Deriving Partial Correctness Logics From Evolving Algebras

Arnd Poetzsch–Heffter [1]

Fakultät für Informatik
Technische Universität
D–80290 München
poetzsch@informatik.tu-muenchen.de

1 Introduction

This extended abstract gives an introduction into the development of partial correctness logics for programming languages specified by evolving algebras. A *partial correctness logic* is a programming logic that allows to prove program properties of the form: "whenever program point P is reached during execution, assertion A is true". We derive a basic axiom (schema) from an evolving algebra and use this axiom to obtain more convenient logics.

This work aims to develop the foundations for programming environments that support formal reasoning about programs. One of the major problems with this challenge is the systematic design of programming logics for realistic programming languages. Experiences e.g. with Hoare logic have shown that it can be difficult to design consistent programming logics even for simple languages from scratch (cf. [O'D81]). Using evolving algebras as semantical basis has two advantages:

1. They support appropriate specifications of control flow (e.g. for jumps and exceptions). This enables to simplify these aspects in the logic.

2. Proving rules sound w.r.t. a given semantics or systematically deriving rules from semantics specification is simpler than proving the consistency of a logic.

Essentially, we relate evolving algebras and temporal logic and apply this combination to develop partial correctness logics. We build on existing approaches to programming logic like [CO78] and [Flo67] and on experiences from research on interface specifications for programming languages (cf. e.g.[GH91]). In this abstract, we focus on the dynamic aspects; aspects of program representation and axiomatization are discussed in [PH94].

Overview Section 2 relates a restricted version of evolving algebras to temporal logic. Section 3 explains a basic logic for evolving algebras. Section 4 shows how this basic logic can be used to proof the soundness of a proof outline logic for a class of simple languages. In section 5, we sketch how partial correctness logics for realistic programming languages can be developed based on the presented approach.

2 Evolving Algebras and Temporal Logic

Evolving algebras are a powerful framework for specifying programming languages (for an introduction see [Gur91]). They have been used to specify the semantics of many different programming languages including C ([GH92]), PROLOG ([BR92]), and Occam ([GM89]). This section provides the needed definitions for evolving algebras and temporal logic. In particular, we introduce so–called simple evolving algebras and show how general evolving algebras can

[1] Supported by DFG grant Po 432/2-1.

be transformed into simple ones. Simple evolving algebras will reduce the technical apparatus for the rest of this abstract.

Definition 2.1 Let Σ be a signature, i.e. a family of finite sets $(FUNC_s)_{s\in\mathbb{N}}$.

- An *update* (over Σ) is an expression of the form $f(\vec{s}) := t$ where $\vec{s}$ is a vector of Σ–ground–terms and t is a Σ–ground–term.
- A *transition rule* is an expression R of the form **if** *guard(R)* **then** *updates(R)* where *guard(R)* is a quantifier–free closed Σ–formula and *updates(R)* is a finite set of updates. A transition rule R is called *consistent*, if its guard implies $\vec{s_1} \neq \vec{s_2}$ for each pair of updates $\langle f(\vec{s_1}) := s_1 \,,\, f(\vec{s_2}) := s_2 \rangle$.
- An *evolving algebra* (over Σ) is given by a finite set of transition rules. It is called *consistent* if all its rules are consistent. It is called *complete*, if the disjunction of the guards is valid. It is called *simple*, if it is consistent, complete, and all updates have the form $f() := t$.

Definition 2.1 follows the introduction of evolving algebras in [GR92]. This setting is a bit different from the one given in [Gur91], in particular the execution semantics (see below). Whereas the latter setting provides more flexibility in writing down specifications, the focus here is to keep formal definitions simple.

Each evolving algebra can be made consistent by splitting a rule R with possibly conflicting updates $\langle f(\vec{s_1}) := s_1 \,,\, f(\vec{s_2}) := s_2 \rangle$ into three rules: One rule is obtained from R by adding $\vec{s_1} \neq \vec{s_2}$ to the guard; the other two rules are obtained from R by deleting one of the conflicting updates and adding $\vec{s_1} = \vec{s_2}$ to the guard. An evolving algebra EA can be made complete by adding a rule that has the conjunction of the negated guards of EA as guard and an empty set of updates.

