# Specification Languages in Algebraic Compilers

Eric Van Wyk[1]

*Oxford University Computing Laboratory*
*Wolfson Building, Parks Road*
*Oxford OX1 3QD, UK*

---

**Abstract**

Algebraic compilers provide a powerful and convenient mechanism for specifying language translators. With each source language operation one associates a computation for constructing its target language image; these associated computations, called *derived operations*, are expressed in terms of operations from the target language. Sometimes the target language is not powerful enough to specify the required translation and one may then need to extend the target language algebras with more computationally expressive operations or elements. A better solution is to package these extensions in a *specification language* which can be composed with the target language to ensure that all operations and elements needed or desired for performing the translation are provided. In the example in this paper, we show how imperative and functional specification languages can be composed with a target language to implement a temporal logic model checker as an algebraic compiler and show how specification languages can be seen as components to be combined with a source and target language to generate an algebraic compiler.

*Key words:* Algebraic compiler, specification languages.

---

## 1 Introduction

Language processing tools like attribute grammars [1] and algebraic compilers [2–4] provide powerful and convenient mechanisms for specifying language translators. In both, one associates with each operation of the source language specifications for computations that construct the target language images of

---

[1] Present Address: Department of Computer Science and Engineering, University of Minnesota, 200 SE Union St, Minneapolis, MN 55455, USA

source constructs created by the operation. The complexity of these computations contributes to the complexity of the entire language translator specification. We are interested in means of reducing the specification's complexity by writing these computations in languages appropriate to the translation task at hand. These languages must be computationally expressive enough to specify the necessary computations, and should provide convenient programming constructs which simplify the specification process for the translator implementer. A *specification language* provides additional constructs which are used, along with those from the target language, to specify the translation computation associated with each source language operation. They are essential when the target language is not expressive enough to specify the translation, but also helpful in simplifying the specification by providing more abstract operations. Since algebraic compilers provide a solid mathematical framework and give a clear distinction between the target language and the language used to specify the translation, they provide a better context in which to explore the issues of specification languages.

In this paper we will primarily follow Rus's model [2] of algebraic compilers in which an algebraic compiler $\mathcal{C} : L_S \to L_T$ is a *language–to–language* translator that uses an algorithm for homomorphism computation to embed (the algebras of) a source language $L_S$ into (the algebras of) a target language $L_T$. The computations associated with each source language operation that define an algebraic compiler are specified as terms in the target language syntax algebra and are called *derived operations*. In some cases, the operations and elements provided by the target language algebra are not expressive enough to correctly specify the translation or exist at such a low level of abstraction, with respect to the source language, that the specification is excessively difficult to read and write. There are two types of target algebra deficiencies we will address using specification languages in this paper. The first occurs when, although every element in the source algebra can be mapped to an element of the target algebra, the target algebra operations are not expressive enough to implement the mapping. The specification languages used in the model checking example in this paper address this type of deficiency by providing more computationally expressive operations. The second and less frequent type occurs when there are source algebra elements which can not be expressed in the target language. If we do intend to translate such elements then we have no choice but to extend the target language. However, there are cases in which we only intend to translate elements of the source algebra which do have representations in the target but these source elements contain components (subexpressions) which do not have target algebra representations. Thus, although the source elements of interest can be translated to the original target algebra, the translator can not be implemented as a (generalized) homomorphism. In this case our translator is a partial mapping. Specification languages can be of assistance in this case as well as we will see in the comparison to Mosses [4,5] work in Section 6.1. In both of these cases, the target language algebras are combined with specification language algebras which provide the additional operations or elements to make the translation possible or more easily specifiable. In this paper, we explore how different specification languages can be used in conjunction with the target language to correctly and conveniently specify translators implemented as algebraic compilers without extending the target language.

As an example, we develop a model checker for the temporal logic CTL (computation tree logic) [6] as an algebraic compiler which maps the source language CTL into a target language of satisfiability sets. Since the operations in the target language of sets are not powerful enough to specify general computations, we must use a specification language to provide a more computationally expressive language in which to specify this translation. We show how both functional and imperative style specification languages can be used in the specification, thus giving the language implementer some choice in choosing an appropriate specification language.

Our choice of model checking as an example is not as esoteric as it may appear. Model checking has been used to perform data flow analysis on program control and data flow graphs [7] and to find optimization and parallelization opportunities in program dependency and flow graphs [8,9]. In both cases, temporal logic acts as a specification language for certain patterns in a graph representation of the program which are found by a model checker. Thus, temporal logic does have applications in language processing tools and can be seen as a domain specific specification language (see Section 5.3) in algebraic compilers and attribute grammars. An example is provided to illustrate CTL and model checking as a program analysis tool. Since different analyses can require unique temporal logics it is advantageous to be able to generate model checkers for these different logics from their algebraic specifications [10].

Section 2 describes CTL and model checking. In Section 3 we define algebraic languages and compilers and show how CTL and models can be specified as algebraic languages. Section 4 discusses specification languages in algebraic compilers, specifically the specification languages used to implement a model checker as an algebraic compiler. Section 5 provides the specification of the model checker as an algebraic compiler using both a functional and an imperative specification language and provides some discussion of an algebraic programming environment supporting the development of such translators. Section 6 contains a discussion of related work including different models of algebraic compilers, the use of specification languages in attribute grammars, action semantics and rewriting logics. It also includes a discussion of domain specific language specification techniques related to what we present in this paper. Section 7 contains the concluding remarks.

## 2 Model Checking

Model checking [11] is a formal technique used to verify the correctness of a system according to a given correctness specification. Systems are represented as labeled finite state transition systems called *Kripke models* [12] or simply *models*. Correctness properties are defined by formulas written in a temporal logic. In this paper, we use CTL, a propositional, branching-time temporal logic as our example. A model checking algorithm determines which states in a model satisfy a given temporal logic formula; this algorithm can be seen as a language translator which maps formulas in the temporal logic language to sets in a language defined by the model. Note that this is the "classical" view of model checking. There are other model checking techniques for verifying the correctness of a system, such as the CSP refinement technique of Roscoe [13]. We present the problem of model checking a temporal logic as a language translation problem and implement two solutions as algebraic compilers using different specification languages.

Following Clarke et al. [6], we define a model as a tuple $M = \langle N, E, P\colon AP \to 2^N \rangle$, where $N$ is a finite set of nodes $N = \{n_1, n_2, \ldots, n_m\}$, and $E$ defines directed edges between nodes as a binary relation on $N$, $E \subseteq N \times N$, such that $\forall n \in N, \exists n' \in N, (n, n') \in E$, that is, every state has a successor. For each $n \in N$ we use the notation $succ(n) = \{n' \in N | (n, n') \in E\}$. A *path* is an infinite sequence of nodes $(n_0, n_1, n_2, \ldots)$ such that $\forall i, i \geq 0, (n_i, n_{i+1}) \in E$. $AP$ is a finite set of *atomic propositions*, $AP = \{p_1, p_2, \ldots, p_n\}$, $P$ is a proposition labeling function that maps an *atomic proposition* in $AP$ to the set of nodes in $N$ on which that proposition is *true*.

Figure 1 contains a sample program and its control flow graph which is represented as a model. Nodes correspond to program statements and are numbered to match the statement numbers. Additional *entry* and *exit* nodes are also given and numbered 0 and 9 respectively. The edges in the model represent the possible transitions through the program. The atomic propositions which label nodes in this model are $\{entry, exit, def_a, def_b, def_c, use_a, use_b, use_c\}$. The proposition *entry* labels only the entry node; *exit* labels only the exit node, $def_x$ labels a node if it defines the program variable $x$ and $use_x$ labels a node if it uses $x$. We will see how some program analyses, like dead code detection, can be performed by model checking a temporal logic formula on this model.

The following rules [6] define the set of well-formed CTL formulas:

(1) The logical constants, *true* and *false* are CTL formulas.
(2) Every atomic proposition, $p \in AP$, is a CTL formula.
(3) If $f_1$ and $f_2$ are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$ and $f_1 \vee f_2$.
(4) If $f_1$ and $f_2$ are CTL formulas, then so are $ax f_1$, $ex f_1$, $a[f_1\ u\ f_2]$, and

```
1. read $a$
2. $b := a * a$
3. $c := a + 10$
4. if $a < 100$ then
5.    $a := 100 - a$
   else
6.    $c := 10$
7.    $a := c * 2$
   endif
8. write $a$
```

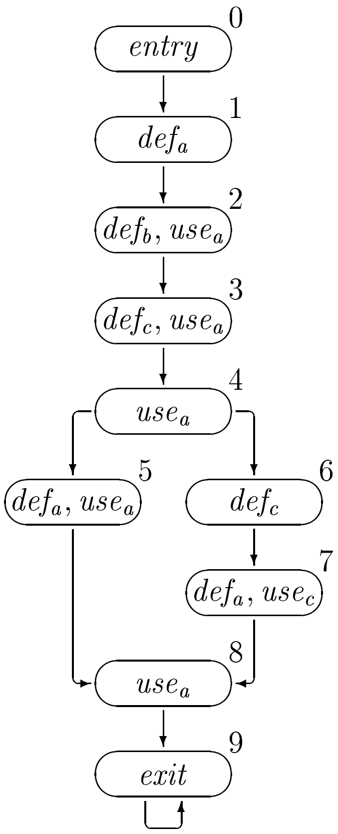

Fig. 1. Example program and control flow model.

$e[f_1 \ u \ f_2]$.

The meaning of a CTL formula is defined by the satisfaction relation, $\models$, presented in Table 1 [6]. By $M, n \models f$, or $n \models f$ where $M$ is implicit, we denote that $n$ *satisfies* the formula $f$ in model $M$. The *satisfiability set* of $f$ in $M$ is defined as $\{n \in N \mid M, n \models f\}$. The *non-temporal* operators defined in rules (1.), (2.) and (3.) have the expected meaning: *e.g.* the formula *true* holds on any node in the model and $\neg f$ holds on a node if $f$ does not hold on that node. The satisfaction of the temporal operators in (4.) depends on more than one node in the model. The formula $ex f$, respectively $ax f$, holds on a node if at least one, respectively all, of its successors satisfies $f$. The formula $e[f_1 \ u \ f_2]$, respectively $a[f_1 \ u \ f_2]$, holds on a node $n$ if on at least one of the, respectively all, paths from this node eventually $f_2$ holds on a node and $f_1$ holds on all intervening nodes.

As an example consider dead code elimination, a program transformation that removes assignment statements which do not affect the outcome of the program. We can use model checking to find such statements. Statement (2.) $b := a * a$ can be removed from the example program with out changing the meaning of the program since the variable $b$ is not used again. We can encode this in the CTL formula $a[\neg use_b \ u \ exit]$. We can safely remove statement (2.)

| | | |
|---|---|---|
| $n \models p$ | iff | $n \in P(p), p \in AP$ |
| $n \models true$ | iff | true |
| $n \models false$ | iff | false |
| $n \models \neg f$ | iff | not $n \models f$ |
| $n \models f_1 \wedge f_2$ | iff | $n \models f_1$ and $n \models f_2$ |
| $n \models f_1 \vee f_2$ | iff | $n \models f_1$ or $n \models f_2$ |
| $n \models ax f_1$ | iff | $\forall (n, n') \in E, n' \models f_1$ |
| $n \models ex f_1$ | iff | $\exists (n, n') \in E, n' \models f_1$ |
| $n \models a[f_1 \ u \ f_2]$ | iff | $\forall$ *paths* $(n = n_0, n_1, n_2, \ldots),$ $\exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow n_j \models f_1]]$ |
| $n \models e[f_1 \ u \ f_2]$ | iff | $\exists$ *a path* $(n = n_0, n_1, n_2, \ldots),$ $\exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow n_j \models f_1]]$ |

Table 1
The CTL satisfiability relation.

since $2 \models a[\neg use_b \ u \ exit]$. Note that although statement (3.) could be removed without affecting the output of the program, $3 \not\models a[\neg use_c \ u \ exit]$ since $c$ is used again, but only after it is redefined. We could refine our CTL formula to $a[\neg use_c \ u \ (def_c \wedge \neg use_c) \vee exit]$. This formula states that on all paths $c$ is not used until either the *exit* node is reached or a node which defines $c$ but does not use $c$ is reached. Since $3 \models a[\neg use_c \ u \ (def_c \wedge \neg use_c) \vee exit]$ we could therefore remove that statement.

