Transformations of Evolving Algebras

Stephan Diehl

FB 14 - Informatik
Universität des Saarlandes
Postfach 15 11 50
66041 Saarbrücken
GERMANY
diehl@cs.uni-sb.de

in Proceedings of the VIII Conference on Logic and Computer Science LIRA'97, Novi Sad, Yugoslavia, September 1-4, 1997, pp. 51-57

Transformations of Evolving Algebras

Stephan Diehl, Universität des Saarlandes

Abstract. We give a precise definition of evolving algebras as nondeterministic, mathematical machines. All proofs in the paper are based on this definition. First we define constant propagation. We extend evolving algebras by macros and define folding and unfolding transformations. Next we introduce a simple transformation to flatten transition rules. Finally a pass separation transformation for evolving algebras is presented. It can be used to derive a compiler and abstract machine from an interpreter. All transformations are proven correct. Finally a comparison to other work is given.

1 Introduction

Evolving algebras (EvAs) have been proposed by Gurevich in [Gur91] and used by Gurevich and others to give the operational semantics of languages like C, Modula-2,Prolog and Occam. Börger and Rosenzweig's proof of the correctness of the Warren Abstract Machine is based on a slight variation of evolving algebras ([BR92]). An evolving algebra may be tailored to the abstraction level necessary for the intended application of the semantics, e.g. we might have a hierarchy of evolving algebras, each being more concrete with respect to certain aspects of the semantics. In this paper we only discuss syntactic-sugar free evolving algebras. As a result reading descriptions of an EvA using this notation is harder than reading descriptions, which make extensive use of syntactic-sugar. The advantage of considering the syntactic-sugar free EvAs is clearly, that we have to deal with less constructs when we define EvAs and a variety of transformations, as well as, when we prove operational equivalence and other properties.

Syntactic-sugar free EvAs For our purposes here, we need a precise definition of what an EvA is, and what a computation of an EvA looks like. An **evolving algebra** Ψ is a quadruple $< \sigma, S, T, \mathcal{I}_0 >$ where [1] σ is a **signature**, i.e. a finite set of function names with associated arity, S is a nonempty set, called the **superuniverse**, T is a finite set of transition rules and $\mathcal{I}_0 : \sigma \rightarrow \bigcup_{n \geq 0}(S^n \rightarrow S)$ is the initial interpretation of functions in σ, i.e. $\mathcal{I}_0$ maps every function name f of arity n to an interpretation function $\mathcal{I}_0(f) : S^n \rightarrow S$.
Transition rules are either **function updates** $\boxed{f(t_1, ..., t_n) := t_0}$, where $f \in \sigma$, $n \geq 0$ is the arity of f and the t_i are terms, or **guarded updates** $\boxed{\textit{if } b \textit{ then } C}$, where b is a term and C is a set of transition rules. A term t is either of the form $f(t_1, ..., t_n)$, where $f \in \sigma$,$n \geq 0$ is the arity of f and the t_i are terms, or $t \in S$.
A function update changes the interpretation of a function f for the arguments $t'_1, \ldots, t'_n$ to the value t'_0, where t'_i is the value of the term t_i in the current interpretation. In a guarded update the updates in C are only executed, if the guard b is true in the current interpretation.

[1] We will assume $\{true, false\} \subseteq S$.

