* properties should be handled in this extended framework. The definition $\mathcal{P}$ of propositions has been changed to include temporal propositions. To describe e.g. invariants, $\mathcal{P}^{inv}$ is introduced:

$$\begin{aligned}\mathcal{P}^{ext} &= \mathcal{P} \cup \mathcal{P}^{inv} \cup \ldots \\ \mathcal{P}^{inv} &= \{p\ \texttt{inv}\ (f\ cxs)\ q \mid q \in \mathcal{P},\ p \in \mathcal{OP},\ f \in \mathcal{F}, cxs \in \langle\mathcal{E}\rangle\}\end{aligned}$$

An invariant proposition "p `inv` $(f\ cxs)\ q$" means that proposition p holds invariantly during the evaluation of "$f\ cxs$" with respect to the precondition q. In the definition above, f is a function symbol, and cxs is an actual parameter list containing expressions of (the original) Sparkle Core. Furthermore, q is a proposition

of the basic logic of (the original) Sparkle, referring to the variables occurring in cxs. On the other hand, p is an "object proposition" ($\mathcal{OP}$), which can refer to object identifiers as well. $\mathcal{OP}$ differs from $\mathcal{P}$ in that the expressions occurring in it come from a modified set of expressions $\mathcal{E}^{temp}$ instead of $\mathcal{E}$. The definition of $\mathcal{E}^{temp}$ adds the alternative $\mathcal{O}$ to $\mathcal{E}$. (Notice the difference between $\mathcal{E}^{temp}$ and $\mathcal{E}^{ext}$. The latter introduces an alternative for $\mathcal{O}^{\texttt{def}}$, not $\mathcal{O}$, and a further alternative for steps.)

As an example, consider the following invariant property of the `sort3` function. It states that, given the precondition $x + y + z = 0$, the sum of the three objects is equal to 0 during the evaluation of "`sort3 x y z`".

$$\forall x \forall y \forall z \ (obj_1 + obj_2 + obj_3 = 0) \ \texttt{inv} \ (sort3 \ x \ y \ z) \ (x + y + z = 0)$$

The names identifying the objects (obj_1, obj_2 and obj_3) are declared in the `sort3` function: the object names appearing in an invariant proposition are resolved in the scope of the function the proposition is referring to.

In extended Sparkle Core the above invariant is formulated in the following way:

```
(forall exprs (exprvar 53) (forall exprs (exprvar 54) (forall exprs (exprvar 55)
  ((symbol (+) σs ⟨ (objid 1),
                    (symbol (+) σs ⟨ (objid 2), (objid 3) ⟩) ⟩)
                      equals (basic (int 0)) )

  inv (sort3 ⟨ (var (expvar 53)), (var (exprvar 54)), (var (exprvar 55))⟩)

  ((symbol (+) σs ⟨ (var (exprvar 53)),
          (symbol (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
                  equals (basic (int 0))) )))
```

Invariants should follow from the precondition, and must be preserved by the (atomic) state transitions [4, 13]. The preservation of a statement with respect to a state transition is expressed with the weakest precondition operator wp [7]. In this case there are three state transitions, corresponding to the three evaluations of `sort2`. Let us abbreviate "$obj_1 + obj_2 + obj_3 = 0$" with p and "$x + y + z = 0$" with q. After introducing (fixing) the universally quantified variables x, y and z, the invariant can be rewritten to the conjunction of the following four proposition.

1. $q \Rightarrow p$
2. $q \wedge p \ \Rightarrow \ wp(obj_1, \ obj_2 \ = \ sort2 \ obj_1 \ obj_2)(p)$
3. $q \wedge p \ \Rightarrow \ wp(obj_2, \ obj_3 \ = \ sort2 \ obj_2 \ obj_3)(p)$
4. $q \wedge p \ \Rightarrow \ wp(obj_1, \ obj_3 \ = \ sort2 \ obj_1 \ obj_3)(p)$

Since the precondition may only refer to variables, and not to objects ($q \in \mathcal{P}$), it can be used as a hypothesis in each generated propositions.

The meaning of the invariant proposition, of course, is given with respect to a program context, e.g. the definition of the function `sort3`. In the appendix the function computing the semantical value of invariant propositions is provided. Applying this semantical function, the semantics of the above invariant is obtained in the Sparkle system.

```
(forall exprs (exprvar 53) (forall exprs (exprvar 54) (forall exprs (exprvar 55)
      binary (binary (binary prop_init and prop_1) and prop_2) and prop_3
)))
```

with the following abbreviations:

```
prop_init =
 (forall exprs (exprvar 56) (forall exprs (exprvar 57) (forall exprs (exprvar 58)
  (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
             (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
            equals (basic (int 0)))
    implies
     (binary ( binary ( binary ( (var (exprvar 56) ) equals (var (exprvar 53) ) )
        and ( (var (exprvar 57) ) equals (var (exprvar 54) ) ) )
        and ( (var (exprvar 58) ) equals (var (exprvar 55) ) ) )
     implies
        ((symbol  (+) σs ⟨ (var (exprvar 56)),
            (symbol  (+) σs ⟨ (var (exprvar 57)), (var (exprvar 58)) ⟩) ⟩)
        equals (basic (int 0)) ) ) ) )))

prop_1 =
 (forall exprs (exprvar 59) (forall exprs (exprvar 60)
 (forall exprs (exprvar 61) (forall exprs (exprvar 62) (forall exprs (exprvar 63)
  (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
             (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
            equals (basic (int 0)))
    implies
  (binary ((symbol (+) σs ⟨ (var (exprvar 59)),
                  (symbol(+) σs ⟨ (var (exprvar 60)), (var (exprvar 61)) ⟩) ⟩)
                        equals (basic (int 0)))
  implies
     (binary (binary (binary ( (var (exprvar 4))
       equals ( symbol sort2  σs_2 ⟨ (var (exprvar 59)), (var (exprvar 60) )⟩ ) )
     and ((var (exprvar 62)) equals (symbol _tupleselect_2_1  σs_3
                                          ⟨ (var (exprvar 4))⟩ )) )
     and ((var (exprvar 63)) equals (symbol _tupleselect_2_2  σs_3
                                          ⟨ (var (exprvar 4))⟩ ) ) )
  implies
     ((symbol (+) σs ⟨ (var (exprvar 62)),
              (symbol (+) σs ⟨ (var (exprvar 63)), (var (exprvar 61)) ⟩) ⟩)
                    equals (basic (int 0))) )))
```

```
prop2 =
  (forall exprs (exprvar 64) (forall exprs (exprvar 65)
  (forall exprs (exprvar 66) (forall exprs (exprvar 67) (forall exprs (exprvar 68)
   (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
              (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
                equals (basic (int 0)))
     implies
   (binary ((symbol (+) σs ⟨ (var (exprvar 64)),
                    (symbol (+) σs ⟨ (var (exprvar 65)), (var (exprvar 66)) ⟩) ⟩)
                          equals (basic (int 0)))
   implies
      (binary (binary (binary ( (var (exprvar 7))
        equals ( symbol sort2  σs2 ⟨ (var (exprvar 65)), (var (exprvar 66) )⟩ ) )
      and ((var (exprvar 67)) equals (symbol _tupleselect_2_1  σs3
                                          ⟨ (var (exprvar 7))⟩ )) )
      and ((var (exprvar 68)) equals (symbol _tupleselect_2_2  σs3
                                          ⟨ (var (exprvar 7))⟩ ) ) )
   implies
      ((symbol (+) σs ⟨ (var (exprvar 64)),
               (symbol (+) σs ⟨ (var (exprvar 67)), (var (exprvar 68)) ⟩) ⟩)
                     equals (basic (int 0))) )))
```

```
prop3 =
  (forall exprs (exprvar 69) (forall exprs (exprvar 70)
  (forall exprs (exprvar 71) (forall exprs (exprvar 72) (forall exprs (exprvar 73)
   (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
              (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
                equals (basic (int 0)))
     implies
   (binary ((symbol (+) σs ⟨ (var (exprvar 69)),
                    (symbol (+) σs ⟨ (var (exprvar 70)), (var (exprvar 71)) ⟩) ⟩)
                          equals (basic (int 0)))
   implies
      (binary(binary (binary ( (var (exprvar 10))
        equals ( symbol sort2  σs2 ⟨ (var (exprvar 69)), (var (exprvar 71) )⟩ ) )
      and ((var (exprvar 72)) equals (symbol _tupleselect_2_1  σs3
                                          ⟨ (var (exprvar 10))⟩ )) )
      and ((var (exprvar 73)) equals (symbol _tupleselect_2_2  σs3
                                          ⟨ (var (exprvar 10))⟩ ) ) )
   implies
      ((symbol (+) σs ⟨ (var (exprvar 72)),
               (symbol (+) σs ⟨ (var (exprvar 70)), (var (exprvar 73)) ⟩) ⟩)
                     equals (basic (int 0))) )))
```