We present both a functional and an imperative version of a CTL model checker implemented as an algebraic compiler [2] $MC : L_S \rightarrow L_T$ where the source language $L_S$ is CTL and the target language $L_T$ is a language describing the satisfiability sets of nodes of the model $M$. The algebraic compiler $MC$ translates a CTL formula $f$, to the set of nodes, $N'$, on which the formula $f$ holds. That is, $MC(f) = N'$ where $N' = \{n \in N | M, n \models f\}$.

## 3 Algebraic compilers

### *3.1 Σ–algebras and Σ–languages*

An *operator scheme* is a tuple $\Sigma = \langle S, Op, \sigma \rangle$ where $S$ is a set of sorts, $Op$ is a set of operator names, and $\sigma$ is a mapping defining the signatures of the operator names in $Op$ over the sorts in $S$. That is, $\sigma \colon Op \rightarrow S^* \times S$ such that if, for example, $s_0$, $s_1$, and $s_2$ are sorts in $S$ and $op$ is an operator name in $Op$

which stands for operations which take an element of sort $s_1$ and an element of sort $s_2$ and generates an element of sort $s_0$, then $\sigma(op) = s_1 \times s_2 \to s_0$.

A $\Sigma$–algebra is a family of non–empty sets, called the *carrier sets*, indexed by the sorts $S$ of $\Sigma$ and a set of $Op$ named operations over the elements of these sets whose signatures are given by $\sigma$. There may be many different algebras for the same operator scheme $\Sigma$. These algebras are called *similar* and are members of the same *class of similarity*, denoted $\mathcal{C}(\Sigma)$. An interesting member of $\mathcal{C}(\Sigma)$ is the *word* or *term* algebra for $\Sigma$. This algebra is parameterized by a family of variables $V = \{V_s\}_{s \in S}$ and is often denoted $W_\Sigma(V)$. Its carrier sets contain words formed from the variables of $V$ and operator names of $Op$ and its operators construct well formed formulas called words according to the operation signatures defined by $\sigma$ [14]. Variables in $V$ and the nullary operators are called *generators* and they are thus said to *generate* $W_\Sigma(V)$.

A $\Sigma$–language [2] $L$ is defined as the tuple $\langle \mathcal{A}^{sem}, \mathcal{A}^{syn}, \mathcal{L} \colon \mathcal{A}^{sem} \to \mathcal{A}^{syn} \rangle$ where $\mathcal{A}^{sem}$ is a $\Sigma$-algebra which is the language semantics, $\mathcal{A}^{syn}$ is a $\Sigma$ word algebra which is the language syntax, and $\mathcal{L}$ is a partial mapping called the *language learning function* [2,15]. The mapping $\mathcal{L}$ maps semantic constructs in $\mathcal{A}^{sem}$ to their expressions as syntactic constructs in $\mathcal{A}^{syn}$ such that there exists a complementary homomorphism $\mathcal{E} \colon \mathcal{A}^{syn} \to \mathcal{A}^{sem}$ where if $\alpha \in \mathcal{A}^{sem}$ and $\mathcal{L}(\alpha)$ is defined then $\mathcal{E}(\mathcal{L}(\alpha)) = \alpha$. The mapping $\mathcal{E}$ is called the *language evaluation function* and maps expressions in $\mathcal{A}^{syn}$ to their semantic constructs in $\mathcal{A}^{sem}$. $\mathcal{L}$ may be a relation instead of a function, but $\mathcal{E}$ is always a function since semantic constructs in $\mathcal{A}^{sem}$ may be expressed in many ways in $\mathcal{A}^{syn}$, but syntactic constructs in $\mathcal{A}^{syn}$ have exactly one meaning in $\mathcal{A}^{sem}$. In what follows we will define both CTL and the model $M$ to be checked as $\Sigma$-languages.

### *3.1.1 CTL as a $\Sigma$–language.*

CTL can be specified as the $\Sigma$–language $L_{ctl} = \langle \mathcal{A}^{sem}_{ctl}, \mathcal{A}^{syn}_{ctl}, \mathcal{L}_{ctl} \rangle$ [16] using the operator scheme $\Sigma_{ctl} = \langle S_{ctl}, Op_{ctl}, \sigma_{ctl} \rangle$ where $S_{ctl} = \{F\}$, the set of sorts containing only one sort for "formula", $Op_{ctl} = \{true, false, \neg, \wedge, \vee, ax, ex, au, eu\}$, and $\sigma_{ctl}$ is defined in Table 2. As CTL formulas are written using atomic propositions from a specific model $M$, the syntax algebra $\mathcal{A}^{syn}_{ctl}$ is

$$\begin{array}{llrcl@{\qquad}llrcl}
\sigma_{ctl}(true) & = & \emptyset & \to & F & \sigma_{ctl}(ax) & = & F & \to & F \\
\sigma_{ctl}(false) & = & \emptyset & \to & F & \sigma_{ctl}(ex) & = & F & \to & F \\
\sigma_{ctl}(\neg) & = & F & \to & F & \sigma_{ctl}(au) & = & F \times F & \to & F \\
\sigma_{ctl}(\wedge) & = & F \times F & \to & F & \sigma_{ctl}(eu) & = & F \times F & \to & F \\
\sigma_{ctl}(\vee) & = & F \times F & \to & F & & & & &
\end{array}$$

Table 2

The signature $\sigma_{ctl}$ of $Op_{ctl}$.

parameterized by the set of atomic propositions $AP$ from $M$ and is denoted as $\mathcal{A}^{syn}_{ctl}(AP)$. For example, the formula $a[\neg use_b\ u\ exit]$ shown above has variables $use_b$ and $exit$ from $AP$ of the above model and the $\neg$ and $au$ operations construct the CTL formula (in the syntax word algebra) from these variables. The algebra $\mathcal{A}^{syn}_{ctl}(AP)$ has as its carrier set all possible CTL formulas written using the atomic propositions in $AP$. The operations of this algebra construct formulas (words) from variables and operator names. The set of variables $AP$ generates the algebra $\mathcal{A}^{syn}_{ctl}(AP)$.

Just as the syntactic algebra $\mathcal{A}^{syn}_{ctl}(AP)$ is parameterized by the atomic propositions $AP$ of the model $M$, the semantic algebra $\mathcal{A}^{sem}_{ctl}$ is also parameterized by $M$ in that the carrier set of the semantic algebra $\mathcal{A}^{sem}_{ctl}$ is the power set of the set of nodes of the model $M$. The operations in this algebra, while *similar* (that is, having the same signature) to those in $\mathcal{A}^{syn}_{ctl}$, operate on sets, not formulas, since the meaning of a CTL formula is in fact its satisfiability set. Although the operations in the word algebra $\mathcal{A}^{syn}_{ctl}(AP)$ are easily defined as simply concatenating operation names and operands together, the operations in the semantic algebra $\mathcal{A}^{sem}_{ctl}$ are less easily defined. The operation names $\{true, false, \neg, \wedge, \vee, ax, ex, au, eu\}$ in $Op_{ctl}$ are interpreted in $\mathcal{A}^{sem}_{ctl}$ by the respective operations $\{N, \emptyset, \mathcal{C}, \cap, \cup, next_{all}, next_{some}, lfp_{all}, lfp_{some}\}$ where

- The nullary operators $N$ and $\emptyset$ are, respectively, the constant set of all nodes in $M$ and the constant empty set.
- The unary operator $\mathcal{C}$ produces the complement in the set $N$ of its argument.
- The binary operators $\cap$ and $\cup$ are the standard set union and intersection.
- The unary operators $next_{all}$ and $next_{some}$ are defined as
  - · $next_{all}(\alpha) = \{n \in N | successors(n) \subseteq \alpha\}$, $\alpha \in 2^N$
  - · $next_{some}(\alpha) = \{n \in N | successors(n) \cap \alpha \neq \emptyset\}$, $\alpha \in 2^N$

  Here $successors(n)$ denotes the successors in $M$ of node $n$.
- The binary operators $lfp_{all}$ and $lfp_{some}$ are defined, for $\alpha, \beta \in 2^N$, as
  - · $lfp_{all}(\alpha, \beta)$ computes the least fixed point of the equation $Z = \beta \cup (\alpha \cap \{n \in N | successors(n) \subseteq (\alpha \cap Z)\})$
  - · $lfp_{some}(\alpha, \beta)$ computes the least fixed point of the equation $Z = \beta \cup (\alpha \cap \{n \in N | (successors(n) \cap \alpha \cap Z) \neq \emptyset\})$ [6].

Although we do have a mathematical formulation of the semantic algebra $\mathcal{A}^{sem}_{ctl}$, language learning function $\mathcal{L}_{ctl}$ and the language evaluation function $\mathcal{E}_{ctl}$ they are not used in constructing the model checking software artifact which performs the actual model checking process. In Sections 5.1 and 5.2 we will define model checkers as compositions of several functions and relations including the semantic algebra and language learning and evaluation functions of other $\Sigma$-languages. These will be discussed as we encounter them. Even though some components of various $\Sigma$-languages will not be explicitly used they are all defined.

### 3.1.2 A model as a $\Sigma$–language.

As the target language of our algebraic model checker, we develop a $\Sigma$–language based on sets which is parameterized by a specific model. For a model $M$, $L_M = \langle \mathcal{A}_M^{sem}, \mathcal{A}_M^{syn}, \mathcal{L}_M \rangle$ using operator scheme $\Sigma_M = \langle S_M, Op_M, \sigma_M \rangle$ where $S_M = \{Set, Node, Boole\}$, $Op_M = \{\emptyset, N, \cup, \cap, \backslash,$ $succ$, $\in, \subseteq, =,$ "$\{\_\}$", $insert$, $get\_one$, $get\_rest$ $\}$, and $\sigma_M$ is defined in Table 3. The operators in the syntax and semantics algebras of $L_M$ are mostly self–descriptive. The nullary operators $\emptyset$ and $N$ generate respectively the empty set and the full set of nodes $N$. The binary operators $\cup$, $\cap$, and $\backslash$ are respectively set union, intersection and difference. We also have the subset ($\subseteq$), set equality ($=$), and membership operations ($\in$), the successor function $succ$ and singleton set creation function denoted by $\{\_\}$. The $get\_one$ operation returns a node from a non–empty set, $get\_rest$ returns all but one node from a non–empty set and $insert$ adds an element to an existing set. They are defined such that for any non–empty set $S$, $insert(get\_one(S), get\_rest(S)) = S$. These operators build set expressions in the syntax algebra $\mathcal{A}_M^{syn}$ and sets in the semantic algebra $\mathcal{A}_M^{sem}$.

$$\begin{aligned}
\sigma_M(\emptyset) &= & \emptyset &\to Set \\
\sigma_M(N) &= & \emptyset &\to Set \\
\sigma_M(\cup) &= & Set \times Set &\to Set \\
\sigma_M(\cap) &= & Set \times Set &\to Set \\
\sigma_M(\backslash) &= & Set \times Set &\to Set \\
\sigma_M(succ) &= & Node &\to Set \\
\sigma_M(\{\_\}) &= & Node &\to Set \\
\sigma_M(\in) &= & Node \times Set &\to Boole \\
\sigma_M(\subseteq) &= & Set \times Set &\to Boole \\
\sigma_M(=) &= & Set \times Set &\to Boole \\
\sigma_M(insert) &= & Node \times Set &\to Set \\
\sigma_M(get\_one) &= & Set &\to Node \\
\sigma_M(get\_rest) &= & Set &\to Set
\end{aligned}$$

Table 3
The signature $\sigma_M$ of $Op_M$.