We will use the notation $\mathcal{I} \xrightarrow{\Psi} \mathcal{I}'$ to indicate, that $\mathcal{I}'$ is the result of applying the transition rules of Ψ to $\mathcal{I}$. We will call this a **step** of the evolving algebra. Before we can define a step of an EvA, we have to introduce some notation. First we define the value of a term t in an interpretation $\mathcal{I}$ and the evaluated form of a function update:
$eval(f(t_1,...,t_n),\mathcal{I}) = \quad \mathcal{I}(f)(eval(t_1,\mathcal{I}),\dots,eval(t_n,\mathcal{I}))\ for\ n \geq 0$
$eval(f(t_1,...,t_n) := t_0,\mathcal{I}) = f(eval(t_1,\mathcal{I}),\dots,eval(t_n,\mathcal{I})) := eval(t_0,\mathcal{I})\ for\ n \geq 0$
Let T be a set of transition rules and $\mathcal{I}$ be an interpretation, then those function updates occurring in T can be executed, which either depend on guards evaluating to true in the interpretation or on no guard at all. We define $updates(T,\mathcal{I}) = \{eval(u,\mathcal{I}) : u \in T \wedge u$ is a function update$\} \cup\ updates(U,\mathcal{I})$ where U is the union of all C, such that $\boxed{if\ b\ then\ C} \in T$ and $eval(b,\mathcal{I}) = true$.
There can be several conflicting function updates in $updates(T,\mathcal{I})$, i.e. evaluated function updates, which change the interpretation of a function for the same arguments to different values. Let M be a set of evaluated function updates, then $\overline{M}$ denotes the set of all greatest subsets A of M, such that if $\boxed{f(t_1,\dots,t_n) := t_0}$ in A then there is no update $\boxed{f(t_1,\dots,t_n) := t_0'}$ in A where $t_0 \neq t_0'$. The relation $\xrightarrow{\Psi}$ is defined as follows: $\mathcal{I} \xrightarrow{\Psi} \mathcal{I}' \Leftrightarrow \exists U \in \overline{updates(T,\mathcal{I})}\quad \forall \tilde{a} \in S^*, s \in S, f \in \sigma$:
$$\mathcal{I}'(f)(\tilde{a}) = \begin{cases} s & if\ \boxed{f(\tilde{a}) := s} \in U \\ i & if\ f\ is\ an\ external\ function\ \ (for\ some\ i \in S) \\ \mathcal{I}(f)(\tilde{a}) & otherwise \end{cases}$$
Note, that if $\overline{updates(T,\mathcal{I})}$ is not a singleton, then from every set of conflicting updates only one member is chosen nondeterministically.
A terminating **computation** of an evolving algebra Ψ is a sequence $< \mathcal{I}_0, \mathcal{I}_1, ..., \mathcal{I}_k >$, such that $\mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_1 \xrightarrow{\Psi} ... \xrightarrow{\Psi} \mathcal{I}_k$ and $updates(T,\mathcal{I}_k) = \emptyset$. Sometimes we will use the notation $\mathcal{I}_0 \overset{\Psi}{\twoheadrightarrow} \mathcal{I}_k$ to refer to a computation. Furthermore the set $reach(\mathcal{I}_0)$ is defined as $\{\mathcal{I}_m : \exists \mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_1 \xrightarrow{\Psi} \dots \xrightarrow{\Psi} \mathcal{I}_m\}$.

Proof Method Let Ψ and Ψ' be EvAs and $\mathcal{F}$ be a partial mapping of interpretations in Ψ' to those in Ψ. Then Ψ' is **correct** wrt. Ψ iff $\mathcal{I}_0 = \mathcal{F}(\mathcal{I}_0')$ and for every terminating computation $\mathcal{I}_0' \overset{\Psi'}{\twoheadrightarrow} \mathcal{I}_k'$ there is a terminating computation $\mathcal{I}_0 \overset{\Psi}{\twoheadrightarrow} \mathcal{F}(\mathcal{I}_k')$. Furthermore Ψ is **complete** wrt. Ψ, iff for every terminating computation $\mathcal{I}_0 \overset{\Psi}{\twoheadrightarrow} \mathcal{I}_k$ there is a terminating computation $\mathcal{I}_0' \overset{\Psi'}{\twoheadrightarrow} \mathcal{I}_m'$ such that $I_k = \mathcal{F}(\mathcal{I}_m')$. If Ψ' is both correct and complete wrt. Ψ, then Ψ' and Ψ are **operational equivalent**. The proof method is discussed in more detail in [BR92].

2 Transformations

Constant Propagation In evolving algebras functions are classified as internal or external. External functions mimic input to the evolving algebra, i.e. how their interpretation changes at each step of the evolving algebra can not be foreseen. An internal function f is called static, if there is no function update to f in the transition rules. We will extend this classification by allowing external functions to be static or dynamic. We will call an external function static, if we know its value on all arguments a priori. We actually turn an external function into an internal static one. Now we will show, how a given EvA can be partially evaluated with respect