A consistent and complete evolving algebra EA can be made simple by the following technique: Let f be an a-ary dynamic function symbol, i.e. a function that occurs in outermost position on the left–hand side of an update. By using an $a+1$-ary function $apply_f$ to apply f to an argument vector and a $a+2$-ary function upd_f to update f at an argument vector by a value, we can transform EA into a simple evolving algebra as follows: each function update $f(\vec{s}) := t$ becomes $f := upd_f(f, \vec{s}, t)$ and each remaining function application $f(\vec{s})$ becomes $apply_f(f, \vec{s})$. If there is a rule in EA with a simultaneous update of f, e.g. $f(\vec{s_1}) := s_1 \,,\, f(\vec{s_2}) := s_2$, these updates have to be packed together in one update, e.g. $f := upd_f(\, upd_f(f, \vec{s_1}, s_1), \vec{s_2}, s_2)$. As the guard guarantees $\vec{s_1} \neq \vec{s_2}$, the order of the packing is irrelevant. For a discussion of this transformation, the following remarks may be helpful:

- It is a standard technique to express higher–order functions in a first–order setting.
- Handling functions by *apply* and *upd* is technically the same as handling lists by $first$, $rest$, and $append$; in particular, algebraic methods can be used to reason about the underlying data types (see e.g. [BW82]).
- The transformation can be performed automatically and therefore be hidden by a system so that users need not care about it.

As all dynamic function symbols of a simple evolving algebra are of arity zero, we call them *dynamic variables*.

Temporal Logic This paragraph introduces the needed concepts[2] of temporal logic and sketches the tight relation between simple evolving algebras and temporal logic.

Definition 2.2 Let Σ be a signature and VAR be a set of variables. The set of *(temporal) formulas* over Σ is inductively defined as follows:

- if t_1, t_2 are Σ–terms, then $t_1 = t_2$ is a formula;
- if F, G are formulas and X is a variable, then $\neg F$, $F \rightarrow G$, $\circ F$, $\Box F$, and $\forall X.F$ are formulas.

The operators $\circ$ and $\Box$ express temporal properties: $\circ F$ is valid in the current state if F is valid in the next state; $\Box F$ is valid in the current state if F is valid in the current and all future states. A formula not containing a temporal operator is called a *state formula.*

In temporal logic, formulas are interpreted w.r.t. computations. As we work with simple evolving algebras, we can use the following definition of "computation" for both frameworks:

Definition 2.3 Let Σ be a signature, SEA a simple evolving algebra, and $DVAR$ the dynamic variables of SEA. A *computation* $\mathcal{C}$ is given by

- a set U called the universe;
- an interpretation φ of the non-dynamic function symbols, $\varphi = (\varphi^s)_{s \in \mathbb{N}}$ with $\varphi^s : FUNC_s \rightarrow \mathcal{F}(U^s, U)$ where $\mathcal{F}(U^s, U)$ denotes the functions from U^s to U;
- a sequence of valuations $(\eta_i)_{i \in \mathbb{N}}$ of the dynamic variables, $\eta_i : DVAR \rightarrow U$; η_i is called the i-th *state* of the computation.

Definition 2.3 reflects the temporal logic semantics. It only allows dynamic variables to change their interpretation during a state transition. This makes the logic simpler[3]. The validity of temporal formulas in a computation is defined as usual (cf. [Krö87]). Definition 2.3 provides as well the semantics for simple evolving algebras: A computation is a *computation of a simple evolving algebra* SEA if for each $i \in \mathbb{N}$ there is a rule R such that *guard(R)* is true in algebra (U, φ, η_i) and the updates of R lead to state η_{i+1}.

3 The Basic Logic of Simple Evolving Algebras

This section presents the basic axiom (schema) that expresses a simple evolving algebra in logical terms. To show where this axiom comes from, the section introduces invariance properties and their proofs by computational induction. In the following, we assume that $\mathcal{PL}$ is a programming language specified by the simple evolving algebra SEA and Π is a $\mathcal{PL}$ program.

A state formula INV expresses an *invariance property* of Π (or is an invariant of Π) if INV is valid in every state of a computation of SEA that executes Π. This can be expressed by the formula

$$START_\Pi \rightarrow \Box\, INV$$

where $START_\Pi$ is a state formula expressing the relevant properties of the initial execution state of Π.

[2] The introduction of other operators and concepts is straightforward (cf. e.g. [Krö87]).

[3] E.g. g(X)=f(X)→ ∘(g(X)=f(X)) is valid in temporal logic, but not if the interpretation of function symbols can change in a computation step.