6 Some more complex examples

In this section two more complex examples are presented. The first example introduces a simple program modelling a database of financial transactions, and an invariant property of this program is provided. The second example is an implementation of the dining philosophers' problem, and it is used to illustrate unless properties.

6.1 Transactions with an invariant property

In this example a transaction is made up of a timestamp, describing when the transaction occurred, and an integer number describing the amount of money transferred in the transaction. The database contains a list of transactions and the overall sum of the amounts transferred in the transactions. The following definitions are written in Clean.

The representation of the type `Timestamp` is irrelevant in this example, hence this type is defined abstract. Two operations are needed on `Timestamp`, namely "`<`" and "`eval`". The type class "`<`" (from the standard library of Clean) denotes an ordering, while the type class "`eval`" (from the standard library of Sparkle) is used to rule out (partially) undefined expressions.

```
:: Timestamp
instance < Timestamp
instance eval Timestamp
```

The operations have to satisfy the following lemma. If the timestamps t_1 and t_2 are not partially undefined, then $t_1 < t_2$ is not partially undefined, either.

$$\forall t_1, t_2 \in \mathrm{T}imestamp: \ eval(t_1) \wedge eval(t_2) \Rightarrow eval(t_1 < t_2)$$

Type `Transaction` also has "`<`" and "`eval`" operations, furthermore, two getter operations have been provided as well. Transactions are ordered according to their timestamps, and a transaction is not partially undefined if (and only if) neither of its two components are.

```
:: Transaction = Tx Timestamp Int
timestamp (Tx timestamp amount) = timestamp
amount    (Tx timestamp amount) = amount
instance < Transaction where (<) (Tx t1 a1) (Tx t2 a2) = t1 < t2
instance eval Transaction
    where eval (Tx timestamp amount) = eval timestamp && eval amount
```

The database – given by the synonym type `DB` – is also an instance of the type class "`eval`". (In this simple example program it might be assumed, but it is not obligatory, that the timestamp of the transactions is a primary key.)

```
:: DB :== (Int,[Transaction])
instance eval DB where eval (sum,list) = eval sum && eval list
```

This example is based on the following database operations. The first one creates an empty database, the second one adds a transaction to the beginning of the transaction list, the third one removes the first transaction from the list and, finally, the fourth one sorts the transactions according to their timestamps. These operations describe some basic state transitions of the databases.

```
newDB :: DB
newDB = (0,[])

insertFirst :: Transaction DB -> DB
insertFirst tx=:(Tx tstamp amount) (sum,txs) = (sum+amount,[tx:txs])

removeFirst :: DB -> DB
removeFirst db=:(_,[]) = db
removeFirst (sum,[(Tx tstamp amount):txs]) = (sum-amount,txs)

sortDB (sum,txs) = (sum, isort txs)
```

For sorting we have applied a simple insertion sort function, also used by [20].

```
ins :: a [a] -> [a] | < a
ins e [] = [e]
ins e [x:xs] = if (x<e) [x:ins e xs] [e:x:xs]

isort :: [a] -> [a]  | < a
isort [] = []
isort [x:xs] = ins x (isort xs)
```

Now a simple "scenario" application can be developed, built upon the basic operations, which 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 the transaction is inserted into the database, then the resulting database is sorted, finally the first transaction stored in the (sorted) database is removed.

```
scenario :: DB Transaction -> DB
scenario db tx
   # db = insertFirst tx db
   # db = sortDB db
   # db = removeFirst db
   = db
```

Now this scenario can be rewritten in extended Clean, making use of the object abstraction technique.

```
scenario (.|. obj db) tx
   .#. (.|. obj db) = insertFirst tx (.|. obj db)
   .#. (.|. obj db) = sortDB         (.|. obj db)
   .#. (.|. obj db) = removeFirst    (.|. obj db)
    =  (.|. obj db)
```

As mentioned earlier, the invariant property of databases is that the first component equals to the total sum of the money transferred by the transactions stored in the second component. In Sparkle, the definitions (functions and predicates) required to formulate a theorem should be given in Clean. A function that sums up the amounts of money appearing in a list of transactions will be used here.

```
sumUp :: [Transaction] -> Int
sumUp [] = 0
sumUp [tx:txs] = amount tx + sumUp txs
```

The invariant property can be formalised now:

$$\forall db\ \forall tx :$$
$$\big(\text{eval obj} \ \wedge\ \text{fst obj} = \text{sumUp (snd obj)}\big) \quad \mathbf{i}\text{nv} \quad \big(\text{scenario db tx}\big)$$
$$\big(\text{eval tx} \ \wedge\ \text{eval db} \ \wedge\ \text{fst db} = \text{sumUp (snd db)}\big)$$

Let us introduce the following abbreviations:

$$\begin{aligned} \mathrm{I}(x) &= \big(\text{eval x} \ \wedge\ \text{fst x} = \text{sumUp (snd x)}\big) \\ \mathrm{P}RE &= \big(\text{eval tx} \ \wedge\ \mathrm{I}(\text{db})\big) \end{aligned}$$

The "Invariant" tactic of extended Sparkle will introduce the universally quantified variables *db* and *tx* among the declared variables, and then rewrite the above invariant into the following subgoals.