to its static functions. First we define the result of constant propagation $\pi(t)$ of a term t. If $t \equiv f(t_1, ..., t_n)$ and f is static then $\pi(t) = \mathcal{I}(f)(\pi(t_1), ..., \pi(t_n))$ else $\pi(t) = f(\pi(t_1), ..., \pi(t_n))$. A term is defined to be static, if it does not contain any dynamic function, i.e. t is static iff $t \in S$ or $t = f(t_1, ..., t_n)$ where $n \geq 0$ and all t_i and the function f are static.
Let C be a set of transition rules. We construct the set $\pi(C)$ of the transition rules after constant propagation by induction. $\pi(C)$ is also called the residual of C. For all $r \in C$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $\boxed{f(\pi(t_1), ..., \pi(t_n)) := \pi(t_0)} \in \pi(C)$. If $r \equiv \boxed{if\ b\ then\ D}$ and $\pi(b) \notin \{true, false\}$ then $\boxed{if\ \pi(b)\ then\ \pi(D)} \in \pi(C)$. Finally, If $r \equiv \boxed{if\ b\ then\ D}$ and $\pi(b) = true$ then $\pi(D) \subseteq \pi(C)$.
Theroem: Let $\Psi = < \sigma, S, T, \mathcal{I}_0 >$ and let $\pi(\Psi)$ denote the residual $< \sigma, S, \pi(T), \mathcal{I}_0 >$ of Ψ. Then $\pi(\Psi)$ is operationally equivalent to Ψ.
Proof: After constant propagation in the resulting algebra the same updates are done as before, we only changed the amount of work which is necessary to evaluate terms. So the inital interpretation and the terminal interpretations are preserved (correctness). Furthermore for every terminating computation in Ψ there is a terminating computation in $\pi(\Psi)$ (completeness). The operational equivalence follows immediately from the correctness and completeness. $\square$

Macro Definitions Readability of an evolving algebra can be increased, if we define functions in terms of other functions. First we might think of macro definitions as simple combinations of functions like $snd = fst \circ rest$ implying $\mathcal{I}(snd) = \mathcal{I}(fst) \circ \mathcal{I}(rest)$. But this is not powerful enough. So we will consider macro definitions of a different form,e.g. $mult\_twice(x, y) = mult(plus(x, x), plus(y, y))$, which is to imply $\forall x, y \in S : \mathcal{I}(mult\_twice)(x, y) = \mathcal{I}(mult)(\mathcal{I}(plus)(x, x), \mathcal{I}(plus)(y, y))$.
Let $\tilde{\sigma}$ be the set of all static functions in σ, $f \in \tilde{\sigma}$ and t_0 be a first-order term consisting of function names in $\tilde{\sigma}$ and $x_1, \ldots, x_n$ distinct variables, then a macro definition is of the form $f(x_1, \ldots, x_n) = t_0$. A macro definition is **valid**, iff $eval(f(s_1, \ldots, s_n), \mathcal{I}) = eval(t_0[x_1 \mapsto s_1, \ldots, x_n \mapsto s_n], \mathcal{I})$ for all $s_i \in S$ and all $\mathcal{I} \in reach(\mathcal{I}_0)$.
We have several choices to restrict macros: no additional restrictions on the macros (1), allow only non-recursive definitions (2) or none of the macros defined, may occur in the right hand side of a macro definition (3). We will address the implications of these restrictions in the next section.

Unfolding Macros Let Δ be a set of macro definitions. First we define the Δ-unfolding of a term t, which we will write as $t \uparrow \Delta$. If $t \equiv f(t_1, ..., t_n)$ and $(f(x_1, ..., x_n) = t_0) \in \Delta$ then $t \uparrow \Delta = t_0[x_1 \mapsto t_1 \uparrow \Delta, ..., x_n \mapsto t_n \uparrow \Delta]$ else $t \uparrow \Delta = t$
We will denote $\underbrace{t \uparrow \Delta ... \uparrow \Delta}_{n\ times}$ by $t \uparrow^n \Delta$. The above mentioned restrictions on macro definitions have the following implications with respect to the Δ-unfolding of a term: it is possible, that there is no n such that $t \uparrow^n \Delta = t \uparrow^{n+1} \Delta$, e.g. $\Delta = \{f(x) = f(x)\}$ there is an n such that $t \uparrow^n \Delta = t \uparrow^{n+1} \Delta$ or $t \uparrow \Delta = t \uparrow^2 \Delta$.
Now we define the Δ-unfolding of a set of transition rules T, which we will write as $T \uparrow \Delta$. Let $r \in T$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $\boxed{f(t_1 \uparrow \Delta, ..., t_n \uparrow \Delta) := t_0 \uparrow \Delta} \in T \uparrow \Delta$. If $r \equiv \boxed{if\ b\ then\ D}$ and $b \uparrow \Delta \notin \{true, false\}$ then $\boxed{if\ b \uparrow \Delta\ then\ D \uparrow \Delta} \in T \uparrow \Delta$. Finally, if $r \equiv \boxed{if\ b\ then\ D}$ and $b \uparrow \Delta = true$ then $D \uparrow \Delta \in T \uparrow \Delta$