This language learning function $\mathcal{L}_M$ is a relation that maps set values $s$ to set expressions and $\mathcal{E}_M$ evaluates these set expressions to generate sets. They are defined such that $\forall s \in \mathcal{A}_M^{sem}, \mathcal{E}_M(\mathcal{L}_M(s)) = s$. In our model checkers in Sections 5.1 and 5.2, $\mathcal{L}_M$ is used as the final step to map set values to set expressions. It in effect acts as a "print" function for the algebra. Although it is a relation, we will apply it as a function under the assumption that it will generate the most compact set expression for a given set by simply listing the set elements and not forming complex expressions. It is defined as expected.

### *3.2 Algebraic compilers*

An *algebraic compiler* [2,15] $\mathcal{C}\colon L_S \rightarrow L_T$ that maps the language $L_S = \langle \mathcal{A}_S^{syn}, \mathcal{A}_S^{sem}, \mathcal{L}_S \rangle$ into the language $L_T = \langle \mathcal{A}_T^{syn}, \mathcal{A}_T^{sem}, \mathcal{L}_T \rangle$ is a pair of (generalized) homomorphisms $(H_{syn}\colon \mathcal{A}_S^{syn} \rightarrow \mathcal{A}_T^{syn}, H_{sem}\colon \mathcal{A}_S^{sem} \rightarrow \mathcal{A}_T^{sem})$ defined such that the diagram in Figure 2 commutes. In general, the operator schemes of the algebras in these two languages may not be similar, as is the case with the operator schemes $\Sigma_{ctl}$ and $\Sigma_M$ for the languages $L_{ctl}$ and $L_M$. Thus there may not be a homomorphism between the algebras of the source and target languages. Instead, for each source algebra operation we will compose an appropriate operation from several target algebra operations. Such operations are called *derived operations*. Derived operations are written using words from the target word algebra parameterized by a set of *specification*[2] *variables*. We will use sub-scripted versions of the sort names from the source language operator scheme as specification variables. The word "$N \setminus F_1$", is a word in the algebra $\mathcal{A}_M^{syn}(\{F_1\})$ which specifies the unary derived operation for taking the complement of a set with respect to the full set of nodes $N$. The specification variable $F_1$ is the *formal parameter* of the derived operation. We will associate this derived operation with the CTL operation $\neg$ since given the satisfiability set of a formula $f$, it will generate the satisfiability set of the formula $\neg\, f$.

$$\begin{tikzcd}
\mathcal{A}_S^{sem} \arrow[r, "\mathcal{L}_S"] \arrow[d, "H_{sem}"] & \mathcal{A}_S^{syn} \arrow[r, "\mathcal{E}_S"] \arrow[d, "H_{syn}"] & \mathcal{A}_S^{sem} \arrow[d, "H_{sem}"] \\
\mathcal{A}_T^{sem} & \mathcal{A}_T^{syn} \arrow[l, "\mathcal{E}_T"'] & \mathcal{A}_T^{sem} \arrow[l, "\mathcal{L}_T"']
\end{tikzcd}$$

Fig. 2. An algebraic compiler.

To define a generalized homomorphism [17] $H$ from algebra $\mathcal{A}_{\Sigma_S}$ with operator scheme $\Sigma_S = \langle S_S, Op_S, \sigma_S \rangle$ to algebra $\mathcal{A}_{\Sigma_T}$ with the possibly dissimilar operator scheme $\Sigma_T = \langle S_T, Op_T, \sigma_T \rangle$ we must define the following mappings:

(1) a *sort map*, $sm\colon S_S \rightarrow S_T$ which maps source algebra sorts to target algebra sorts. In a generalized homomorphism, an object of sort $a$ of $\Sigma_S$ will be mapped to an object of sort $sm(a)$ of $\Sigma_T$.
(2) an *operator map*, $om\colon Op_S \rightarrow W_{\Sigma_T}(S'_S)$, which maps operators in the source algebra to words in the target syntax algebra with specification variables $S'_S$ – the source sort names with subscripts. These words specify the derived operations used in both the syntax and semantic target

---

[2] In previous work we have referred to these as *meta variables*.

algebras $\mathcal{A}_{ST}^{syn}$ and $\mathcal{A}_{ST}^{sem}$ respectively of the hybrid language $L_{ST}$ defined below.

The derived operations, which take operands from the target algebra, have the same signatures as their counterparts in the source algebra, and thus we implicitly create an intermediate hybrid algebra $L_{ST} = \langle \mathcal{A}_{ST}^{sem}, \mathcal{A}_{ST}^{syn}, \mathcal{L}_{ST} \rangle$ which has the same operator scheme $\Sigma_S$ as the source algebras, but whose carrier sets are populated by values from the target algebras and whose operations are the derived operations specified by the operator map *om*. A generalized homomorphism $H_s\colon \mathcal{A}_S^s \to \mathcal{A}_T^s, s \in \{sem, syn\}$ is thus the composition of an *embedding* homomorphism from $\mathcal{A}_S^s$ to the intermediate algebra $\mathcal{A}_{ST}^s$, $(em_s\colon \mathcal{A}_S^s \to \mathcal{A}_{ST}^s)$ with an identity *injection mapping* from the intermediate algebra to $\mathcal{A}_T^s$, $(im_s\colon \mathcal{A}_{ST}^s \to \mathcal{A}_T^s)$ [2,18]. The mappings $im_s$ are identity mappings that map elements in sort $a, a \in S_S$ in $\mathcal{A}_{ST}^s$ to the same value in sort $sm(a) \in S_T$ in $\mathcal{A}_T^s$. Thus $H_s = em_s \circ im_s$. Note that in this paper the arguments of function composition ($\circ$) are written in diagrammatic order as opposed to following the standard convention. Thus $(f \circ g)(x) = g(f(x))$. In later sections where we compose several functions to define a model checker this ordering makes it easier to "follow the path" through the commutative diagrams. Since both the syntax and semantic generalized homomorphisms of Figure 2 are implemented in this manner, the intermediate algebras form an intermediate $\Sigma$–language $L_{ST}$ and thus, the diagram of Figure 3 becomes the commutative diagram in Figure 3.

$$\begin{tikzcd}
\mathcal{A}_S^{sem} \arrow[r, "\mathcal{L}_S"] \arrow[d, "em_{sem}"] & \mathcal{A}_S^{syn} \arrow[r, "\mathcal{E}_S"] \arrow[d, "em_{syn}"] & \mathcal{A}_S^{sem} \arrow[d, "em_{sem}"] \\
\mathcal{A}_{ST}^{sem} \arrow[d, "im_{sem}"] & \mathcal{A}_{ST}^{syn} \arrow[l, "\mathcal{E}_{ST}"'] \arrow[d, "im_{syn}"] & \mathcal{A}_{ST}^{sem} \arrow[l, "\mathcal{L}_{ST}"'] \arrow[d, "im_{sem}"] \\
\mathcal{A}_T^{sem} \arrow[r, "\mathcal{L}_T"] & \mathcal{A}_T^{syn} \arrow[r, "\mathcal{E}_T"] & \mathcal{A}_T^{sem}
\end{tikzcd}$$

Fig. 3. An algebraic compiler with the intermediate language displayed.

Given a mapping $H' = \{H'_a\colon a \to sm(a)\}_{a \in S_S}$ that maps generators $G = \{G_a\}_{a \in S_S}$ of the source algebra into the target algebra, $H'$ can be uniquely extended to a homomorphism $H\colon \mathcal{A}_S \to \mathcal{A}_T$ [17,2]. The algorithm for implementing a generalized homomorphism $H$ from a $\Sigma_S$ algebra generated by $G$ is

$$\begin{aligned} H(x) = \mathbf{if}\ & x \in G_a \text{ for some } a \in S_S \ \mathbf{then}\ H'_a(x) \\ & \mathbf{else\ if}\ x = f(x_1, x_2, \ldots, x_n) \text{ for some } f \in Op_S \\ & \qquad \mathbf{then}\ om(f)(H(x_1), H(x_2), \ldots, H(x_n)) \end{aligned} \tag{1}$$

This is all made clear by examining it in the context of our model checker as an algebraic compiler. For starters, the sort map $sm$ simply maps the sort $F$ in $\Sigma_{ctl}$ to the sort $Set$ in $\Sigma_M$. The generators $G$ are the set of atomic propositions, $G_F = AP$, and $H'_F$ is the proposition modeling function $P$ from $M$ which maps atomic propositions to their satisfiability sets. What is left then, is to define the operator map $om$ which maps CTL operators in $Op_{ctl}$ to derived operations over satisfiability sets. We saw above how the word "$N \setminus F_1$" could be used to specify the derived operation for the CTL operation $\neg : F_1 \to F_0$. The use of the indexed sort name $F$ ($F_1$) as the specification variable is to show the correspondence between the parameters of the source and derived operations. The subscripts are used to distinguish between multiple parameters of the same sort, different sorts will have different names.

Consider now the CTL operator $ax$. We cannot write a correct derived operation using only the operators from the target language. We need additional constructs with which to compose a derived operation. It is at this point that we can begin to speak of *specification languages*[3] used in the specification of algebraic compilers instead of just *specification variables*. By introducing some functional language constructs into the language in which we write derived operations, we may like to write the derived operation for $ax$ as

$$om(ax\colon F_1 \to F_0) = \text{filter}\ (\ (\lambda\ \text{n}\ .\ succ(\text{n}) \subseteq F_1\ ),\ S\ )$$

where "filter" is a generic operation which applies a predicate (given by the $\lambda$–expression) to each element of a container type, returning a similar container type which contains only those elements from the original which satisfy the predicate. Where $F_1$ represents the satisfiability set of a CTL formula $f$, the derived operation denoted by this term will compute the satisfiability set of the CTL formula $ax\ f$. It does this by extracting from $N$ those nodes that satisfy the condition that all of their successors satisfy the formula $f$.

Instead of extending the target algebra with these operations, we show in the following section how a specification language containing these constructs can be used in conjunction with the target language to write the appropriate derived operations. The deficiency of the target language algebras $\mathcal{A}_M^{sem}$ and $\mathcal{A}_M^{syn}$ is of the first variety we mentioned in the introduction. It is clear the every formula in $\mathcal{A}_{ctl}^{syn}$ has a representation of it satisfiability set in $\mathcal{A}_M^{syn}$ and $\mathcal{A}_M^{sem}$. The operations in $\mathcal{A}_M^{syn}$ however are not computationally powerful enough to compute the set expressions representing the satisfiability sets since they only create terms (set expressions) in $\mathcal{A}_M^{syn}$ by concatenating terms together. Similarly, the operations in $\mathcal{A}_M^{sem}$ perform set operations but there is no facility for general computation and thus we do not have the facilities to compute the satisfiability set for formulas created using the temporal operators.