Invariance properties are classically proved by computational induction, i.e. by showing that the property holds in the initial state and is invariant during computation:

$$1.\ START_\Pi \rightarrow INV \qquad\qquad 2.\ INV \rightarrow \circ\, INV$$

Whereas the proof obligation for the initial state (1.) can be shown by classical predicate logic using static (syntactic) program properties, the second obligation can only be proved by refering to the dynamic semantics of $\mathcal{PL}$. To capture SEA by logical means, we define a weakest backward transformer **wb** on state formulas: $\mathbf{wb}[F]$ is the weakest formula such that the validity of $\mathbf{wb}[F]$ in the current state implies the validity of F in the next state; i.e. for any state formula F, we have the axiom $\mathbf{wb}[F] \rightarrow \circ F$. This approach slightly extends the sentential operator defined in [GR92], p. 189, in that **wb** is defined for an evolving algebra and not only for one rule[4]. Essentially, it applies the weakest precondition transformer of Hoare logic to evolving algebras, i.e. to the semantic formalism instead of to program parts: The transformer does not yield the precondition of a statement, but transforms a (global) property of a state into a (sufficient) property of previous states.

Definition 3.1 Let SEA = $\{ R_1, \ldots, R_n \}$ and let us denote the vector of dynamic variables occurring on the left hand side in the updates of R_i by $\vec{x}_i$ and the vector of right hand sides by $\vec{s}_i$. Furthermore, let F be a state formula. Denoting the simultaneous substitution of $\vec{y}$ by a vector of terms $\vec{u}$ in a formula F by $F[\vec{y}/\vec{u}]$, the weakest backward transformer **wb** is defined as follows:

$$\mathbf{wb}[F] \equiv_{def} \bigwedge_{i=1}^{n} (\ guard(R_i) \ \rightarrow \ F[\vec{x}_i/\vec{s}_i] \ ) \ .$$

The validity of the axiom $\mathbf{wb}[F] \rightarrow \circ F$ w.r.t. SEA (i.e. its validity in all computations of SEA) follows from the defintion of simple evolving algebra computations and the definition of the nexttime operator $\circ$. On the other hand, it is not difficult to show that every computation in which $\mathbf{wb}[F] \rightarrow \circ F$ is valid (for every state formula F) is a computation of SEA.

The axiom $\mathbf{wb}[F] \rightarrow \circ F$ is the key to prove the induction step of the computational induction (and thereby invariance properties): $INV \rightarrow \circ\, INV$ can be proved by showing $INV \rightarrow \mathbf{wb}[INV]$; and that reduces the problem to predicate logic reasoning. In summary, the *basic programming logic* consists of the axioms and rules of temporal logic, the axiom $\mathbf{wb}[F] \rightarrow \circ F$, and the non–temporal axioms expressing the static program properties and the properties of the basic data types (not shown).

4 Deriving Partial Correctness Logics

The basic logic of the previous section is rather inconvenient for practical purposes. In this section, we use the basic logic to derive a practical partial correctness logic for a class of sequential deterministic programming languages without procedures. This class is characterized as follows:

- its programs are represented by control flow or *task graphs*[5] (cf. [GH92] or [PH94] for task graphs);
- a task t is either a branch having the two successors $ttsucc(t)$ and $ffsucc(t)$; or it is of some sort "Task$_{kind}$" (where *kind* ranges over a finite set) and has a unique successor $succ(t)$;

[4] On the other hand, **wb** is more restrictive, because it is only defined for simple evolving algebras.

[5] We assume here that the graph is connected and contains a unique start and end task.

- each evolving algebra rule has one of the following forms (the dynamic variable CT denotes the current task, and UPDATES is a set of updates not containing CT):

```
if  isBranch(CT)=true ∧ cond(CT)=true  then  CT:=ttsucc(CT)
if  isBranch(CT)=true ∧ cond(CT)=false then  CT:=ffsucc(CT)
if  isTask_kind(CT)=true               then  CT:=succ(CT)     UPDATES
```

Let t_0 be a task of a given program Π of a language in that class. A *partial correctness property at* t_0 is given by a state formula $PROP_{t_0}$ not containing CT. Such a property is satisfied if it is valid in all states where execution is at t_0, i.e. iff the following formula is valid

$$START_\Pi \ \rightarrow \ \Box(\mathrm{CT} = t_0 \ \rightarrow \ PROP_{t_0}) \quad .$$

Partial correctness w.r.t. a precondition P can be considered by adding P to the start condition $START_\Pi$. Partial correctness of a program is partial correctness of its end task.