Theorem: Let $\Psi =< \sigma, S, T, \mathcal{I}_0 >$, let Δ be a set of valid macro definitions and let $\Psi \uparrow \Delta$ denote the evolving algebra $< \sigma, S, T \uparrow \Delta, \mathcal{I}_0 >$. Then $\Psi \uparrow \Delta$ is operationally equivalent to Ψ.
Proof: In the unfolded algebra the same updates are done as before, we only changed the structure of the terms, not their interpretation, i.e. the value they evaluate to. The operational equivalence follows by the same argument used for the proof in the previous section. $\square$

Folding Macros As before let Δ be a set of macro definitions. First we define the Δ-folding of a term t, which we will write as $t \downarrow \Delta$. Furthermore we will use $\sqcap$ to denote unification of first-order terms.If $t \equiv f(t_1, ..., t_n)$ and $t_i^* \in t_i \downarrow \Delta$ then $f(t_1^*, ..., t_n^*) \in t \downarrow \Delta$. Furthermore, if $f(t_1, ..., t_n)$ and t_0 are unifiable, i.e. $f(t_1, ..., t_n) \sqcap t_0$ is defined and $g(x_1, ..., x_m) = t_0 \in \Delta$ then $g(\hat{x}_1, ..., \hat{x}_m) \in t \downarrow \Delta$, where the $\hat{x}_i$ are terms, such that $f(t_1, ..., t_n) = t_0[x_1 \mapsto \hat{x}_1, ..., x_m \mapsto \hat{x}_m]$ Note, that in an implementation we do not need an occurs check here, because we always unify a variable free term and a term. Now we define the Δ-folding of a set of transition rules T, which we will write as $T \downarrow \Delta$. Let $r \in T$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $\{\boxed{f(t_1^*, ..., t_n^*) := t_0^*}\} \cup T^* \in T \downarrow \Delta$, where $t_i^* \in t_i \downarrow \Delta$ and $T^* \in T \setminus \{r\} \downarrow \Delta$. If $r \equiv \boxed{if\ b\ then\ D}$ and $b \uparrow \Delta \notin \{true, false\}$ then $\{\boxed{if\ b^*\ then\ D^*}\} \cup T^* \in T \downarrow \Delta$, where $b^* \in b \downarrow \Delta, D^* \in D \downarrow \Delta$ and $T^* \in T \setminus \{r\} \downarrow \Delta$ Note, that $T \downarrow \Delta$ is the set of all possible foldings of the rules in T.
Theorem: Let $\Psi =< \sigma, S, T, \mathcal{I}_0 >$. Let Δ be a set of macro definitions and $T^* \in T \downarrow \Delta$. $< \sigma, S, T^*, \mathcal{I}_0 >$ is operationally equivalent to Ψ.
Proof: In the folded algebra the same updates are done as before, we only changed the structure of the terms, not their interpretation, i.e. the value they evaluate to. The operational equivalence follows by the same argument used for the proofs in the previous sections. $\square$
Clearly, in practice we are interested in one set of folded rules. Thus in an implementation we would have to choose one $T^* \in T \downarrow \Delta$. The choice can be based on heuristics. Both, folding and unfolding transformations did only change the terms occuring in rules. Next we will address transformations, which change the structure of a set of rules.

Flattening Next we consider a simple transformation, which is helpful to prepare a set of rules to apply other transformations. Let C be a set of rules, then we construct the set of flat rules $\mathcal{F}(C)$ as follows. For each $r \in C$ we have: If $r \equiv \boxed{if\ b_1\ then\ D} \in C$ then $\{\boxed{if\ b_1\ then\ u} : u \in D$ is function update $\} \cup \{\boxed{if\ b_1 \& b_2\ then\ u} : \boxed{if\ b_2\ then\ u} \in \mathcal{F}(D)\} \subseteq \mathcal{F}(C)$. If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $r \in \mathcal{F}(C)$. For this construction to be semantics preserving, the interpretation of & has to be $\forall \tilde{a} \in S : \mathcal{I}(\&)(\tilde{a}) = \begin{cases} true & if\ \tilde{a} = (true, true) \\ false & otherwise \end{cases}$
Note that in the definition of a computation of an EvA, we defined *updates*, such that the rules of a guarded update are only considered, if the condition evaluates to *true*. Flattening and its inverse transformation ("crushing"), can be used to restructure a set of rules, e.g.: $\{if\ b_1\ then\ \{u_1,\ if\ b_2\ then\ u_2\},\ if\ b_1\ then\ u_3\}$ can be transformed into $\{if\ b_1 \& b_2\ then\ u_2,\ if\ b_1\ then\ \{u_1, u_3\}\}$