The advantage of keeping the specification language separate from the target

---

[3] In a previous work [19,20] we have referred to these as *meta languages*.

language is that we can populate an algebraic language processing environment with several reusable specification languages which a language designer may use to build translators.

### *3.3 Evaluation of derived operations*

As we have seen, derived operations are specified by words from the target language syntax algebra $\mathcal{A}_T^{syn}(S'_S)$ over a sub-scripted set of specification variables from the source signature set of sorts $S_S$. In Figure 3, the same words from $\mathcal{A}_T^{syn}(S'_S)$ are used to specify the operations of the syntax algebra $\mathcal{A}_{ST}^{syn}$ and the semantics algebra $\mathcal{A}_{ST}^{sem}$. Thus, we could build a generalized homomorphism $H_e \colon \mathcal{A}_S^{syn} \to \mathcal{A}_{ST}^{sem}$ which maps words in $\mathcal{A}_S^{syn}$ directly to values in $\mathcal{A}_{ST}^{sem}$. $H_e$ is equivalent to the composition of the embedding morphism $em_{syn} \colon \mathcal{A}_S^{syn} \to \mathcal{A}_{ST}^{syn}$ and the $L_{ST}$ evaluation function $\mathcal{E}_{ST}$, i.e. $H_e = em_{syn} \circ \mathcal{E}_{ST}$. In the case of our model checker, such a homomorphism would map CTL formulas directly to their satisfiability sets in the intermediate semantic algebra. For efficiency reasons this may be desirable and is often the way we will actually implement model checkers as algebraic compilers.

## 4 Specification languages in algebraic compilers

A *specification language* as used in an algebraic compiler is a parameterized $\Sigma$-language used in conjunction with the target language to specify derived operations. It has the additional constructs required to correctly write the derived operations which specify the translation. In the functional instance of the model checker, these specification language operations will include the *filter* and $\lambda$–expression operators we saw above. In the imperative instance, the specification language constructs will include *if*, *while* and assignment constructs as well as a *for each* loop operation. These operations, in combination with the target language operations of set intersection, union, membership, etc., are used to write the derived operations specifying the model checker. In this section we first briefly discuss macro languages as instances of specification languages and then discuss specification languages in general and how they are *instantiated* with a specific target language to provide a language which can be used to specify the derived operations of an algebraic compiler. Following this we describe the functional and imperative specification languages and their instantiations with the model target language $L_M$.

### *4.1 Macro languages as specification languages*

Macro processing has long been used as a mechanism for implementing language translators [21–25]. Our colleagues and we have used macro processing in the framework of algebraic compilers in many different instances [26–28,18,29]. In all of these cases, the macro languages act as a kind of translator specification language. In the realm of algebraic compilers, the macro language acts as a specification language for specifying derived operations.

To use macro languages in specifying derived operations we specify, for each source language operation, a macro whose actual parameters are the target images of the components of the source language construct. Its formal parameters are the sub-scripted sorts from the signature of the source language operation. The process of expanding this macro at compile time generates the target language image of the source construct. Consider, for example, a translator for an imperative programming language whose target language is a stack machine assembly language. The source language has a binary addition operator with signature $\sigma(add) = Expr \times Expr \rightarrow Expr$ for integer or real number addition (without type coercion for simplicity). The target images of expressions are assembly language code fragments which leave their result on the top of the stack. We can thus specify the translation of *add* by the following (semantic) macro which upon macro expansion, generates the target language code fragment consisting of the target images of the components of *add* followed by the integer or real number add instruction, *addi* or *addr* respectively, depending on the type of the first component. This macro expands into code which computes the value of the expression and leaves that value on the top of the stack.

$add:\ Expr_0 ::= Expr_1\ Expr_2$
$macro:\ Expr_1$
$\quad Expr_2$
$\quad \#if\ \$type(Expr_1) = Integer$
$\quad addi$
$\quad \#else$
$\quad addr$
$\quad \#endif$

(By using Maddox's semantic macros [25] in algebraic compilers [18], we can access semantic information such as an expression's type, $\$type(Expr_1)$, during macro expansion.)

Of interest here is the fact that we've used the macro language *#if* construct to specify the derived operation which computes the target language image, in the target language syntax algebra, of *add* expressions. The *#if* construct is the required operation – which does not exist in the target language syntax algebra – we need to specify this derived operation. In this case, the target language has the first type of deficiency we mentioned in the introduction; although it contains the elements (target language programs) in the range of the compiler, it does not contain the operations required to correctly construct these target language images. A subtle point to observe is that although the target language may have a branch operation, in the target syntax algebra this operation would only concatenate words together; it would not perform the branch computation needed to determine which add instruction to use in the target image. The operations in the target syntax algebra only concatenate words together and have no computational facilities for branching. The macro language provides the required additional capabilities.

The specification languages presented in this paper should be seen as generalizations of macro languages, but the specification languages are defined algebraically and are independent of the target language. We are not adding new constructs to the target language to make the translation possible, but instead are introducing a specification language with the required constructs that, as we will see in the following section, sits between the source and target language and enables the translation.

### *4.2 Specification language instantiation*

A specification language $L_{Sp}$ used in an algebraic compiler is essentially a parameterized $\Sigma$–language. To use a specification language it must be *instantiated* with the target language of the algebraic compiler. Like all $\Sigma$–languages, a specification language has an operator scheme $\Sigma_{Sp}$, syntax and semantic algebras $\mathcal{A}_{Sp}^{sem}$ and $\mathcal{A}_{Sp}^{syn}$, and a language learning function $\mathcal{L}_{Sp}$. The operator scheme $\Sigma_{Sp}$ is the tuple $\langle S_{Sp}, Op_{Sp}, \sigma_{Sp} \rangle$ where $S_{Sp}$ and $Op_{Sp}$ are a set of sorts and operator names as seen above. The signatures of these operator names, however, may include parameters as well as sorts from $S_{Sp}$. That is, $\sigma_{Sp} \colon Op_{Sp} \to PS_{Sp}^{*} \times PS_{Sp}$, where $PS_{Sp} = S_{Sp} \cup Param$ and $Param$ is a set of parameter names. The syntax and semantic algebras of a specification language contain carrier sets and operations as expected, but these will be augmented with carrier sets and operations from the target language algebras. Components of the target language are the actual parameters which are used to instantiate the specification language so that it can be used in an algebraic compiler.

To write derived operations using specification ($L_{Sp}$) and target ($L_T$) language operations, the instantiation of the specification language is created (by the language processing environment) from these two languages. A specification language $L_{Sp}$ instantiated with a target language $L_T$ is denoted

$L_{Sp^T} = \langle \mathcal{A}^{sem}_{Sp^T}, \mathcal{A}^{syn}_{Sp^T}, \mathcal{L}_{Sp^T} \rangle$ with operator scheme $\Sigma_{Sp^T}$. To instantiate a specification language the following tasks must be performed:

1. *Instantiate the operator scheme* $\Sigma_{Sp^T}$. $\Sigma_{Sp^T} = \langle S_{Sp^T}, Op_{Sp^T}, \sigma_{Sp^T} \rangle$ where the set of sorts $S_{Sp^T}$ is the union of the specification and target sorts $S_{Sp} \cup S_T$ and the operator names $Op_{Sp^T}$ are the union of specification and target operator names $Op_{Sp} \cup \ Op_T$. The signatures of the instantiated operations are defined by $\sigma_{Sp^T} : Op_{Sp^T} \to {S_{Sp^T}}^* \times S_{Sp^T}$. Note that there are no parameters in these signatures. These signatures are created by replacing parameters in $\sigma_{Sp}$ signatures with sort names in $S_T$ and $S_{Sp}$ and adding the target languages signatures in $\sigma_T$. In our model checker, the target language sorts *Node*, *Set* and *Boole* replace the parameters in the specification language signatures. As we will see, this may cause $\sigma_{Sp^T}$ to be a relation instead of a function and thus the same operator name maps to several operations on different sorts.
2. *Instantiate the syntax algebra* $\mathcal{A}^{syn}_{Sp^T}$. The carrier sets of this algebra are the words with sorts $S_{Sp^T}$. These contain more than simply the appropriate union of the carrier sets of the uninstantiated specification language and the target language, but all words created by the operations in the instantiated syntax algebra. We need operations for each signature in $\sigma_{Sp^T}$ generated above. These operations may combine words from specification and target language sorts but these operations can be automatically constructed from the specification and target syntax algebra operations since they simply paste words together.
3. *Instantiate the semantic algebra* $\mathcal{A}^{sem}_{Sp^T}$. We must also instantiate the operations of this algebra. Either they are explicitly constructed for the new types, a kind of ad hoc polymorphism, or, preferably, the existing specification language operations are *generic* (polymorphic or polytypic) [30] and can thus automatically work on the data-types from the target algebra or are defined in terms of existing operations in the specification and target semantic algebras. This process is dependent on the specification and target algebras and is discussed in more detail below when we present the functional and imperative specification languages.

Derived operations for the generalized homomorphism are now written in $\mathcal{A}^{syn}_{Sp^T}(S'_S)$, the instantiated specification language word algebra with specification variables $S'_S$, instead of the syntax algebra $\mathcal{A}^{syn}_T(S'_S)$ of the target language $L_T$ as done before. Thus, the operator map *om* used in defining the generalized homomorphism has the signature $om : Op_S \to \mathcal{A}^{syn}_{Sp^T}(S'_S)$. It maps source language operators to words containing operator names from the specification and target language. These words specify the derived operations which create the target images of source language constructs. The sort map *sm* is the same as before so that target images of source language constructs are still objects of sorts in the target language, not sorts of the specification language.

When building such an algebraic compiler the hybrid intermediate language $L_{ST}$ from Figure 3 is replaced by the hybrid intermediate language $L_{SSp^T} = \langle \mathcal{A}^{sem}_{SSp^T}, \mathcal{A}^{syn}_{SSp^T}, \mathcal{L}_{SSp^T} \rangle$ as shown in Figure 4. Like $L_{ST}$, this language has the same operator scheme $\Sigma_S$ as the source language, but has operations built using the operations from $L_{Sp^T}$. The embedding morphisms $em_{syn}$ and $em_{sem}$ in Figure 4 are computed in the same manner as those in Figure 3. We also add an extra pair of identity injection mappings between $L_{SSp^T}$ and $L_{Sp^T}$.

$$\begin{tikzcd}
\mathcal{A}^{sem}_S \arrow[r, "\mathcal{L}_S"] \arrow[d, "em_{sem}"] & \mathcal{A}^{syn}_S \arrow[r, "\mathcal{E}_S"] \arrow[d, "em_{syn}"] & \mathcal{A}^{sem}_S \arrow[d, "em_{sem}"] \\
\mathcal{A}^{sem}_{SSp^T} \arrow[d, "im2_{sem}"] & \mathcal{A}^{syn}_{SSp^T} \arrow[l, "\mathcal{E}_{SSp^T}"'] \arrow[d, "im2_{syn}"] & \mathcal{A}^{sem}_{SSp^T} \arrow[l, "\mathcal{L}_{SSp^T}"'] \arrow[d, "im2_{sem}"] \\
\mathcal{A}^{sem}_{Sp^T} \arrow[r, "\mathcal{L}_{Sp^T}"] \arrow[d, "im1_{sem}"] & \mathcal{A}^{syn}_{Sp^T} \arrow[r, "\mathcal{E}_{Sp^T}"] \arrow[d, "im1_{syn}"] & \mathcal{A}^{sem}_{Sp^T} \arrow[d, "im1_{sem}"] \\
\mathcal{A}^{sem}_T & \mathcal{A}^{syn}_T \arrow[l, "\mathcal{E}_T"'] & \mathcal{A}^{sem}_T \arrow[l, "\mathcal{L}_T"']
\end{tikzcd}$$

Fig. 4. An algebraic compiler with a meta language layer.