1. $\forall \mathrm{db}_1$: PRE $\wedge$ db $=$ db_1 $\Rightarrow$ $\mathrm{I}(\mathrm{db}_1)$
2. $\forall \mathrm{db}_1\ \forall \mathrm{db}_2$: PRE $\wedge$ $\mathrm{I}(\mathrm{db}_1)$ $\wedge$ db_2 = insertFirst db_1 tx $\Rightarrow$ $\mathrm{I}(\mathrm{db}_2)$
3. $\forall \mathrm{db}_1\ \forall \mathrm{db}_2$: PRE $\wedge$ $\mathrm{I}(\mathrm{db}_1)$ $\wedge$ db_2 = sortDB db_1 $\Rightarrow$ $\mathrm{I}(\mathrm{db}_2)$
4. $\forall \mathrm{db}_1\ \forall \mathrm{db}_2$: PRE $\wedge$ $\mathrm{I}(\mathrm{db}_1)$ $\wedge$ db_2 = removeFirst db_1 $\Rightarrow$ $\mathrm{I}(\mathrm{db}_2)$

These subgoals can then be proved with Sparkle – the proofs require about 300 steps.

6.2 Dining philosophers with an unless property

This example uses an implementation of Dijkstra's famous "dining philosophers' problem" [6]. In the middle of a dining room there is a table with a big plate of spaghetti. Around the table there are five philosophers spending their lives thinking and eating spaghetti. A philosopher needs two forks for eating spaghetti. However, there are only five forks available, one between each pair of philosophers. Hence two neighbouring philosophers can never eat simultaneously. At the beginning of the program each philosopher is thinking. When a philosopher becomes hungry, he tries to pick up the two forks that he is sharing with his two neighbours. If he

manages to do so, he eats for a while, and then he releases the forks and starts thinking. A hungry philosopher has to wait, if one of the neighbouring philosophers is using the fork shared between them.

The example program uses concurrent ObjectIO processes. It has a graphical user interface for the manipulation of the philosophers. Each philosopher is implemented as a process with its own local state. Moreover, the application also contains a server process that provides the necessary synchronisation. An important part of the server process is the `next_event` function, which controls the critical state transitions of the philosophers.

The local state of the philosopher processes is a value of type `State`.

```
:: State = Thinking | Hungry | Eating
```

The local state of the server process is a list of `State`s. The i^{th} element of this list is invariantly equal to the local state of the i^{th} philosopher process.

If a philosopher would like to change its local state from Thinking to Eating, or from Eating to Thinking, the server computes the new states for all of the philosophers. This computation is implemented in the `next_event` function.

```
next_event :: [State] Int -> [State]
```

The function has two arguments. The first one is the local state of the server and the second one is the ordinal number of the philosopher requesting the state transition. The result of the function is the new local state of the server.

In order to present an *unless* property in a very simple context, this concurrent program will be simulated with the following function.

```
process_events:: [State] [Int] -> [State]
process_events states [] = states
process_events states [index:indices]
  | index < 0 || index >= length states        // illegal index
       = process_events states indices         // discarding...
  | otherwise
       # states = next_event states index      // process index
       = process_events states indices         // process rest
```

Since the current implementation of the extended Sparkle system cannot handle rule alternatives, the guards have to be eliminated and the second function alternative has to be reformulated with an `if`-construct. Furthermore, the state transitions have to be made explicit with let-before constructs: this justifies the presence of the second, at first glance unnecessary, such construct in the following function alternative.

```
process_events states [index:indices]
  # states = if (index < 0 || index >= length states)
               states                          // skip illegal index
               (next_event states index)       // process index
  # states = process_events states indices     // process rest
  = states
```

Let us introduce an object that corresponds to the local state of the server process. Then the `process_event` function can be rewritten to extended Clean.

```
process_events :: [State] [Int] -> [State]
process_events (.|. obj1 states1) [] = (.|. obj1 states1)
process_events(.|. obj1 states1) [index:indices]
  .#. (.|. obj1 states2) =
        if ((index < 0) || (index >= length (.|. obj1 states1)))
           (.|. obj1 states1)
           ( next_event (.|. obj1 states1) index )
  .#. (.|. obj1 states3) process_events (.|. obj1 states2) indices
   =  (.|. obj1 states3)
```

The *unless* property expresses the following. Given a Hungry philosopher and his Eating right neighbour, they will not change their states unless the Eating right neighbour starts Thinking. To be less informal, this can be written in the following way.

$\forall i \in \mathrm{d}omain(states)$:

$$\begin{aligned} &\mathrm{states}_i = \mathrm{Hungry} \quad \wedge \quad \mathrm{states}_{rightneighbour(i)} = \mathrm{Eating} \\ &\mathbf{u}nless(\mathrm{p}rocess\_events) \\ &\mathrm{states}_{rightneighbour(i)} = \mathrm{Thinking} \end{aligned}$$

The property "P $\mathbf{u}nless_{\mathrm{p}rog}$ Q" means that during the execution of the program "prog", the statement P remains to hold until the statement Q becomes true. This kind of safety properties can be expressed with the *weakest precondition* operator: for every atomic state transition of the program "prog", the weakest precondition of $P \vee Q$ with respect to the state transition follows from $P \ \wedge \neg Q$.

$$\forall st \in \mathrm{p}rog : \quad P \ \wedge \neg Q \ \Rightarrow \ wp(st, P \vee Q)$$

Now let us formulate the *unless* property in a more precise way. To increase readability, the syntax of the extended Sparkle Core is not followed rigorously.