Pass Separation Now we will classify dynamic functions as compile-time or run-time functions. The value of a compile-time function is known, before that of a run-time function, e.g. in an interpreter we might consider the program as compile-time data and the input to the program as run-time data. The idea is now to classify the rules: There is one group of rules, which depend only on compile-time functions and the remaining rules depend on compile-time or run-time functions. In practice we consider some of the external functions not to be known before run-time. Since other dynamic functions can depend on these functions, we have to classify these dynamic functions as run-time functions, too. In the literature on partial evaluation (e.g. [JGS93]) this process is called binding-time analysis.
Classification of Functions: Let R be the initial set of run-time functions and $\Psi =< \sigma, S, T, \mathcal{I}_0 >$. Now we classify the functions in S as follows:

1. Let $R' = R$
2. For all $r \in \mathcal{F}(T)$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ and there is a function name $g \in R'$, such that g occurs at least in one of the terms $t_0, ..., t_n$, then $f \in R'$. If $r \equiv \boxed{if\ b\ then\ f(t_1, ..., t_n) := t_0}$ and there is a function name $g \in R'$, such that g occurs at least in one of the terms $b, t_0, ..., t_n$, then $f \in R'$.
3. If $R' = R$ then return R else set $R := R'$ and goto 2

Now the set of all compile-time functions is just $C = \sigma - R$. Note, that all static functions are classified as compile-time functions. The classification of functions terminates in time $O(|\sigma|)$, because in each iteration the $|R'|$ decreases and $|R'| < |\sigma|$.
Classification of Rules: Next we have to classify rules as compile- or run-time rules: $r \in T$ is a run-time rule, if $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ and there occurs at least one run-time function in one of the terms $t_0, ..., t_n$, or if $r \equiv \boxed{if\ b\ then\ D}$ and there occurs at least one run-time function in b or there is a run-time rule in D. Otherwise r is a compile-time rule. This classification of rules terminates in time $O(|T|)$.
For the pass separation transformation, we require that the top-level conditions in the run-time rules are **mutually exclusive**, i.e. if $\{\boxed{if\ b_1\ then\ u_1}, \ldots, \boxed{if\ b_n\ then\ u_n}\}$ is the set of all run-time rules in T, then we require: for all interpretations $\mathcal{I} \in reach(\mathcal{I}_0) : eval(b_k, \mathcal{I}) = true \Rightarrow$ for all $i \neq k : eval(b_i, \mathcal{I}) = false$
An evolving algebras is **separable**, if the top-level conditions of the run-time rules are mutually exclusive and consist of compile-time functions only, and if there occurs no term $f(t_1, ..., t_n)$ in any of the run-time rules, where f is a dynamic compile-time function and a run-time function occurs in at least on of the t_i.
Now we construct two evolving algebras: one which generates a program, and one which executes this program. In the following we assume, that the usual non-destructive list functions $(cons, fst, rest, reverse, nth, islist)$ are static functions in the evolving algebra and that it is separable. For each run-time rule $\boxed{if\ b\ then\ D}$ in T let $i \in S$ be a new instruction and add the following rules to T_e and T_c:

compilation: $\boxed{if\ b\ then\ D^C \cup \{prg := cons(cons(i, args), prg)\}} \in T_c$

execution: $\boxed{if\ islist(prg) \& fst(fst(prg)) = i\ then\ \tilde{D}^R} \in T_e$