Just as the intermediate hybrid language $L_{ST}$ in Figure 3 is automatically created, so is $L_{SSp^T} = \langle \mathcal{A}^{sem}_{SSp^T}, \mathcal{A}^{syn}_{SSp^T}, \mathcal{L}_{SSp^T} \rangle$. However, we do need to explicitly create (portions of) the specification language $L_{Sp^T}$ using the process sketched above and employed for the functional and imperative languages below. But, this makes sense; we should not expect to get this language entirely "for free." Whereas before we specified the source and target language of the algebraic compiler and wrote derived operations in the target syntax algebra with specification variables, we must now specify the specification language we wish to use as well. The derived operations are then written in the instantiated specification language syntax algebra.

An appropriate set of algebraic language processing tools can automatically instantiate much, if not all, of the specification language. Since the syntax algebra operations can always be automatically instantiated, it is the semantic algebra operations – the ones which do the actual computation in algebraic compilers – which may in a few cases need to be done by hand. The degree to which this process can be automated for a specification language determines the convenience of using that specification language. If the specification language semantic algebra operations are polymorphic, polytypic (generic) or defined in terms of existing operations in the specification and target algebra then this process can be automated as is the case for the specification languages presented here.

### *4.3 A functional specification language*

As alluded to above, we can use a functional specification language in specifying our algebraic model checker $MC\colon L_{ctl} \to L_M$. This allows us to write derived operations for the temporal logic operators $ax, ex, au$, and $eu$ using functional language constructs and thus provide concise specifications for our model checker. Although a functional specification language would have many other higher order functions, like *map* and *fold*, we only describe here the operations which are used in our algebraic specification of the model checker. We do however use $\lambda$ expressions and higher order functions *filter*, *limit* and *iterate* which are defined below.

Our functional specification language $L_F = \langle\ \mathcal{A}_F^{sem},\ \mathcal{A}_F^{syn},\ \mathcal{L}_F\rangle$ has operator scheme $\Sigma_F = \langle S_F = \{Boole, Var, Func, List\},\ Op_F = \{not,\ and,\ empty,$ $insert,\ get\_one,\ get\_rest,\ \lambda,\ fetch,\ limit,\ iterate,\ filter\}, \sigma_F\rangle$, where $\sigma_F$ uses the parameter $a \in Param$ and is defined in Table 4.

| | | | | |
|---|---|---|---|---|
| $\sigma_F(not)$ | $=$ | $Boole$ | $\to$ | $Boole$ |
| $\sigma_F(and)$ | $=$ | $Boole \times Boole$ | $\to$ | $Boole$ |
| $\sigma_F(empty)$ | $=$ | $\emptyset$ | $\to$ | $List$ |
| $\sigma_F(insert)$ | $=$ | $a \times List$ | $\to$ | $List$ |
| $\sigma_F(get\_one)$ | $=$ | $List$ | $\to$ | $a$ |
| $\sigma_F(get\_rest)$ | $=$ | $List$ | $\to$ | $List$ |
| $\sigma_F(\lambda)$ | $=$ | $Var \times a$ | $\to$ | $Func$ |
| $\sigma_F(fetch)$ | $=$ | $Var$ | $\to$ | $a$ |
| $\sigma_F(limit)$ | $=$ | $List$ | $\to$ | $a$ |
| $\sigma_F(iterate)$ | $=$ | $Func \times a$ | $\to$ | $List$ |
| $\sigma_F(filter)$ | $=$ | $Func \times a$ | $\to$ | $a$ |

Table 4
The signature $\sigma_F$ of $Op_F$.

The *Boole* sort is for Boolean values and variables and corresponds to the *Boole* sort from the model operator scheme $\Sigma_M$. Note that the operators *not* and *and* above are distinct from those in CTL. *Var* is for variables used in $\lambda$–expressions. As indicated by their names, *Func* is for functions and *List* for lists.

The syntax algebra $\mathcal{A}_F^{syn}$ has operations for building words (programs) and carrier sets which contains these words. The syntax operations are defined as we expect.

The semantic algebra $\mathcal{A}_F^{sem}$ provides an evaluation of programs in the syntactic algebra. The carrier sets contain the values which result from the evaluation of the specification language constructs such as *not* and *filter*. The *Boole* carrier set in $\mathcal{A}_F^{sem}$ contains the semantic value *true* and *false* and the semantic operations *and* and *not* are the expected Boolean operations. The semantic carrier set *Func* contains, as expected, functions. In this language, and higher order functional languages in general, the semantic algebra operations are first class citizens of the language which means that these *operations* are also *elements* of the *Func* carrier set.

The *List* carrier set is slightly different since we would like to allow (possibly) infinite lists to be represented in our specification language. Thus the *List* carrier set will contain "lazy lists" implemented as *list computations* which are evaluated lazily to create list values only as they are needed. One could correctly say that all operations in this language are strict except for the list operations which are non-strict and calculated via lazy evaluation. The semantic operation *limit* is a function which lazily evaluates a list of elements, returning the first element in the list which is followed by a element of the same value. For example, $limit[1, 2, 3, 3, 4, 5, \ldots]$ evaluates to 3. Even if the elements of this operand list continue to increase and thus form an infinite list, the limit operation is well defined since the lists are lazily evaluated. That is, we do not first compute the entire (in this case infinite) list and pass it to the *limit* operation, but pass the *list computation* which could potentially build this infinite list. Since the limit operation will only query its operand for a succeeding elements of the list if the previous two values were different it is possible for *limit* to return the value 3 above without calculating the complete value of the infinite list. On lists where there are no two adjacent equal values, the *limit* function does not terminate and thus *limit* is a partial function. This operation is a polymorphic operation in that it works on lists containing elements of any sort, as long as there is an equality operation on values of that sort.

The list manipulation operation *empty* creates the empty lazy list, *insert* creates a new list by adding an element to the beginning of another list, *get_one* returns the first element in a list, and *get_rest* returns the list containing all but the first element of a list.

The *filter* operation applies a Boolean function to each element of a container type, and constructs a new container type with only those original elements which evaluate to *true* under the Boolean function. *filter* is defined as follows:

```
$filter\ (f, c) =$ if $c = empty$ then $empty$
                   else if $f(get\_one(c))$
                        then $insert(get\_one(c), filter(f, get\_rest(c)))$
                        else $filter(f, get\_rest(c))$
```

Since the container *List* has implementations of operations =, *empty*, *get_one*, *get_rest*, and *insert* operations, the filter operation can be applied to lists. Note that filter examines every element of the container type and thus if it is applied to a list, that list must be finite or the computation will not terminate.

The *iterate* semantic operation is also lazy and repeatedly applies a unary function first using a given initial value and then to the value returned from the previous application. That is, $\mathit{iterate}(f, x) = \mathit{insert}(x, (\mathit{iterate}(f, f(x))))$. For example, with an initial integer value 3 and the increment-by-one function *inc*, *iterate inc* 3 produces an infinite lazy list that when evaluated produces the values $[3, 4, 5, 6, ...]$.

### *4.3.1 Instantiating the functional specification language.*

To use this functional specification language in our model checker specifications we must first instantiate it with the target language $L_M$. We can create the instantiated specification language $L_{FM} = \langle \mathcal{A}^{sem}_{FM}, \mathcal{A}^{syn}_{FM}, \mathcal{L}_{FM} \rangle$ with the operator scheme $\Sigma_{FM}$ from the specification language $L_F$ and the model language $L_M$ using the process described above. We begin by instantiating new operator signatures by replacing the parameter $a$ in $\sigma_F$ with sort names *Set*, *Node* and *Boole* from the model language operator scheme $\Sigma_M$. We will thus create new signatures for operations which previously did not exist, such as $\sigma_{FM}(\mathit{filter}) = \mathit{Func} \times \mathit{Set} \rightarrow \mathit{Set}$.

Instantiating the operations in the syntax algebra can be done automatically since they simply paste together words and the sorts of the component words do not affect the operations behavior. In the case of $\sigma_{FM}(\mathit{filter}) = \mathit{Func} \times \mathit{Set} \rightarrow \mathit{Set}$ the *filter* operation from the syntax algebra $\mathcal{A}^{syn}_F$ is also used to create words of the sort *Set*. All syntax algebra operations can be instantiated in this way.

We must also instantiate the operations in the semantics algebra. The semantic operations from $\mathcal{A}^{sem}_M$ are included in $\mathcal{A}^{sem}_{FM}$ as they are. More interestingly, some of the operations suggested by the instantiation of signatures by replacing parameters with sort names would be invalid and would not be used in any program. Thus, we need not concern ourselves with creating semantic operations for these signatures. For example, consider instantiating the signature $\sigma_F(\mathit{filter}) = \mathit{Func} \times a \rightarrow a$ by replacing parameter $a$ with the sort *Boole*. It is invalid to apply a *filter* operation to a Boolean value since there are no *get_one* or *get_rest* operations for *Boole* values. Thus we do not provide an implementation for this semantic operation. In some cases, there may also be valid operations which we do not intend to use in the derived operations of the algebraic compiler, and therefore we do not need to instantiate them either.

We do, however, need some new operations; for example, the operations with the signatures $\sigma_{FM}(\mathit{filter}) = \mathit{Func} \times \mathit{Set} \rightarrow \mathit{Set}$ and $\sigma_{FM}(\mathit{iterate}) = \mathit{Func} \times \mathit{Set} \rightarrow \mathit{List}$ are used in our derived operations. Do we have to manually provide implementations for this operations? No, since *filter* and *iterate* are defined in terms of existing operations, its implementation is automatically provided. In the case of *filter* over sets, since the sort *Set* has implementations of the operations =, *empty*, *get_one*, *get_rest*, and *insert* the filter operation can be applied to sets as well as lists.

Clearly, these operation names were not chosen by accident or included in the model language without an understanding how they would ultimately be used. This is similar to what happens in modern programming languages such as Java [31] and Haskell [32]. In Java, an “interface” plays the role of what we have presented above. A class is said to “implement an interface” if it provides method definitions for the methods named in the interface. In our case, we could have a *filter* interface consisting of method names =, *empty*, *get_one*, *get_rest*, and *insert*. The *filter* operation could then be applied to sets if the *Set* sort implements these operations. In Haskell, a similar functionality is provided by type classes. A data type is a member of a type class if it provides implementations for the functions named in the type class. We might define a *filter* type class to contain the signatures of the required operations and define *Set* to be an instance of that type class by providing definitions of these functions.

In the case of *iterate*, no restrictions are placed on the parameter sort since *iterate* creates lists by lazily applying the function to values of that sort to create a list of elements of that sort. The iterate operation provided by the uninstantiated specification language in $\mathcal{A}_F^{sem}$ is polymorphic and works with functions and initial values of any type, assuming of course that the functions input and output types are those of the initial value.

As we will see in Section 5.1, the language learning function $\mathcal{L}_{FM}$ is not used directly in the model checker, but the evaluation function $\mathcal{E}_{FM}$ is. It executes programs in $\mathcal{A}_{FM}^{syn}$ by mapping them to their values in $\mathcal{A}_{FM}^{sem}$.