$\forall$`states` $\forall$`indices` $\forall$`i`:

```
(eval states) ∧ (eval indices) ∧ (i >= 0) ∧ (i < length states) ∧
(obj1!!i == Hungry) ∧ (obj1!!(rightneighbour obj1 i) == Eating)

UNLESS (process_events states indices)

(obj1!!(rightneighbour obj1 i) == Thinking)
```

The appendix provides the function computing the semantical value of *unless* propositions. Let us investigate what proposition results from applying this semantical function. In the extended Clean version of the function `process_events` there

are two steps, two atomic state transitions affecting the object $\texttt{obj}_1$. The first step is the `if`-construct, and the second step is the recursive call to `process_events`. When reasoning about safety properties, proofs about recursive calls of functions with the same object argument(s) can be omitted. Hence there is only one proposition to prove for the afore-mentioned *unless* property:

```
∀states ∀indices' ∀i ∀states' ∀index ∀indices

(eval states) ∧ (eval indices') ∧ (i >= 0) ∧ (i < length states) ∧
(states!!i == Hungry) ∧
(states!!(rightneighbour states i) == Eating) ∧
¬ (states!!(rightneighbour states i) == Thinking)  ∧
(indices' = [index:indices]) ∧
(states' = if (index < 0 || index >= length states)
              states
              (next_event states index))

⇒

((eval states') ∧ (eval indices') ∧ (i >= 0) ∧ (i < length states') ∧
(states'!!i == Hungry) ∧
(states'!!(rightneighbour states' i) == Eating))
∨ (states'!!(rightneighbour states' i) == Thinking)
```

7 Conclusions and future work

The authors have studied a method that allows the formulation and proof of safety properties (namely invariants and unless) in pure functional languages. The concept of object abstraction has been presented, which is based on contracting functional variables into objects with dynamic (temporal) behaviour. Language constructs describing object abstraction have been introduced into the purely functional programming language Clean. This extended Clean language is considered as an intermediate language: programs written in this language are intended to be generated by an appropriate integrated development environment containing a refactoring tool. Support for the new language constructs will thus be provided by an interactive environment.

Programs written in the extended Clean language are processed by a theorem prover framework. This framework was obtained by enabling Sparkle, the theorem prover designed for Clean, to manage object abstraction and temporal propositions. This paper describes invariant and unless propositions, with a focus on the semantic function computing the meaning of invariant propositions in the logic framework of the Sparkle system.

In the future the object abstraction technique will be generalised to enable the definition of the states of the same object within more than one function.

This will make it possible to express that the atomic state transitions of an object can be spread among many function bodies. This generalisation requires a more sophisticated view of "time" (with respect to temporal operators) and a hierarchical system of state transitions.

The authors also plan to implement an integrated software development environment, which supports object abstraction in the user interface and related refactoring possibilities. This IDE will eliminate the need for programming in extended Clean. Furthermore, the Sparkle framework will be made capable of handling additional temporal propositions, so that it will be possible to express progress propositions (such as "ensures" and "leads-to" [4]) as well. Finally, in order to make temporal reasoning less cumbersome in practice, the authors will provide useful theorems about temporal operators (such as the "weakening" rule or the "conjunction with invariants" rule [13]) as axioms.

The main advantage of the method described in this paper is that in this extended logical framework certain important properties of programs can be expressed conveniently and briefly, at a high level of abstraction. The sophisticated logical operators of temporal logics can neatly express safety and progress properties of programs, and these properties, as the examples of this paper have illustrated, are sensible and useful also in the world of functional programming. Moreover, the addition of theorems about temporal logical operators can make reasoning about programs even less tiring and less complicated.

Another important issue of this approach is that the proofs constructed in the extended Sparkle system are represented in a completely machine processable form. As a consequence, not only the program, but also its proved temporal properties and the proofs themselves can be stored, transmitted and checked by a computer. This allows the transmission of safe code among (components of) applications. A detailed presentation of this proof-carrying code technique can be found in [8].

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] 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.

[3] Butterfield, Andrew: Reasoning about I/O and Exceptions. *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 33-48.

[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. *Hierarchical ordering of sequential processes.* Acta Informatica, 1, 115–138, 1971.

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

[8] Daxkobler K., Horváth Z., Kozsik T.: A Prototype of CPPCC - Safe Functional Mobile Code in Clean. *Proceedings of Implementation of Functional Languages'02*, Madrid, Spain, Sept. 15-19, 2002. pp. 301-310.

[9] Diviánszky P. - Szabó-Nacsa R. - Horváth Z.: A Framework for Refactoring Clean Programs. *6th International Conference on Applied Informatics*, Eger, Hungary January 27-31 2004.

[10] Dowse, M., Butterfield, A., van Eekelen, M., de Mol, M., Plasmeijer, R.: Towards Machine-Verified Proofs for I/O *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 469-480.

[11] Dowse, M., Butterfield, A.: A Language for Reasoning about Concurrent Functional I/O (Draft) *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 129-141.

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

[13] Horváth Z.: The Formal Specification of a Problem Solved by a Parallel Program—a Relational Model. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, Tomus XVII. (1998) pp. 173–191.

[14] 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.

[15] 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.

[16] Horváth Z., Hernyák Z., Zsók V.: Coordination Language for Distributed Clean. To appear in *Acta Cybernetica*, Szeged, Hungary, 2005.

[17] 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

[18] Horváth Z. - Kozsik T. - Tejfel M.: Verifying invariants of abstract functional objects - a case study. *6th International Conference on Applied Informatics*, Eger, Hungary January 27-31 2004.

[19] 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.

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

[21] de Mol, Maarten. PhD thesis (in preparation), Radboud University Nijmegen.

[22] 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.

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

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

[25] Szabó-Nacsa R., Diviánszky P., Horváth Z.: An Environment for Safe Refactoring Clean Programs. *CSCS 2004, The Fourth Conference of PhD Students in Computer Science*, Szeged, Hungary, July 1-4, 2004.

A The semantics of invariant and unless propositions

The semantics of an invariant and an unless proposition in a program context ψ can be computed according to the following definition:

$$Sem\ ((p\ \texttt{unless}\ f\ cxs\ q), \psi) = \begin{cases} SemFunUnless(p, Def(f, \psi),\ cxs,\ q) & \text{if } cxs = Arity^2(f) \\ undef & \text{otherwise} \end{cases}$$

$$SemFunUnless: \mathcal{OP} \times (\langle \mathcal{V}_e^\bullet \cup \mathcal{O}^{\texttt{def}} \rangle \times \langle \mathcal{V}_t^\bullet \rangle \times \mathcal{E}^{ext}) \times \langle \mathcal{E}^{ext} \rangle \times \mathcal{P} \to \mathcal{P}$$

$$SemFunUnless\ (p,\ (xs,\ \alpha s,\ e),\ cxs,\ q) = SemExprUnless(p,\ q,\ e,\ True)$$

$$SemExprUnless: \mathcal{OP} \times \mathcal{OP} \times \mathcal{E}^{ext} \times \mathcal{P} \to \mathcal{P}$$
$$SemExprUnless(p,\ q,\ (\texttt{let}\ binds\ \texttt{in}\ e),\ pred) = SemExprUnless(p,\ q,\ e,\ pred)$$

$$SemExprUnless(p,\ q,\ (\texttt{step}\ stepbinds\ \texttt{in}\ e),\ pred) = SemExprUnless(p,\ q,\ e,\ (\texttt{binary}\ pred\ \texttt{and}\ (WpImpCalcUn(p,\ q,\ stepbinds))))$$

$$SemExprUnless(p,\ q,\ \_,\ pred) = pred$$

$$WpImpCalcUn: \mathcal{OP} \times \mathcal{OP} \times \langle StepBind \rangle \to \mathcal{P}$$