where D^C is the set of compile-time rules in D, D^R is the set of run-time rules in D and $args = [a_1, ..., a_m]$ is the list of all maximal subterms occurring in D^R, which only consist of compile-time functions. $\tilde{D}^R$ is obtained from D^R by replacing every occurrence of a_i by $nth(i+1, fst(prg))$. Furthermore the $islist$ function yields true, if its argument is a non-empty list. Finally we have $\boxed{if\ islist(prg)\ then\ prg := rest(prg)} \in$ T_e and all compile-time rules are elements of T_c. Obviously splitting the rule set T can be done in time $O(|T|)$. Now we define the following evolving algebras [2] : $\Psi_c =< \sigma \cup \{prg\}, S, T_c, \mathcal{I}_0^c >$ where $\mathcal{I}_0^c|_\sigma = \mathcal{I}_0$ and $\mathcal{I}_0^c(prg) = nil$ and $\Psi_{\mathcal{I}_m^c} =< \sigma \cup \{prg\}, S, T_e, \mathcal{I}_0^e >$ where $\mathcal{I}_0^e|_\sigma = \mathcal{I}_m^c|_\sigma$ and $\mathcal{I}_0^e(prg)() = \mathcal{I}_m^c(reverse)(\mathcal{I}_m^c(prg)())$. We call the algebra executing the program $\Psi_{\mathcal{I}_m^c}$ to make explicit, that it depends on the terminal state of the compiling algebra. Taking the time complexities of all phases of the pass separation into account, the transformation needs time $O(max(|\sigma|, |T|))$

Theorem: If $\mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_m$ is a computation in Ψ, then in the compiling algebra Ψ_c there exists a computation $\mathcal{I}_0^c \xrightarrow{\Psi_c} \mathcal{I}_m^c$ and in the executing algebra $\Psi_{\mathcal{I}_m^c}$ there exists a computation $\mathcal{I}_0^e \xrightarrow{\Psi_{\mathcal{I}_m^c}} \mathcal{I}_q^e$, where $q \leq m$. Furthermore we have $\mathcal{I}_q^e|_\sigma = \mathcal{I}_m$.

Proof: First we note, that no dynamic compile-time functions occur in T_e. Let C be the set of compile-time rules in T and R be the set of run-time rules. We will prove the five stronger properties

Lemma: The following properties hold: (1) $\forall j \in \{0, ..., m\} : \mathcal{I}_j^c|_R = \mathcal{I}_0|_R$ and (2) $\forall j \in \{0, ..., m\} : \mathcal{I}_j^c|_C = \mathcal{I}_j|_C$ and (3) $\forall j \in \{0, ..., q\} : \mathcal{I}_j^e|_C = \mathcal{I}_m|_C$ and (4) $\exists i_0, ..., i_q \leq m, i_k < i_{k+1} : \forall j \in \{0, ..., q\} : \mathcal{I}_j^e|_R = \mathcal{I}_{i_j}|_R$ and (5) $\mathcal{I}_q^e|_R = \mathcal{I}_m|_R$

(1): Clearly $\mathcal{I}_j^c|_R = \mathcal{I}_0|_R$, because there is no update to a run-time function in any of the rules in T_c.

(2): This part follows by induction on the steps of the computations:

$j = 0$: by definition we have: $\mathcal{I}_0^c|_\sigma = \mathcal{I}_0$ and as a consequence $\mathcal{I}_0^c|_C = \mathcal{I}_0|_C$

$j + 1$: In T_c are only updates to compile-time functions, because any rule containing an update to a run-time function is considered a run-time rule. As a consequence, for all updates u to compile-time functions we have $u \in updates(T, \mathcal{I}_j) \Leftrightarrow u \in updates(T_c, \mathcal{I}_j^c)$, because the conditions,which have to be true for adding u to $updates(T_c, \mathcal{I}_j^c)$ contain only compile-time functions, for which we know, that $\mathcal{I}_j|_C = \mathcal{I}_j^c|_C$ by the induction hypothesis. By the definition of a computation step it follows, that $\mathcal{I}_{j+1}|_C = \mathcal{I}_{j+1}^c|_C$.

(*): Furthermore we know, that only the guard of one run-time rule can be true (mutually exclusive rules). In this case prg is updated:
$\mathcal{I}_{j+1}^c(prg)() = \mathcal{I}_j^c(cons)([i, eval(a_1, \mathcal{I}_j^c), ..., eval(a_n, \mathcal{I}_j^c)], \mathcal{I}_j^c(prg))$.
By the induction hypothesis it follows, that $eval(a_k, \mathcal{I}_j^c) = eval(a_k, \mathcal{I}_j)$

(3): Since T_e does not contain an update to a compile-time function, we have $\mathcal{I}_j^e|_C = \mathcal{I}_m^c|_C$ and by (1) we have $\mathcal{I}_m^c|_C = \mathcal{I}_m|_C$.