To prove a partial correctness property $PROP_{t_0}$, we have to find a program invariant that implies the above formula. A standard way is to use an invariant INV of the form

$$\bigwedge_{t \in Task} (\ \mathrm{CT} = t \ \rightarrow \ PROP_t) \quad .$$

The non trivial part is of course to chose the $PROP_t$ such that INV can be proved to be invariant in Π. That is why the annotation of the tasks by $PROP_t$ is often called a *proof outline*. A proof outline is called *valid* if INV is an invariant of Π. According to section 3, the essential part of proving INV to be invariant is to show

$$INV \ \rightarrow \ \mathbf{wb}[INV] \quad . \tag{4.1}$$

For realistic size programs with thousands of tasks, this formula is even mechanically difficult to handle (for each rule of the evolving algebra there is a conjunction ranging over the number of tasks). But, it can be used to derive proof obligations of a simpler form. For our language class, we can formally prove that the following proof obligations imply (4.1). (The proof uses the fact that **wb** distributes over $\wedge$ and that $\bigvee_{t \in Task} \mathrm{CT} = t$ is an invariant of every computation which in turn is proved by the basic logic.)

Proof Obligations Let Π be a $\mathcal{PL}$ program and t, t' be two tasks.

- if $ttsucc(t) = t'$ prove $\quad PROP_t \ \wedge \ cond(t) = true \ \rightarrow \ PROP_{t'}$;
- if $ffsucc(t) = t'$ prove $\quad PROP_t \ \wedge \ cond(t) = false \ \rightarrow \ PROP_{t'}$;
- if $succ(t) = t'$ and $\mathrm{isTask}_{kind}(t)$ prove $\ PROP_t \ \rightarrow \ PROP_{t'}[\vec{x}_{kind}/\vec{s}_{kind}]$
  where $\vec{x}_{kind}$ denotes the dynamic variables occurring on the left hand side in the updates of the rule for isTask_{kind}, and $\vec{s}_{kind}$ denotes the right hand sides.

The above proof obligations are the result of systematically simplifying formula (4.1). If the proof outline consisting of the $PROP_t$ is selected carefully, the proof obligations can be discarded automatically. And in cases in which this is not possible, an interactive proof environment can point to those tasks where proof obligations could not be discarded.

5 Towards Realistic Programming Logics

To focus on the central aspects of deriving partial correctness logics, the previous section applied the method to a rather simple, well studied language class. This should not be misleading. As the method is based on a general framework for specifying language semantics, it applies to the systematic design of programming logics for realistic programming languages. Of course, even a *s*ystematic design of a logic for a complex language is a non trivial task. The advantage of the presented method over designing programming logics from scratch is that the meaning of higher–level logics can be precisely expressed in the basic logic and that the logic can be proved sound w.r.t. the language semantics. In this section, we discuss some of the problems that arise when applying the method to realistic languages.

Simplifying Proof Outlines In the previous section, we required that a proof outline contains an assertion $PROP_t$ for each task t of the given program. Even if users are supported by powerful graphical interactive environments, this would lead to systems that are very tedious to use. Fortunately, proof outlines can be tremendously simplified: It is sufficient to give an assertion $PROP_t$ for (at least) one task of each loop. Assertions for the other tasks can be computed using **wb** in almost the same way as weakest preconditions are computed in Hoare logic.

Nondeterminism Evolving algebras allow to express nondeterminism of programming languages in different ways. To mention only two let us consider the nondeterministic evaluation order of expressions in C:

- Implicit: Represent the evaluation state of an expression E by a dynamic function recording which parts of E are evaluated. That is the technique used in [GH92].
- Explicit: Construct a task thread for each possible evaluation sequence[6].

For explicitly specified nondeterminism, programming logics can be derived as explained in the previous section; the only difference is that *succ* is no more a function, but a relation. In cases in which the implicit specification of nondeterminism is a better choice, the derivation of programming logics depends on how the dynamic functions record the evaluation state. To systematically derive a programming logic for these cases, it is helpful to consider the implicit specification as a coding of the explicit one.

Handling Procedures Based on the above approach, languages with procedures could be handled by using invariants INV of the form

$$\bigwedge_{t \in Task} (\mathrm{CT} = t \wedge top(\mathrm{CTRLSTACK}) = callsite \rightarrow PROP_{t,callsite}) \qquad (5.1)$$

where CTRLSTACK denotes the current control stack containing the call sites of the active procedures. A proof outline would consist of assertions $PROP_{t,callsite}$ for each task t and each call site *callsite* of the enclosing procedure of t.

The resulting logic has two disadvantages: First, the user has to provide too many assertions. And second, the logic does not support procedural abstraction; i.e. the behaviour of a procedure is not specified by relating input, side effects, and output, but by describing only those invariance properties that are relevant for the calling contexts occurring in the program. Proof outlines supporting procedural abstraction can be considered as enriching procedure specifications (cf. [Luc90, GH91]) by assertions for the tasks in the procedure bodies. Such proof outlines consist of

[6] There is only a finite number.