$$WpImpCalcUn(p,\ q,\ stepbinds) = ObjToVarLeft((\texttt{binary}\ p\ \texttt{or}\ q), ObjToVarRight((\texttt{binary}\ p\ \texttt{and}\ (\texttt{unary not}\ q)), \texttt{nil}, stepbinds, \texttt{nil}),\ \texttt{nil}, \texttt{nil})$$

$$Sem\ ((p\ \texttt{inv}\ f\ cxs\ q), \psi) = \begin{cases} SemFunInv(p, Def(f, \psi),\ cxs,\ q) & \text{if } cxs = Arity^2(f) \\ undef & \text{otherwise} \end{cases}$$

$$SemFunInv: \mathcal{OP} \times (\langle \mathcal{V}_e^\bullet \cup \mathcal{O}^{\texttt{def}} \rangle \times \langle \mathcal{V}_t^\bullet \rangle \times \mathcal{E}^{ext}) \times \langle \mathcal{E}^{ext} \rangle \times \mathcal{P} \to \mathcal{P}$$

$$SemFunInv\ (p,\ (xs,\ \alpha s,\ e),\ cxs,\ q) = SemExprInv(p,\ e,\ (ForallPred(q,\ (\texttt{binary}\ q\ \texttt{implies}\ ObjSubst(Parameter(xs, cxs),\ p)))),\ q)$$

$$SemExprInv: \mathcal{OP} \times \mathcal{E}^{ext} \times \mathcal{P} \times \mathcal{P} \to \mathcal{P}$$

$$SemExprInv(p,\ (\texttt{let}\ binds\ \texttt{in}\ e),\ pred,\ q) = SemExprInv(p,\ e,\ pred,\ \texttt{binary}\ q\ \texttt{and}\ BindsToEqs(binds))$$

$$SemExprInv(p,\ (\texttt{step}\ stepbinds\ \texttt{in}\ e),\ pred,\ q)$$
$$= SemExprInv(p,\ e,\ (\texttt{binary}\ pred\ \texttt{and}$$
$$(ForallPred(q,\ (\texttt{binary}\ q\ \texttt{implies}\ WpImpCalc(p,\ stepbinds))))),\ q)$$

$$SemExprInv(p,\ \_,\ pred,\ q)\ =\ pred$$

$$WpImpCalc:\ \mathcal{OP}\ \times\ \langle StepBind\rangle\ \rightarrow\ \mathcal{P}$$

$$WpImpCalc(p,\ stepbinds)$$
$$=\ ObjToVarLeft(p,\ ObjToVarRight(p, \texttt{nil}, stepbinds, \texttt{nil}),\ \texttt{nil}, \texttt{nil})$$

$$ObjToVarLeft:\ \mathcal{OP}\ \times\ (\mathcal{P}, \langle StepBind\rangle, \langle(\mathcal{O}, \mathcal{V}_e^\bullet)\rangle)\ \times\ \langle LetBind\rangle\ \times\ \langle(\mathcal{O}, \mathcal{V}_e^\bullet)\rangle\ \rightarrow\ \mathcal{P}$$

$$ObjToVarLeft(p,\ (oldp, \texttt{nil}, changedlist),\ letbinds, leftchangedlist)$$
$$= CreateForalls(changedlist\ *\ leftchangedlist,$$
$$\texttt{binary}\ oldp\ \texttt{implies}$$
$$(\ \texttt{binary}\ BindsToEqs(letbinds)$$
$$\texttt{implies}\ ObjListChangeInPred(changedlist,\ p)\ )\ )$$

$$ObjToVarLeft(p,\ (oldp,\ \texttt{cons}\ (\ (\texttt{obj}\ o\ x)\ \texttt{binds}\ e)\ sbs,\ changedlist),$$
$$letbinds,\ leftchangedlist)$$
$$= ObjToVarLeft(ObjChange(o,\ ox,\ p),\ (oldp,\ sbs,\ changedlist),$$
$$\texttt{cons}\ (ox\ \texttt{binds}\ e)\ letbinds,\ \texttt{cons}\ (o, ox)\ leftchangedlist),$$
$$\texttt{where}\ ox\ =\ NewVar()$$

$$ObjToVarLeft(p,\ (oldp,\ \texttt{cons}\ (\ x\ \texttt{binds}\ e)\ sbs,\ changedlist),$$
$$letbinds,\ leftchangedlist)$$
$$= ObjToVarLeft(p, (oldp,\ sbs,\ changedlist),$$
$$\texttt{cons}\ (x\ \texttt{binds}\ e)\ letbinds, leftchangedlist),\quad \texttt{if}\ x \in \mathcal{V}_e^\bullet$$

$$CreateForalls:\ \langle(\mathcal{O}, \mathcal{V}_e^\bullet)\rangle\ \times\ \mathcal{P}\ \rightarrow\ \mathcal{P}$$

$$CreateForalls(nil,\ pred)\ =\ pred$$

$$CreateForalls(cons\ (o, x)\ changedlist,\ pred)$$
$$=\ CreateForalls(changedlist,\ \texttt{forall exprs}\ x\ (pred))$$

$$BindsToEqs:\ \langle LetBind\rangle\ \rightarrow\ \mathcal{P}$$
$$BindsToEqs(\texttt{nil})\ =\ \texttt{constant}\ true$$

$$BindsToEqs(\texttt{cons}\ (x\ \texttt{binds}\ e)\ bs)$$
$$=\ \texttt{binary}\ (\ (\texttt{var}\ x)\ \texttt{equals}\ e)\ \texttt{and}\ BindsToEqs(bs)$$

$$ObjToVarRight:\ \mathcal{OP}\ \times\ \langle StepBind\rangle\ \times\ \langle StepBind\rangle\ \times\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle$$
$$\rightarrow\ (\mathcal{P},\ \langle StepBind\rangle,\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle)$$

$$ObjToVarRight(p,\ changedbinds,\ \texttt{nil},\ changedlist)$$
$$= (newp,\ changedbinds,\ (changedlist\ *\ newchangedlist)),$$
$$\texttt{where}\ (newp,\ newchangedlist)\ =\ ObjToNewVars_2(p)$$

$$ObjToVarRight(p,\ changedbinds,\ cons\ (x\ \texttt{binds}\ e)\ bs,\ oldchangedlist)$$
$$= ObjToVarRight(ObjListChangeInPred(changedlist,\ p),$$
$$changedbinds * (\texttt{cons}\ (x\ \texttt{binds}\ changedexpr)\ \texttt{nil}),$$
$$ObjListChangeInBinds(changedlist,\ bs),$$
$$(oldchangedlist\ *\ changedlist)),$$
$$\texttt{where}\ (changedlist,\ changedexpr)\ =\ ObjToVarInExpr(e)$$