(4): This part follows by induction on the steps of the computations:

$j = 0$: By definition we have: $\mathcal{I}_0^e|_\sigma = \mathcal{I}_m^c|_\sigma$ and by (1) $\mathcal{I}_m^c|_R = \mathcal{I}_0|_R$. Thus it follows, that $\mathcal{I}_0^e|_R = \mathcal{I}_0|_R$ and $i_0 = 0$. $j + 1$:

case 1: There is a computation step $\mathcal{I}_{i_{j+1}-1} \xrightarrow{\Psi} \mathcal{I}_{i_{j+1}}$, where $i_j < i_{j+1}$ and a top-level condition of a run-time rule evaluates to $true$. Then this guard also evaluates to $true$ in the step $\mathcal{I}_{i_{j+1}-1}^c \xrightarrow{\Psi_c} \mathcal{I}_{i_{j+1}}^c$ and $[i, \tilde{a}_1, ..., \tilde{a}_k]$ is cons'ed to prg. The rules involved are: $\boxed{if\ b\ then\ D} \in T$, $\boxed{if\ b\ then\ D^C \cup \{prg := cons(cons(i, args), prg)\}} \in T_c$ and $\boxed{if\ islist(prg) \& fst(fst(prg)) = i\ then\ \tilde{D}^R} \in T_e$

Since in $\Psi_{\mathcal{I}_m^c}$ the value of prg has been reversed and at each step $prg := rest(prg)$

[2] The restriction of a function f to a set A is defined as $f|_A = \{(a, f(a)) : a \in A\}$.

is executed, it is easy to see, that $[i, \tilde{a_1}, ..., \tilde{a_k}]$ is the first element of *prg* in $\mathcal{I}^e_j$. As a consequence we have: $updates(T_e, \mathcal{I}^e_j) = updates(\tilde{D}^R, \mathcal{I}^e_j) \cup \{eval(prg := rest(prg))\}$ Since there is no other update to a run-time function in an intermediate step, we have $\mathcal{I}_{i_j}|_R = \mathcal{I}_{i_{j+1}-1}|_R$ and by the induction hypothesis, $\mathcal{I}^e_j|_R = I_{i_j}|_R$. Now it follows, that $updates(\tilde{D}^R, I^e_j) = updates(\tilde{D}^R, \mathcal{I}_{i_{j+1}-1} \cup \mathcal{I}^e_j|_{\{prg\}})$ and by (*) we know that $\tilde{a_k} = eval(a_k, I^c_{i_{j+1}-1}) = eval(a_k, I_{i_{j+1}-1})$ and thus $updates(\tilde{D}^R, \mathcal{I}_{i_{j+1}-1} \cup \mathcal{I}^e_j|_{\{prg\}}) = updates(D, \mathcal{I}_{i_{j+1}-1})$. And by the definition of a computation step: $\mathcal{I}^e_{j+1}|_R = \mathcal{I}_{i_{j+1}}|_R$.
case 2: There is no such computation step. Then $j = q$ and we conclude, that $\mathcal{I}_q|_R = \mathcal{I}_m|_R$ and by the induction hypothesis $\mathcal{I}^e_q|_R = \mathcal{I}_q|_R$ and thus $\mathcal{I}^e_q|_R = \mathcal{I}_m|_R$, which is point (5) of the above lemma. □

An Example Next we will apply pass separation to an interpreter for simple arithmetic expressions ($E \rightarrow VAR \mid INT \mid (E\ OP\ E)$). We assume, that `in` is a list of symbols representing an expression, e.g. `in` $= [$"(", `X`, $+$, "(", `7`, $*$, `3`, ")", ")"$]$. Furthermore **`env`** maps variable names to values, e.g. `env(X)` $=$ `3`.

```
if islist(in) then
 { if fst(in)="(" then in:=rest(in),
   if isop(fst(in)) { opstack:=cons(fst(in),opstack), in:=rest(in) },
   if isint(fst(in)) then { estack:=cons(fst(in),estack), in:=rest(in) },
   if isvar(fst(in)) then { estack:=cons(env(fst(in)),estack), in:=rest(in) },
   if fst(in)=")" then { opstack:=rest(opstack),
                         estack:=cons(apply(fst(opstack),snd(estack),
                                                fst(estack)),
                                      rest(rest(estack))),
                         in:=rest(in) } }
```