- a precondition $PREC_p$ for each procedure p (often called the "requires" clause); $PREC_p$ may contain free variables to record the values of the input parameters;
- a postcondition $POSTC_p$ for each procedure that may use the variables introduced in $PREC_p$ (often called the "ensures" clause);
- assertions $PROP_t$ for each task t.

For brevity, let us assume (a) that the whole program is given as a procedure, (b) that each procedure has a unique end task t_p such that $POSTC_p$ equals $PROP_{t_p}$, and (c) that the evolving algebra provides a dynamic variable CI that records the current incarnation number of a procedure[7]. Under these assumptions, the meaning of the above proof outlines can be formalized by the following formula:

$$\bigwedge_{p \in Procedure} \; (\, \mathrm{CT} = \mathit{before}(p) \wedge \mathrm{CI} = \mathit{INCAR} \wedge \mathit{PREC}_p \rightarrow \Box \, (\bigwedge_{t \in Task(p)} (\, \mathrm{CT} = t \wedge \mathrm{CI} = \mathit{INCAR} \rightarrow \mathit{PROP}_t)) \;) \tag{5.2}$$

Formula (5.2) guarantees that if we start execution of some incarnation $INCAR$ of procedure p in a state where $PREC_p$ is valid, then the assertions $PROP_t$ are valid in all states of $INCAR$ where t is the current task. This holds in particular for the end task of p. Whereas in section 4 we had a precondition for the whole program ($START_\Pi$) and a conjunction over all tasks, we have here a formula of the same form for each procedure (the only difference is the reference to the current incarnation). As in section 4, we have to derive practical proof obligations that are sufficient to prove (5.2). In addition to proof obligations similar to section 4, we get proof obligations for procedure calls and returns. To show that these proof obligations really imply (5.2) is not as simple as for the non–procedural case, in particular in the presence of recursive procedures. The proof that we carried out for an example language is based on the invariance technique given by formula (5.1).

The presented work is only a first step towards deriving programming logics from programming language specifications. Our next steps are the application of the approach to a realistic programming language and the development of reasoning tools, in particluar in connection with program specifications. In addition to this, it is certainly interesting to apply this approach to parallel languages and to the systematic development of stronger programming logics, e.g. total correctness logics.

References

[BR92] E. Börger and D. Rosenzweig. A simple mathematical model for full Prolog. Technical Report TR–33/92, Dipartimento di Informatica, Universita di Pisa, 1992.

[BW82] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer–Verlag, 1982.

[CO78] R. L. Constable and M. J. O'Donnell. *A Programming Logic: With an Introduction to the PL/CV Verifier.* Winthrop Publishers, 1978.

[7] Such a "counter" can be easily added if necessary (cf. [PH94]).

[Flo67] R. W. Floyd. Assigning meanigs to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967. American Math. Soc.

[GH91] John J. Guttag and James J. Horning. A tutorial on Larch and LCL, a Larch/C interface language. In S. Prehn and W. J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, 1991. LNCS 552.

[GH92] Y. Gurevich and J. Huggins. The evolving algebra semantics of C: Preliminary version. Technical Report CSE–TR–141–92, EECS Department, University of Michigan, 1992.

[GM89] Y. Gurevich and L. Moss. Algebraic operational semantics and Occam. In E. Börger et al., editor, *Computer Science Logic*, pages 176–192, 1989. LNCS 440.

[GR92] P. Glavan and D. Rosenzweig. Communication evolving algebras. In E. Börger et al., editor, *Computer Science Logic*, pages 182–215, 1992. LNCS 702.

[Gur91] Y. Gurevich. *Evolving Algebras*, volume 43, pages 264–284. EATCS Bulletin, 1991.

[Krö87] F. Kröger. *Temporal Logic of Programs*. Springer–Verlag, 1987.

[Luc90] David C. Luckham. *Programming with Specifications: An Introduction to Anna. A Language for Specifying Ada Programs*. Springer-Verlag, 1990.

[O'D81] Micheal J. O'Donnell. A critique of the foundations of hoare–style programming logics. In D. Kozen, editor, *Logics of Programs*, pages 349–374, 1981. LNCS 131.

[PH94] A. Poetzsch-Heffter. Developing efficient interpreters based on formal language specifications. In P. Fritzson, editor, *Compiler Construction*, 1994. appears in LNCS.