$$ObjListChangeInPred:\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle\ \times\ \mathcal{OP}\ \rightarrow\ \mathcal{OP}$$
$$ObjListChangeInPred(\texttt{nil},\ p)\ =\ p$$

$$ObjListChangeInPred(\texttt{cons}\ (o,\ x)\ oxs,\ p)$$
$$= ObjListChangeInPred(oxs,\ ObjChange(o,\ x,\ p))$$

$$ObjListChangeInBinds:\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle\ \times\ \langle StepBind\rangle\ \rightarrow\ \langle StepBind\rangle$$
$$ObjListChangeInBinds(oxs,\ \texttt{nil})\ =\ \texttt{nil}$$

$$ObjListChangeInBinds(oxs,\ \texttt{cons}\ (x\ \texttt{binds}\ e)\ es)$$
$$= \texttt{cons}\ (\ x\ \texttt{binds}\ (ObjListChangeInExpr(oxs,\ e))\ )$$
$$ObjListChangeInBinds(oxs,\ es)$$

$$ObjToVarInExpr:\ \mathcal{E}^{ext}\ \rightarrow\ (\langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle,\ \mathcal{E}$$
$$ObjToVarInExpr(\texttt{var}\ x)\ =\ (\texttt{nil},\ \texttt{var}\ x)$$

$$ObjToVarInExpr(\texttt{obj}\ o\ x)\ =\ (cons\ (o,\ ox)\ \texttt{nil},\ \texttt{var}\ ox),$$
$$\texttt{where}\ ox\ =\ NewVar()$$

$$ObjToVarInExpr(\texttt{basic}\ b)\ =\ (\texttt{nil},\ \texttt{basic}\ b)$$

$$ObjToVarInExpr(\texttt{symbol}\ s\ \sigma s\ es)$$
$$= (changedlist,\ \texttt{symbol}\ s\ \sigma s\ changedexprlist),$$
$$\texttt{where}\ (changedlist, changedexprlist)\ =\ ObjToVarInExprList(es)$$

$$ObjToVarInExpr(\texttt{apply}\ e_1\ \texttt{to}\ e_2)$$
$$= (changedlist_1\ *\ changedlist_2,\ \texttt{apply}\ changedexpr_1\ \texttt{to}\ changedexpr_2),$$
$$\texttt{where}\ (changedlist_1, changedexpr_1)\ =\ ObjToVarInExpr(e_1),$$
$$(changedlist_2, changedexpr_2)$$
$$=\ ObjToVarInExpr(ObjListChangeInExpr(changedlist_1,\ e_2))$$

$$ObjToVarInExpr(\texttt{case}\ e\ \texttt{of}\ alts)$$
$$= (changedlist,\ \texttt{case}\ changedexpr\ \texttt{of}\ alts),$$
$$\texttt{where}\ (changedlist, changedexpr)\ =\ ObjToVarInExpr(e)$$

$$ObjToVarInExpr(\texttt{let}\ binds\ \texttt{in}\ e)$$
$$= (changedlist, \texttt{let}\ binds\ \texttt{in}\ changedexpr),$$
$$\texttt{where}\ (changedlist, changedexpr)\ =\ ObjToVarInExpr(e)$$

$ObjToVarInExpr(\texttt{step}\ stepbinds\ \texttt{in}\ e)$
$= (changedlist_1\ *\ changedlist_2,\ \texttt{step}\ changedstepbinds\ \texttt{in}\ changedexpr),$
$\quad \texttt{where}\ (changedlist_1,\ changedstepbinds) = ObjToVarInBinds(stepbinds),$
$\qquad (changedlist_2,\ changedexpr)$
$\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(changedlist_1,\ e))$

$ObjToVarInExpr(\texttt{let!}\ x_1 = e_1\ \texttt{in}\ e_2)$
$= (changedlist_1\ *\ changedlist_2, \texttt{let!}\ x_1 = changedexpr_1\ \texttt{in}\ changedexpr_2),$
$\quad \texttt{where}\ (changedlist_1, changedexpr_1) = ObjToVarInExpr(e_1),$
$\qquad (changedlist_2, changedexpr_2)$
$\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(changedlist_1,\ e_2))$

$ObjToVarInExpr(\bot_\sigma) = (\texttt{nil},\ \bot_\sigma)$
$ObjToVarInExprList : \langle \mathcal{E}^{ext} \rangle \rightarrow (\langle (\mathcal{O},\ \mathcal{V}_e^\bullet) \rangle,\ \langle \mathcal{E} \rangle$
$ObjToVarInExprList(\texttt{nil}) = (\texttt{nil},\ \texttt{nil})$

$ObjToVarInExprList(\texttt{cons}\ e\ es)$
$= (changedlist_1\ *\ changedlist_2,\ \texttt{cons}\ changedexpr\ changedexprlist),$
$\quad \texttt{where}(changedlist_1,\ changedexpr) = ObjToVarInExpr(e),$
$\qquad (changedlist_2,\ changedexprlist)$
$\qquad\quad = ObjToVarInExprList(ObjListChangeInExprList(changedlist_1,\ es))$

$ObjToVarInBinds : \langle StepBind \rangle \rightarrow (\langle (\mathcal{O},\ \mathcal{V}_e^\bullet) \rangle,\ \langle LetBind \rangle)$
$ObjToVarInBinds(\texttt{nil}) = (\texttt{nil},\ \texttt{nil})$

$ObjToVarInBinds(\texttt{cons}\ sb\ sbs)$
$= (changedlist_1\ *\ changedlist_2,\ \texttt{cons}\ changedbind\ changedbinds)$
$\quad \texttt{where}\ (changedlist_1,\ changedbind) = ObjToVarInBind(sb),$
$\qquad (changedlist_2,\ changedbinds)$
$\qquad\quad = ObjToVarInBinds(ObjListBindsChange(changedlist_1,\ sbs))$

$ObjToVarInBind : StepBind \rightarrow (\langle (\mathcal{O},\ \mathcal{V}_e^\bullet) \rangle,\ LetBind)$

$ObjToVarInBind((\texttt{obj}\ o\ x)\ \texttt{binds}\ e)$
$= (\texttt{cons}\ (o,\ ox)\ changedlist,\ ox\ \texttt{binds}\ changedexpr),$
$\quad \texttt{where}\ ox = NewVar(),$
$\qquad (changedlist,\ changedexpr)$
$\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(\texttt{cons}\ (o,\ ox)\ \texttt{nil},\ e))$

$ObjToVarInBind(x\ \texttt{binds}\ e) = ObjToVarInExpr(e),\ \texttt{if}\ x \in \mathcal{V}_e^\bullet$
$ObjListBindsChange : \langle (\mathcal{O},\ \mathcal{V}_e^\bullet) \rangle \times \langle StepBind \rangle \rightarrow \langle StepBind \rangle$
$ObjListBindsChange(oxs,\ \texttt{nil}) = \texttt{nil}$