Using flattening the above transition rule can be converted into a set of transition rules, which is more suitable for applying the pass separation transformation:

```
if islist(in) then in:=rest(in),
if islist(in) & isop(fst(in)) then opstack:=cons(fst(in),opstack),
if islist(in) & isint(fst(in)) then estack:=cons(fst(in),estack),
if islist(in) & isvar(fst(in)) then estack:=cons(env(fst(in)),estack),
if islist(in) & (fst(in)=")") then  opstack:=rest(opstack),
if islist(in) & (fst(in)=")" then estack:=cons(apply(fst(opstack),snd(estack),
                                                       fst(estack)),
                                                 rest(rest(estack)))
```

We assume, that **`in`** is known at compile-time and **`env`** not before run-time and classify functions and rules as described above. Now we can apply the pass separation transformation[3] to generate a simple compiler

```
if islist(in) then
 { in:=rest(in),
   if isop(fst(in)) then opstack:=cons(fst(in),opstack),
   if isint(fst(in)) then prg:=cons(cons("pushint",fst(in)),prg),
   if isvar(fst(in)) then prg:=cons(cons("pushvar",fst(in)),prg),
   if fst(in)=")" then opstack:=rest(opstack),
   if fst(in)=")" then prg:=cons(cons("app",fst(opstack)),prg) }
```

[3] To increase readability we applied the "crushing" transformation, see the conditions `islist(in)` and `islist(prg)`.

and an abstract target machine

```
 if islist(prg) then
  { if fst(fst(prg))="pushint" then estack:=cons(rest(fst(prg)),estack),
    if fst(fst(prg))="pushvar" then estack:=cons(env(rest(fst(prg))),estack),
    if fst(fst(prg))="app" then estack:=cons(apply(rest(fst(prg)),
                                                   snd(estack),fst(estack)),
                                             rest(rest(estack))),
    prg := rest(prg) }
```

For example given the value `in` = ["(", X, +, "(", 7, ∗, 3, ")", ")"] at compile time, the compiler will generate the abstract machine program: **prg** =[**(pushvar X)**, **(pushint 7)**, **(pushint 3)**, **(app ∗)**, **(app +)**]. The above example shows, that pass separation can be used for semantics-directed compiler generation.

Implementation All transformations in this paper can be automated, but testing the mutual exclusion of run-time rules is not even decidable. Nevertheless heuristics can be used to decide, whether the conditions are mutually exclusive. Even checking mutual exclusion at run-time is co-NP complete ([Gur91]).

3 Other Work

In [JS86] the authors use pass separation to generate a compiler and an abstract machine for a functional language from a specification of an abstract interpreter. The transformations are very sophisticated, but they are neither formally defined, nor is it likely that they can be automated. In [Han91] John Hannan defines a pass separation transformation of a very restricted class of term rewriting systems. From an interpreter for a simple functional language, which he calls the CLS machine, he derives a compiler and an abstract machine similar to the CAM ([CCM85]).

4 Conclusions

We defined evolving algebras in automata theoretic terms and used this definition as a basis to define some transformations on evolving algebras and prove some essential properties of these. The pass separation transformation can be used to split simple interpreters into compilers and abstract machines.

References

[BR92] Egon Börger and Dean Rosenzweig. The WAM – Definition and Compiler Correctness. Technical Report TR-14/92, Universita Degli Studi Di Pisa, Pisa, Italy, 1992.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorial Abstract Machine. In **Proceedings of FPCA'85**. Springer, LNCS 201, 1985.

[Gur91] Yuri Gurevich. Evolving Algebras: a tutorial introduction. **Bulletin of the European Association for Theoretical Computer Science**, 43:264–284, 1991.

[Han91] J. Hannan. Staging Transformations for Abstract Machines. In **Partial Evaluation and Semantics-Based Program Manipulation**. SigPlan Notices, vol. 26(9), 1991.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. **Partial Evaluation and Automatic Program Generation**. Englewood Cliffs, NJ: Prentice Hall, 1993.

[JS86] U. Jørring and W.L. Scherlis. Compilers and Staging Transformations. In **13th ACM Symposium on Principles of Programming Languages**, 1986.

This article was processed using the LaTeX macro package with LLNCS style