## *4.4 An imperative specification language*

We similarly design an imperative specification language $L_I = \langle\ \mathcal{A}_I^{sem},\ \mathcal{A}_I^{syn},\ \mathcal{L}_I\rangle$ that has operator scheme $\Sigma_I = \langle S_I, Op_I, \sigma_I\rangle$. The sort set contains sorts $S_I = \{\mathit{Expr}, \mathit{ExprList}, \mathit{Dcl}, \mathit{DclList}, \mathit{Var}, \mathit{Boole}\}$ for expressions, declarations, variables and Boolean expressions, as are familiar in imperative languages. For simplicity, we will not make a syntactic distinction between expressions and statements as is normally done. Some of our expressions will have side effects and thus change the memory state in the same manner as statements do and others will be side effect free like traditional expressions. The operator names $Op_I$ includes the familiar imperative language operations; $Op_I = \{let, begin, if, while, for\ each, assign, not,\ and, ...\}$. The *if*, *while* and *assign* operators are as expected. The *for each* operation executes an expression for each element of a container value. The *let* operation allows the introduction of local variables. The "value" of a let binding is the value of the final expression in its body. The *begin* operator is simply a *let* operation without any declarations. These operator's signatures, and others as defined by $\sigma_I$, are shown in Table 5 where $a \in Param$.

$$\begin{array}{lcrcl}
\sigma_I(let) & = & DclList \times ExprList & \rightarrow & Expr \\
\sigma_I(begin) & = & ExprList & \rightarrow & Expr \\
\sigma_I(if) & = & Boole \times Expr & \rightarrow & Expr \\
\sigma_I(while) & = & Boole \times Expr & \rightarrow & Expr \\
\sigma_I(for\ each) & = & Var \times a \times Expr & \rightarrow & Expr \\
\sigma_I(assign) & = & Var \times Expr & \rightarrow & Expr \\
 & & & & \\
\sigma_I(not) & = & Boole & \rightarrow & Boole \\
\sigma_I(and) & = & Boole \times Boole & \rightarrow & Boole \\
 & & & & \\
\sigma_I(fetch) & = & Var & \rightarrow & Expr \\
\sigma_I(dcl) & = & Var \times Expr & \rightarrow & Dcl \\
\sigma_I(elist_1) & = & Expr & \rightarrow & ExprList \\
\sigma_I(elist_2) & = & ExprList \times Expr & \rightarrow & ExprList \\
\sigma_I(dlist_1) & = & Dcl & \rightarrow & DclList \\
\sigma_I(dlist_2) & = & DclList \times Dcl & \rightarrow & DclList \\
\sigma_I(expr_1) & = & a & \rightarrow & Expr \\
\sigma_I(expr_2) & = & Expr & \rightarrow & a
\end{array}$$

Table 5
The signature $\sigma_I$ of $Op_I$.

As with the functional specification language $L_F$, the imperative syntax algebra $\mathcal{A}_I^{syn}$ contains words, that is programs, written in this imperative language and its operations are defined as expected. For example, the *for each* syntax operation is $for\ each_{syn}(v, e, s) = for\ each\ v\ in\ e\ s.$

The semantic algebra $\mathcal{A}_I^{sem}$ is slightly different from the functional semantic algebra $\mathcal{A}_F^{sem}$ in that carrier sets contain computations, not values, and the operations build these computations. We define, in the traditional manner, a *state* as a mapping $\mathit{State}\colon \mathit{Name} \rightarrow \mathit{Value}$ from variable names to values. A computation is then a mapping of type $\mathit{State} \rightarrow \langle \mathit{Value}, \mathit{State} \rangle$ that takes a state and returns a value and possibly updated state.

The semantic *not* operation in $\mathcal{A}_I^{sem}$ is a function $\mathit{not}(b) = \lambda st \rightarrow \langle \neg v, st' \rangle$ where $\langle v, st' \rangle = b(st)$. It takes a computation $b$, which when given the state $st$ returns the value $v$ of $b$ in state $st$ a the possibly updated state $st'$. The new computation is the function which takes $st$ and returns the negation of $v$ and the state $st'$. The *assign* operator is a function $\mathit{assign}(x, e) = \lambda st \rightarrow \langle v, st' \rangle$ that maps an input state $st$ to an output state $st'$ that maps $x$ to the value of $e$ in state $st$ and $v$ is also the value of $e$ in state $st$. That is, $\mathit{assign}(x, e) = \lambda st \rightarrow \langle v, st' \rangle$ where $\langle v, st'' \rangle = e(st)$ and $st' = st''[x \mapsto v]$. (The state $st[x \mapsto v]$ is the same as $st$ except it maps $x$ to $v$.) Similarly, $\mathit{fetch}(v) = \lambda st \rightarrow \langle st(v), st \rangle$ and $\mathit{dcl}(v, e) = \lambda st \rightarrow \langle -, st'[v \mapsto v_e] $ where $\langle v_e, st' \rangle = e(st)$. ($-$ represent the undefined value.) The operations $\mathit{expr}_1$ and $\mathit{expr}_2$ are used to shuffle values between sorts as needed.

The *for each* semantic operation is defined in terms of existing operations in the specification and target semantic algebras as follows:

```
$for\ each(v, e, s)$ = let $t_1$ := $e$ ,
                           $v$ := $-$
                       in  while not ( $t_1$ = empty ) begin
                                 $v$ := get_one ( $e$ ) ;
                                 $t_1$ := get_rest ( $t_1$ ) ;
                                 $s$
                           end
                       end
```

This operation is the imperative version of the *filter* function in the language $L_F$ and works with any container type implementing the operations *empty*, $=$, *get_one*, and *get_rest*.

### *4.4.1 Instantiating the imperative specification language.*

Instantiating the specification language $L_{IM} = \langle \mathcal{A}_{IM}^{syn}, \mathcal{A}_{IM}^{sem}, \mathcal{L}_{IM} \rangle$ with operator scheme $\Sigma_{IM}$ from $L_I$ and $L_M$ proceeds in the same manner as with the functional specification language. The new operator scheme $\Sigma_{IM}$ is created by replacing the parameter $a$ in $\Sigma_I$ with sort names from $S_I$ and $S_M$. The syntax operations in $\mathcal{A}_{IM}^{syn}$ can be automatically instantiated as before.

Again, it is the instantiation of the semantics algebra $\mathcal{A}_{IM}^{sem}$ which is most interesting. As with the *filter* operation in $L_F$, the semantic *for each* operator is defined in terms of operations in the specification and target language.

Since there are *empty*, =, *get_one* and *get_rest* operations defined on *Sets* the *for each* construct can be instantiated to create the "set iterator" operation with signature $Var \times Set \times Expr \to Expr$.

In this case however, we can not simply include the carrier sets and operations from the target language semantic algebra $\mathcal{A}_M^{sem}$ as they are. In $\mathcal{A}_{IM}^{sem}$, the *Set*, *Node* and *Boole* carrier sets must now be computations with the type $State \to \langle Value, State \rangle$ instead of simple values. In fact, *Set*, *Node* and *Boole* become synonyms for $State \to \langle Value, State \rangle$. The operations from $\mathcal{A}_M^{sem}$ must also be modified to take such types as operands. Thus we will *lift* the semantics algebra operations to take operands of type $State \to \langle Value, State \rangle$. Below, we will subscript sort names and operations with $M$ to indicate the originals from $\mathcal{A}_M^{sem}$; for example, $U_M$ represents the original set union operator and $Set_M$ represents the original *Set* carrier in $\mathcal{A}_M^{sem}$. Consider the set union operation with signature $\sigma_M(\cup) = Set \times Set \to Set$. In $\mathcal{A}_{IM}^{sem}$, $s_1 \cup s_2 = \lambda st \to \langle v_1 \cup_M v_2, st_2 \rangle$ where $\langle v_1, st_1 \rangle = s_1(st)$ and $\langle v_2, st_2 \rangle = s_2(st_1)$. That is, the state transformation will compute the set value $v_1$ and (possibly new) state $st_1$ from $s_1$ using the input state $st$. The state $st_1$ is used by $s_2$ to compute the set value $v_2$ and (possibly new) state $st_2$. The set value $v_1 \cup_M v_2$ is computed using the set union operator from $\mathcal{A}_M^{sem}$. The other component generated by the state transformation is the new state $st_2$.

As before, the learning function $\mathcal{L}_{IM}$ is not used in the implementation of the model checker. Its existence is important in correctness proofs. The evaluation function $\mathcal{E}_{IM}$ maps programs in $\mathcal{A}_{IM}^{syn}$ to state transforming computations in $\mathcal{A}_{IM}^{sem}$. The manner in which these computations are used in implementing the model checker is discussed in Section 5.2.

## 5 Model checker specification and implementation

In this section we show the specifications for the algebraic model checker using the functional and imperative specification languages. We will write the translation specifications for each CTL operation $op \in Op_{ctl}$, by writing the signature of the operation, $\sigma_{ctl}(op)$, followed by its derived operation in the target, $om(op)$, but we will drop the *om* for convenience. The operations signatures are written with the output sort of each operation to the left and the operation name split between the input sorts in a BNF notation. (In fact, some algebraic tools like TICS [28] use this specification to generate a parser for the source language.) The specification variables used in the derived operations are indexed source language sorts found in the source operation signature. In the derived operations, a specification variable for an input sort represents the target image of the corresponding source language component. These specifications are processed by an algebraic language processing environment to automatically generate the model checker [16,33,10].

### *5.1 A functional model checker specification*

With the instantiated functional specification language $L_{FM}$ we can write derived operations to implement a model checker. Figure 4 is replicated in Figure 5 using the functional specification language $L_{FM}$.

$$\begin{tikzcd}
\mathcal{A}^{sem}_{ctl} \arrow[r, "\mathcal{L}_{ctl}"] \arrow[d, "em_{sem}"] & \mathcal{A}^{syn}_{ctl} \arrow[r, "\mathcal{E}_{ctl}"] \arrow[d, "em_{syn}"] \arrow[dl, "em_{alt}"'] & \mathcal{A}^{sem}_{ctl} \arrow[d, "em_{sem}"] \\
\mathcal{A}^{sem}_{SFM} \arrow[d, "im2_{sem}"] & \mathcal{A}^{syn}_{SFM} \arrow[l, "\mathcal{E}_{SFM}"'] \arrow[d, "im2_{syn}"] & \mathcal{A}^{sem}_{SFM} \arrow[l, "\mathcal{L}_{SFM}"'] \arrow[d, "im2_{sem}"] \\
\mathcal{A}^{sem}_{FM} \arrow[r, "\mathcal{L}_{FM}"] \arrow[d, "im1_{sem}"] & \mathcal{A}^{syn}_{FM} \arrow[r, "\mathcal{E}_{FM}"] \arrow[d, "im1_{syn}"] & \mathcal{A}^{sem}_{FM} \arrow[d, "im1_{sem}"] \\
\mathcal{A}^{sem}_{M} & \mathcal{A}^{syn}_{M} \arrow[l, "\mathcal{E}_{M}"'] & \mathcal{A}^{sem}_{M} \arrow[l, "\mathcal{L}_{M}"']
\end{tikzcd}$$

Fig. 5. An algebraic compiler with a functional specification language layer.