$ObjListBindsChange(oxs,\ \texttt{cons}\ sb\ sbs)$
$\quad = \texttt{cons}\ ObjListBindChange(oxs,\ sb)\ ObjListBindsChange(oxs,\ sbs)$

$ObjListBindChange : \langle (\mathcal{O},\ \mathcal{V}_e^\bullet) \rangle \times StepBind \rightarrow StepBind$
$ObjListBindChange(\texttt{nil}, bs) = bs$

$ObjListBindChange(\texttt{cons}\ ox\ oxs, bs)$
$\quad = ObjListBindChange(oxs, ObjBindChange(ox,\ bs))$

$ObjBindChange : (\mathcal{O},\ \mathcal{V}_e^\bullet) \times StepBind \rightarrow StepBind$

$$ObjBindChange((\texttt{objid}\ z),\ x),\ (\texttt{objid}\ z)\ \texttt{binds}\ e)$$
$$=\ x\ \texttt{binds}\ ObjExprChange(e)$$

$$ObjBindChange((\texttt{objid}\ z),\ x),\ (\texttt{objid}\ y)\ \texttt{binds}\ e)$$
$$=\ (\texttt{objid}\ y)\ \texttt{binds}\ ObjExprChange(e),\ \texttt{if}\ \ z \neq\ y$$

$$ObjBindChange((\texttt{objid}\ z),\ x),\ w\ \texttt{binds}\ e)$$
$$=\ w\ \texttt{binds}\ ObjExprChange(e),\ \texttt{if}\ \ w \in \mathcal{V}_e^\bullet$$

$$ObjListChangeInExprList:\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle\ \times\ \langle\mathcal{E}^{ext}\rangle\ \times\langle\mathcal{E}^{ext}\rangle$$
$$ObjListChangeInExprList(oxs,\ \texttt{nil})\ =\ \texttt{nil}$$

$$ObjListChangeInExprList(oxs,\ \texttt{cons}\ e\ es)$$
$$=\ \texttt{cons}\ ObjListChangeInExpr(oxs,\ e)\ ObjListChangeInExprList(oxs,\ es)$$

$$ObjListChangeInExpr:\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle\ \times\ \mathcal{E}^{ext}\ \rightarrow\ \mathcal{E}^{ext}$$
$$ObjListChangeInExpr(\texttt{nil},\ e)\ =\ e$$

$$ObjListChangeInExpr(\texttt{cons}(o,\ x)\ oxs,\ e)$$
$$=\ ObjListChangeInExpr(oxs,\ ObjExprChange^{ext}(o,\ x,\ e))$$

$$ObjExprChange^{ext}:\mathcal{O}\ \times\ \mathcal{V}_e^\bullet\ \times\ \mathcal{E}^{ext}\ \rightarrow\ \mathcal{E}^{ext}$$
$$ObjExprChange^{ext}(o,\ x,\ (\texttt{var}\ y))\ =\ \texttt{var}\ y$$
$$ObjExprChange^{ext}((\texttt{objid}\ z),\ x,\ (\texttt{obj}\ (\texttt{objid}\ z))\ w)\ =\ \texttt{var}\ x$$

$$ObjExprChange^{ext}((\texttt{objid}\ z),\ x,\ (\texttt{obj}\ (\texttt{objid}\ y))\ w))$$
$$=\ \texttt{obj}\ (\texttt{objid}\ z))\ w\ \ \texttt{if}\ y\ \neq\ z$$

$$ObjExprChange^{ext}(o,\ x,\ (basic\ b))\ =\ basic\ b$$

$$ObjExprChange^{ext}(o,\ x,\ (\texttt{symbol}\ s\ \sigma s\ es))$$
$$=\ \texttt{symbol}\ s\ \sigma s\ ObjExprChangeList^{ext}(o,\ x,\ es)$$

$$ObjExprChange^{ext}(o,\ x,\ (\texttt{apply}\ e_1\ \texttt{to}\ e_2))$$
$$=\ \texttt{apply}\ ObjExprChange^{ext}(o,\ x,\ e_1)\ \texttt{to}\ ObjExprChange^{ext}(o,\ x,\ e_2)$$

$$ObjExprChange^{ext}(o,\ x,\ (\texttt{case}\ e\ \texttt{of}\ alts))$$
$$=\ \texttt{case}\ ObjExprChange^{ext}(o,\ x,\ e)\ \texttt{of}\ alts$$

$$ObjExprChange^{ext}(o,\ x,\ (\texttt{let}\ binds\ \texttt{in}\ e))$$
$$=\ \texttt{let}\ binds\ \texttt{in}\ ObjExprChange^{ext}(o,\ x,\ e)$$

$$ObjExprChange^{ext}(o,\ x,\ (\texttt{step}\ stepbinds\ \texttt{in}\ e))$$
$$=\ \texttt{let}\ ObjListBindChange((\texttt{cons}\ (o,\ x)\ \texttt{nil}), stepbinds)\ binds$$
$$\texttt{in}\ ObjExprChange^{ext}(o,\ x,\ e)$$

$$ObjExprChange^{ext}(o,\ x,\ (\texttt{let!}\ x_1\ =\ e_1\ \texttt{in}\ e_2))$$
$$=\ \texttt{let!}\ x_1\ =\ ObjExprChange^{ext}(o,\ x,\ e_1)\ \texttt{in}\ ObjExprChange^{ext}(o,\ x,\ e_2)$$

$$ObjExprChange^{ext}(o,\ x,\ (\bot_\sigma))\ =\bot_\sigma$$
$$ObjExprChangeList^{ext}:\ \mathcal{O}\ \times\ \mathcal{V}_e^\bullet\ \times\ \langle\mathcal{E}^{ext}\rangle\ \rightarrow\ \langle\mathcal{E}^{ext}\rangle$$
$$ObjExprChangeList^{ext}(o,\ x,\ \texttt{nil})\ =\ \texttt{nil}$$

$$ObjExprChangeList^{ext}(o,\ x,\ \texttt{cons}\ e\ es)$$
$$= \texttt{cons}\ ObjExprChange(o,\ x,\ e)\ ObjExprChangeList^{ext}(o,\ x,\ es)$$

$$ObjSubst:\ \langle EQVars\rangle\ \times\ \mathcal{OP}\ \rightarrow\ \mathcal{P}$$
$$ObjSubst(\texttt{nil},\ p)\ =\ ObjToNewVars(p)$$

$$ObjSubst(cons\ ((\texttt{objid}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$$
$$= ObjPropSubst((\texttt{var}\ ox)\ \texttt{equals}\ e,\ eqs,\ ObjChange((\texttt{objid}\ z),\ ox,\ p)),$$
$$\texttt{where}\ ox\ =\ NewVar()$$

$$ObjSubst(cons\ ((\texttt{var}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$$
$$= ObjPropSubst((\texttt{var}\ z)\ \texttt{equals}\ e,\ eqs,\ p)$$