As before, the intermediate hybrid language $L_{SFM} = \langle \mathcal{A}^{sem}_{SFM}, \mathcal{A}^{syn}_{SFM}, \mathcal{L}_{SFM} \rangle$ is automatically created from the source language $L_{ctl}$ and the instantiated specification language $L_{FM}$. It has the same operator scheme as $L_{ctl}$ but its carrier sets contain elements from $L_{FM}$ and its operations are derived operations composed from the operations in $L_{FM}$. The embedding morphism $em_{syn}$ is defined as before in (1) in Section 3 and specified by the derived operations given below. The embedding morphism $em_{sem}$ and the injection mappings $im1_{sem}$ and $im2_{sem}$ are simply identity mappings since the semantic algebras $\mathcal{A}^{sem}_{ctl}$, $\mathcal{A}^{sem}_{SFM}$, $\mathcal{A}^{sem}_{FM}$ and $\mathcal{A}^{sem}_{M}$ all contain the same sets of nodes from the model $M$. The injection mapping $im2_{syn}$ is also an identity mapping. The injection mapping $im1_{syn}$ is not implemented directly since it requires mapping words/programs in $L_{FM}$, which defines how to compute sets, to set words in $L_M$. Thus, $im1_{syn}$ can be implemented as the composition $\mathcal{E}_{FM} \circ im1_{sem} \circ \mathcal{L}_M$.

The functional version of the algebraic model checker $MC_F$ maps CTL formulas in $\mathcal{A}^{syn}_{ctl}(AP)$ to their satisfiability sets in $\mathcal{A}^{syn}_{M}$. It can be defined as $MC_F = em_{syn} \circ im2_{syn} \circ \mathcal{E}_{FM} \circ im1_{sem} \circ \mathcal{L}_M$. A CTL formula is first mapped by $em_{syn}$ to a word in $\mathcal{A}^{syn}_{SFM}$ and then by the identity $im2_{syn}$ into $\mathcal{A}^{syn}_{FM}$. This word is a program in the instantiated specification language which when executed computes the satisfiability set of the CTL formula. The language evaluation function $\mathcal{E}_{FM}$ performs exactly this function. Since the result of this evaluation is a set, it is in the domain of the partial identity mapping $im1_{sem}$ which maps it into $\mathcal{A}_M^{sem}$. The language learning relation $\mathcal{L}_M$ can map this set into a simple representation in $\mathcal{A}_M^{syn}$. Here, $\mathcal{L}_M$ acts as a simple *output* mechanism to display the set. Thus, although we do not use the language learning relation $\mathcal{L}_{ctl}$ in the model checker, we do use the language learning relation $\mathcal{L}_M$ of the target language. As suggested in Section 3.3 we can implement $MC_F$ in an alternative way using the embedding $em_{alt}$ shown in Figure 5.

All that is left to do to specify $MC_F$ is to define the derived operations via the operator map $om$: $Op_{ctl} \rightarrow \mathcal{A}_{FM}^{syn}(S'_{ctl})$. For the non–temporal operators in $L_{ctl}$ we have the straightforward derived operations shown below:

| | | | |
|---|---|---|---|
| $F_0 ::= true$ | $F_0 ::= false$ | $F_0 ::= \neg\ F_1$ | $F_0 ::= F_1 \wedge F_2$ |
| $N$ | $\emptyset$ | $N \setminus F_1$ | $F_1 \cap F_2$ |

The operation *true* has the derived operation $N$ (shown directly below it) indicating that the satisfiability set of *true* is the full set of nodes $N$ in the model $M$; *false* has derived operation $\emptyset$ indicating that the satisfiability set of *false* is the empty set. The derived operation associated with $\neg$ shows that the satisfiability of $\neg\ f$ is the set difference of $N$ and the satisfiability set of $f$, denoted by the sort name $F_1$. Similarly, $\wedge$ is specified by the intersection of the satisfiability sets of the two sub formulas respectively denoted $F_1$ and $F_2$.

In the derived operation for *ax*, seen below, we see the use of some specification language constructs. Here, we define the satisfiability set of *ax* $f$ by filtering the set of nodes by a function which selects only those nodes such that all of their successors are in the satisfiability set of $f$.

$$F_0 ::= axF_1 \qquad \text{filter}\ (\ \lambda\ \text{n} \rightarrow\ succ(\text{n}) \subseteq F_1\ ,\ N\ )$$

The derived operation for *au* is similar, but uses the *limit* and *iterate* operations to implement a type of least fixed point operator of the function specified by the $\lambda$–expression.

$$F_0 ::= a[F_1\ u\ F_2]$$
$$\text{limit}\ (\ \text{iterate}\ \ (\ \lambda\ \text{z} \rightarrow \text{z} \cup \text{filter}\ (\ \lambda\ \text{n} \rightarrow\ succ(\text{n}) \subseteq \text{z},\ F_1\ )\ , F_2\ )\ )$$

The atomic propositions, specified as variables $AP$ in $\mathcal{A}_{ctl}^{syn}(AP)$, are mapped to their satisfiability set by the model labeling function $P$.

$$F_0 ::= p \qquad P(p)$$

### *5.2 An imperative model checker specification*

With the instantiated imperative specification language $L_{IM}$ we can write derived operations that will implement a CTL model checker. Figure 6 shows the intermediate languages and mappings from Figure 4 using the imperative specification language $L_{I^M}$.

$$\begin{tikzcd}
\mathcal{A}_{ctl}^{sem} \arrow[r, "\mathcal{L}_{ctl}"] \arrow[d, "em_{sem}"] & \mathcal{A}_{ctl}^{syn} \arrow[r, "\mathcal{E}_{ctl}"] \arrow[d, "em_{syn}"] \arrow[dl, "em_{alt}"'] & \mathcal{A}_{ctl}^{sem} \arrow[d, "em_{sem}"] \\
\mathcal{A}_{SI^M}^{sem} \arrow[d, "im2_{sem}"] & \mathcal{A}_{SI^M}^{syn} \arrow[l, "\mathcal{E}_{SI^M}"'] \arrow[d, "im2_{syn}"] & \mathcal{A}_{SI^M}^{sem} \arrow[l, "\mathcal{L}_{SI^M}"'] \arrow[d, "im2_{sem}"] \\
\mathcal{A}_{I^M}^{sem} \arrow[r, "\mathcal{L}_{I^M}"] \arrow[d, "im1_{sem}"] & \mathcal{A}_{I^M}^{syn} \arrow[r, "\mathcal{E}_{I^M}"] \arrow[d, "im1_{syn}"] & \mathcal{A}_{I^M}^{sem} \arrow[d, "im1_{sem}"] \\
\mathcal{A}_{M}^{sem} & \mathcal{A}_{M}^{syn} \arrow[l, "\mathcal{E}_{M}"'] & \mathcal{A}_{M}^{sem} \arrow[l, "\mathcal{L}_{M}"']
\end{tikzcd}$$

Fig. 6. An algebraic compiler with an imperative specification language layer.

As with the functional specification language, the intermediate hybrid language $L_{SI^M} = \langle \mathcal{A}_{SI^M}^{sem}, \mathcal{A}_{SI^M}^{syn}, \mathcal{L}_{SI^M} \rangle$ is automatically created from the source language $L_{ctl}$ and the instantiated imperative specification language $L_{I^M}$. The operator scheme of $L_{SI^M}$ is the same as $L_{ctl}$ but its carrier sets contain elements from $L_{I^M}$ and its operations are derived operations composed from $L_{I^M}$ operations. We again specify the embedding morphisms $em_{syn}$ and $em_{sem}$ as in (1) by the derived operations given below. The embedding morphism $em_{sem}$ maps a satisfiability set $s$ in $\mathcal{A}_{ctl}^{sem}$ to a state transforming computation of type $State \rightarrow \langle Value, State \rangle$ in the $Set$ carrier set in $\mathcal{A}_{SI^M}^{sem}$ that maps any state $st$ to the pair $\langle s, st \rangle$. The injection mappings $im2_{sem}$ and $im2_{syn}$ are simply identity mappings. As before, the injection mapping $im1_{syn}$ is not implemented directly since it requires mapping words in $L_{I^M}$ to set words in $L_M$; it is implemented as $im1_{syn} = \mathcal{E}_{I^M} \circ im1_{sem} \circ \mathcal{L}_M$. The injection mapping $im1_{sem}$ is a partial mapping which maps state transformation computations in the carrier set $Set$ to satisfiability sets in $\mathcal{A}_M^{sem}$. The computations in the domain of $im1_{sem}$ are imperative computations that map states to value/state pairs. Thus, $im1_{sem}$ maps a computation $c$ to a satisfiability set $s$ by evaluating $c$ with an initial state $st_0$ (that maps variables to an undefined value) and extracting the value from the resulting value/state pair. That is, $im1_{sem} :: (State \rightarrow \langle Value, State \rangle) \rightarrow Set_M$ and $im1_{sem}(c) = s$ where $\langle s, st' \rangle = c(st_0)$.

The imperative implementation of the model checker, $MC_I$, maps formulas in $\mathcal{A}_{ctl}^{syn}(AP)$ to their satisfiability sets in $\mathcal{A}_M^{syn}$ in much the same manner as the functional version $MC_F$. It is defined using the mappings in Figure 6 as $MC_I = em_{syn} \circ im2_{syn} \circ \mathcal{E}_{I^M} \circ im1_{sem} \circ \mathcal{L}_M$. Alternatively, $MC_I$ can be implemented using the syntax to semantic embedding $em_{alt}$.

In order to specify $MC_I$ we only need to define the derived operations via the operator map $om \colon Op_{ctl} \rightarrow \mathcal{A}_{I^M}^{syn}(S'_{ctl})$. Since the non–temporal CTL opera-

```
$F_0$ ::= $axF_1$
    let $t_1$ := $\emptyset$
    in  for each n in $S$
            if ( $succ$(n) $\subseteq$ ($F_1$) then
               $t_1$ := $t_1 \cup$ { n };
        $t_1$
    end
```

```
$F_0$ ::= $a[F_1\ u\ F_2]$
    let $t_1$ := $\emptyset$ ,
        $t_2$ := $F_2$
    in  while ( not $t_1 = t_2$ ) begin
                $t_1$ := $t_2$ ;
                for each n in $F_1$ do
                    if ($succ$(n) $\subseteq t_1$ ) then
                       $t_2$ := $t_2 \cup \{n\}$ };
        end ;
        $t_1$
    end
```

Fig. 7. The specifications for *ax* and *au* in the imperative specification language.

tors do not use any specification language constructs in their derived operations, they are the same in the imperative specifications as in the functional specifications in Section 5.1. We do not repeat them here and only show the specifications for the temporal operators *ax* and *au* in Figure 7. These derived operations are the imperative versions of the functional derived operations given above in Section 5.1. Here, the *while* and *for each* operators are used to implement a least fixed point operation to compute satisfiability sets.

### *5.3 Discussion*

The specification languages described here are just the required subsets of general purpose specification languages which would populate an algebraic language processing environment. Specification languages should be reusable components in such an environment so that algebraic compiler designers can choose from a collection of existing specification languages in which to write their translator specifications. A well–stocked environment would have functional and imperative specification languages giving the language designer some choice based on personal preference of language style.

An advantage of using a separate specification language, like $L_F$ or $L_I$, over extending the target language is that the specification language can be reused with a different target language in a different algebraic compiler. Both specification languages $L_F$ and $L_I$ can be seen as generalizations of the macro languages discussed earlier and could be used to replace the macro languages used in algebraic compilers for translating programming languages [18]. Because of their generality, they could be reused in many types of algebraic compilers, from traditional programming language translators to problems not typically stated as translations like the model checking example presented in this paper.

We would also expect an algebraic language processing environment to contain *domain specific specification languages* [19] with specialized constructs to address issues found in specific domains commonly encountered in language processing as well as other domains, such as temporal logic model checking, which also have solutions as algebraic compilers. Traditional language processing tasks with specific domains include type checking, optimization and parallelization, and code generation. In a type checker, for example, the target algebras would have operators for the base types and type constructors and carrier sets containing types or type expressions. A domain specific specification language for type checking which has specific constructs for managing symbol tables and environments would be helpful to the implementer and reusable in different compilers. In the case of the model checker, a domain specific specification language would include a least fixed point operator, since this domain would make good use of such a construct.