$$ObjPropSubst:\ \mathcal{P}\ times\ \langle EQVars\rangle\ \times\ \mathcal{OP}\ \rightarrow\ \mathcal{P}$$
$$ObjPropSubst(prop,\ \texttt{nil},\ p)\ =\ \texttt{binary}\ prop\ \texttt{implies}\ ObjToNewVars(p)$$

$$ObjPropSubst(prop,\ cons\ ((\texttt{objid}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$$
$$= ObjPropSubst(\texttt{binary}\ prop\ \texttt{and}\ (\texttt{var}\ ox\ \texttt{equals}\ e),\ eqs,$$
$$ObjChange((\texttt{objid}\ z),\ ox,\ p)),\ \texttt{where}\ ox\ =\ NewVar()$$

$$ObjPropSubst(prop,\ cons\ ((\texttt{var}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$$
$$= ObjPropSubst(\texttt{binary}\ prop\ \texttt{and}\ ((\texttt{var}\ z)\ \texttt{equals}\ e),\ eqs,\ p)$$

$$ObjChange:\ \mathcal{O}\ \times\ \mathcal{V}_e^{\bullet}\ \times\ \mathcal{OP}\ \rightarrow\ \mathcal{OP}$$
$$ObjChange(o,\ x,\ \texttt{unary}\ op\ p)\ =\ \texttt{unary}\ op\ ObjChange(o,\ x,\ p)$$

$$ObjChange(o,\ x,\ \texttt{binary}\ p\ op\ q)$$
$$= \texttt{binary}\ ObjChange(o,\ x,\ p)\ op\ ObjChange(o,\ x,\ q)$$

$$ObjChange(o,\ x,\ \texttt{quantor}\ q\ p)\ =\ \texttt{quantor}\ q\ ObjChange(o,\ x,\ p)$$

$$ObjChange(o,\ x,\ e_1\ \texttt{equals}\ e_2)$$
$$= ObjExprChange(o,\ x,\ e_1)\ \texttt{equals}\ ObjExprChange(o,\ x, e_2)$$

$$ObjChange(o,\ x,\ \texttt{var}\ px)\ =\ \texttt{var}\ px$$
$$ObjChange(o,\ x,\ \texttt{constant}\ c)\ =\ \texttt{constant}\ c$$

$$ObjExprChange^{temp}:\mathcal{O}\ \times\ \mathcal{V}_e^{\bullet}\ \times\ \mathcal{E}^{temp}\ \rightarrow\ \mathcal{E}^{temp}$$
$$ObjExprChange^{temp}(o,\ x,\ (\texttt{var}\ y))\ =\ \texttt{var}\ y$$
$$ObjExprChange^{temp}((\texttt{objid}\ z),\ x,\ (\texttt{objid}\ z))\ =\ \texttt{var}\ x$$
$$ObjExprChange^{temp}((\texttt{objid}\ z),\ x,\ (\texttt{objid}\ y))\ =\ \texttt{objid}\ y\ \ \texttt{if}\ y\ \neq\ z$$
$$ObjExprChange^{temp}(o,\ x,\ (basic\ b))\ =\ basic\ b$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{symbol}\ s\ \sigma s\ es))$$
$$= \texttt{symbol}\ s\ \sigma s\ ObjExprChangeList^{temp}(o,\ x,\ es)$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{apply}\ e_1\ \texttt{to}\ e_2))$$
$$= \texttt{apply}\ ObjExprChange^{temp}(o,\ x,\ e_1)\ \texttt{to}\ ObjExprChange^{temp}(o,\ x,\ e_2)$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{case}\ e\ \texttt{of}\ alts))$$
$$= \texttt{case}\ ObjExprChange^{temp}(o,\ x,\ e)\ \texttt{of}\ alts$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{let}\ binds\ \texttt{in}\ e))$$
$$= \texttt{let}\ binds\ \texttt{in}\ ObjExprChange^{temp}(o,\ x,\ e)$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{let!}\ x_1\ =\ e_1\ \texttt{in}\ e_2))$$
$$= \texttt{let!}\ x_1 = ObjExprChange^{temp}(o,\ x,\ e_1)$$
$$\texttt{in}\ ObjExprChange^{temp}(o,\ x,\ e_2)$$

$$ObjExprChange^{temp}(o,\ x,\ (\bot_\sigma)) = \bot_\sigma$$

$$ObjExprChangeList^{temp} : \mathcal{O} \times \mathcal{V}_e^\bullet \times \langle \mathcal{E}^{temp} \rangle \rightarrow \langle \mathcal{E}^{temp} \rangle$$
$$ObjExprChangeList^{temp}(o,\ x,\ \texttt{nil}) = \texttt{nil}$$

$$ObjExprChangeList^{temp}(o,\ x,\ \texttt{cons}\ e\ es)$$
$$= \texttt{cons}\ ObjExprChange(o,\ x,\ e)\ ObjExprChangeList^{temp}(o,\ x,\ es)$$

$$EQVars = \{e_1\ \texttt{equals}\ e_2 \mid e_1 \in \{\texttt{var}\ x \mid x \in \mathcal{V}_e^\bullet\} \cup \mathcal{O},\ e_2 \in \mathcal{E}^{ext}\}$$
$$Parameter : \langle \mathcal{V}_e^\bullet \cup \mathcal{O}^{\texttt{def}} \rangle \times \langle \mathcal{E}^{ext} \rangle \rightarrow \langle EQVars \rangle$$
$$Parameter(\texttt{nil},\ \texttt{nil}) = \texttt{nil}$$

$$Parameter(cons\ (\texttt{obj}\ o\ x)\ xs,\ cons\ e\ es)$$
$$= (cons\ (o\ \texttt{equals}\ e)\ \texttt{nil})\ *\ Parameter(xs,\ es)$$

$$Parameter(cons\ (\texttt{exprvar}\ z)\ xs,\ cons\ e\ es)$$
$$= (cons\ ((\texttt{var}\ (\texttt{exprvar}\ z))\ \texttt{equals}\ e)\ \texttt{nil})\ *\ Parameter(xs,\ es)$$

The definitions above used some simple functions which are not formally defined here. Informally, they calculate the following:

$ObjToNewVars : \mathcal{OP} \rightarrow \mathcal{P}$ Transforms the object identifiers in an object proposition to fresh variables, so it creates an object-less proposition.

$ObjToNewVars_2 : \mathcal{OP} \rightarrow (\mathcal{P}, \langle (\mathcal{O},\ \mathcal{V}_e^\bullet) \rangle)$ Same as the previous, but it returns the applied mapping from objects to variables.

$NewVar : \mathcal{V}_e^\bullet$ It gives a fresh variable.

$ForallPred : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$ Here $ForallPred(q, pred)$ creates a "`forall exprs` x,, quantor to *pred* for every free expression-variable x of q.