## 6 Related Work

### *6.1 Specification languages in other algebraic compiler models*

In this paper we have concentrated on Rus's [2] algebraic compiler model. An important question is whether or not these ideas can be used in other models of algebraic compilers. There are several other models described in the literature and these works tend to concentrate on either the algebraic *definition* of compilers or the algebraic *construction* of compilers. The work of Morris [34] and Thatcher et al. [3] fall into the first category which provides a definition of a compiler via mappings between the source and target languages and their semantics and shows the compiler correctness by proving that the diagrams created by these mappings commute [35]. The algorithm which implements the compiler is not necessarily of interest here. The second category, the algebraic construction of compilers, contains works which define a compiler algorithm in an algebraic framework in which the compiler correctness can be proved. Works by Mosses [4], Gaudel [36] and Rus [2] fall into this category. Below we discuss how specification languages can fit into these models in these different categories.

#### *6.1.1 Algebraic definition of compilers*

In our discussion of the algebraic definition of compilers we will focus on the model of Thatcher et al. [3]. They present an algebraic compiler for a source language $L$, target language $T$, source language meanings $M$ and target language meanings $U$ in which all are similar heterogeneous algebras. The mappings, shown in Figure 8, between these algebras are all homomorphisms:

$\gamma$ is the "compile" mapping, $\theta$ is the "source semantics" mapping, $\psi$ is the "target semantics" mapping, and $\varepsilon$ is the "encode" mapping. It is Thatcher's proof that $\varepsilon$ is a homomorphism and that the diagram in Figure 8 commutes that provide their definition and proof of compiler correctness. (Morris has a homomorphism $\delta\colon U \to M$ instead of $\varepsilon$.)

$$\begin{tikzcd}
L \arrow[r, "\gamma"] \arrow[d, "\theta"] & T \arrow[d, "\psi"] \\
M \arrow[r, "\varepsilon"] & U
\end{tikzcd}$$

Fig. 8. The algebraic compiler model of Thatcher et al.

$$\begin{tikzcd}
L \arrow[r, "\gamma"] \arrow[d, "\theta"] & T \arrow[r, "\gamma'"] \arrow[d, "\psi"] & T_0 \arrow[d, "\psi'"] \\
M \arrow[r, "\varepsilon"] & U \arrow[r, "\varepsilon'"] & U_0
\end{tikzcd}$$

Fig. 9. An extension of the algebraic compiler model of Thatcher et al.

The model of Thatcher et al. also has algebras $T_0$ and $U_0$ which are similar to each other, but not necessarily similar to the other algebras $L, M, T$ and $U$. These algebras do not appear in their commutative diagrams. We can extend this model with injective mappings $\gamma'$ from $T$ to $T_0$ and $\varepsilon'$ from $U$ to $U_0$ which just map elements from sorts in $T$ and $U$ respectively to the same values in the (different) sorts of $T_0$ and $U_0$. If we also add the mapping $\psi'\colon T_0 \to U_0$ then we have the diagram in Figure 9. We can relate this to the model in Figure 3 by the following equalities: $L = \mathcal{A}_S^{syn}$, $M = \mathcal{A}_S^{sem}$, $T = \mathcal{A}_{ST}^{syn}$, $U = \mathcal{A}_{ST}^{sem}$, $T_0 = \mathcal{A}_T^{syn}$, $U_0 = \mathcal{A}_T^{sem}$, $\gamma = em_{syn}$, $\varepsilon = em_{sem}$, $\gamma' = im_{syn}$, $\varepsilon' = im_{sem}$, $\theta = \mathcal{E}_S$, $\psi = \mathcal{E}_{ST}$ and $\psi' = \mathcal{E}_T$. The compositions $\gamma \circ \gamma'$ and $\varepsilon \circ \varepsilon'$ are, respectively, the generalized homomorphisms $H_{syn}$ and $H_{sem}$ in Rus's model in Figure 2. What we do not find here is any of the language learning relations $\mathcal{L}_S$, $\mathcal{L}_{ST}$ or $\mathcal{L}_T$.

Given this extension to include dissimilar algebras, we can now add a specification language layer between these languages in a manner similar to what was done above. This is shown in Figure 10. The syntax algebra $I_0^{syn}$ and semantics algebra $I_0^{sem}$ are instantiations of the specification language using the target language algebras $T_0$ and $U_0$ and are created in the same manner that we have seen previously. The algebras $I^{syn}$ and $I^{sem}$ are the hybrid algebras which are similar to $L$ and $M$ but whose carrier sets contain elements, respectively, from $I_0^{syn}$ and $I_0^{sem}$. (We can relate these algebras to those in Figure 4 by the following equalities: $I^{syn} = \mathcal{A}_{SML^T}^{syn}$, $I^{sem} = \mathcal{A}_{SML^T}^{sem}$, $I_0^{syn} = \mathcal{A}_{ML^T}^{syn}$ and $I_0^{sem} = \mathcal{A}_{ML^T}^{sem}$.) Recall that we need to execute the computations in $I_0^{sem}$. Thus, we can not simply use the mapping $\gamma \circ \gamma' \circ \gamma''$. This is similar to the need to define $im1_{syn}$ as the composition $\mathcal{E}_{FM} \circ im1_{sem} \circ \mathcal{L}_M$ in Section 5.1. Thus the compiler we can use is $\gamma \circ \gamma' \circ \phi' \circ \varepsilon''$. This maps source language expressions in $L$ to target language meanings in $U_0$, which is almost what we want. A primary difference between the Rus model and that of Thatcher et al. is Rus's model has the language learning relations $\mathcal{L}$, and thus the final printing out of the result falls outside of these other models since we compute the semantic value in $U_0 = A_M^{sem}$ and would like to print this out as a word in $T_0 = A_M^{syn}$.

$$\begin{tikzcd}
L \arrow[r, "\gamma"] \arrow[d, "\theta"] & I^{syn} \arrow[r, "\gamma'"] \arrow[d, "\phi"] & I_0^{syn} \arrow[r, "\gamma''"] \arrow[d, "\phi'"] & T_0 \arrow[d, "\psi'"] \\
M \arrow[r, "\varepsilon"] & I^{sem} \arrow[r, "\varepsilon'"] & I_0^{sem} \arrow[r, "\varepsilon''"] & U_0
\end{tikzcd}$$

Fig. 10. A specification-language in the Thatcher model of an algebraic compiler.

Thus, we see that specification languages can be added to Thatcher's model, but this requires an extension to the model to handle algebras of different similarity and an additional mechanism to replace the target language learning relation to "print" the final result. Since both of these exist *a priori* in the Rus model it is easier to add these extensions there. While there has been some debate in the literature ([35] and [37],page 231) about some of the features of Rus's model, we see that the dissimilar algebras and the language learning relations incorporated in the Rus model are in fact very useful for our purposes and it suggests that because of these added facilities the Rus model is easier to extend.

It is interesting to note that in Thatcher et al. [3] on page 613, the authors hint at such a layering approach when they note that the 'correctness of a composite translation could be obtained by "pasting" commuting squares together.' The intention there was not quite the same as what we have achieved here, however, in that they were anticipating translations through (possibly several) intermediate languages where each intermediate language provided a representation of the source language program in a language progressively closer to the final target language. If these intermediate languages are $I_i, 0 < i < n$, then a compiler $\gamma\colon L \to T$ is the composition $\gamma = \gamma_0 \circ \gamma_1 \circ \ldots \circ \gamma_n$ where $\gamma_0\colon L \to I_1, \gamma_i\colon I_i \to I_{i+1}, 0 < i < n$, and $\gamma_n\colon I_n \to T$. Note that all of these mappings $\gamma_i$ are between the syntax algebras of the languages, not the semantics algebras.

We have also introduced an "intermediate" language in a sense, and even though the commuting diagrams have a similar form, the function of the intermediate language is different. In our case, its semantics, that is, computations in this language, are used to calculate the translation; it is not used solely as an intermediate representation for constructs in the source language. In order to map a source text to a target text, we need to execute operations in the intermediate (what we have called specification) language. That is, we must involve its semantics algebra. Thus, the language learning relation $\mathcal{L}$

in the Rus model is helpful for mapping semantics constructs back to their representation in the syntax algebra.

### *6.1.2 Algebraic construction of compilers*

The algebraic compiler models of Mosses [4], Gaudel [36] and Rus [2], among others, also have notions of correctness similar to that defined by Thatcher et al. but go further by suggesting algorithms and tools for generating compilers from algebraic specifications of the languages and the mappings between them.

In this section we show how a specification language can be used in the model proposed by Mosses [4]. As we will see, his extension ($Tx'$) to the target language ($T$) of his compiler can be seen as a specification language. In this case, the specification language addresses issues which arise when the target language can not express all elements of the source language. That is, the target language does not contains some required elements. Although the translator we are interested in is partial and only maps to expressions in $T$ this, we will see, hinders our specification of the translator as a homomorphism.

Mosses provides a "constructive approach to compiler correctness" by showing how the compiler from the paper by Thatcher et al. can be constructed. Thatcher et al. provide a definition of compiler correctness, as does Mosses, but Mosses also shows how to construct the compiler. Mosses presents the compiler in a slightly different form in which $L$ is the source language, $S$ is a standard semantics and $T$ is the target language. The mappings in Mosses's compiler are shown in the commutative diagram in Figure 11. The semantic algebras $M$ and $U$ from Thatcher et al. are not used in Mosses's proof of correctness and the mappings incident on them are not labeled. The *sem* mapping

$$\begin{tikzcd}
 & L \arrow[dl, "sem"'] \arrow[dr, "comp"] & \\
S \arrow[rr, "impl"] \arrow[d] & & T \arrow[d] \\
M \arrow[rr] & & U
\end{tikzcd}$$

Fig. 11. The algebraic compiler of Mosses.

is a generalized homomorphism which embeds $L$ into a standard semantics $S$ and *impl* maps $S$ into the target $T$. Mosses shows that *impl* is injective and thus correct in the sense that if $s = s'$ for $s, s' \in S$ (with respect to the equations defining equivalent semantics in $S$) then $impl(s) = impl(s')$. Now, given a correct semantics of $L$ in the form of *sem* , a correct compiler *comp* , in the form of a generalized homomorphism, can be constructed by composing *sem* and *impl* . That is, $comp = sem \circ impl$.

It is in the definition of *impl* that we find a possible use of specification languages. Consider an example phrase from $L$, $x := -y$, and its embedding by *sem* in $S$, $\mathit{sem}(x := -y) = \mathit{contents}_y \mathrel{\succ\!\!-} z.(-z)! \mathrel{\succ\!\!-} \mathit{update}_x$. This semantics phrase maps via *impl* to the phrase $\mathit{contents}_y \rightarrow - \rightarrow \mathit{update}_x$ in $T$. Without concerning ourselves with the meaning of the various constructs in $L$, $S$ and $T$ we can still see that the mapping *impl* is not a generalized homomorphism because the free variables ($z$ in the example above) in $S$ can not be represented in $T$. This does not prevent the construction of *comp* as a generalized homomorphism since every phrase from $L$ can be represented in $T$.

To define *impl*, Mosses extends $T$ to $Tx$ and then to $Tx'$ which contains additional operators so that the generalized homomorphism $\mathit{impl}'\colon S \rightarrow Tx'$ can be defined such that our example phrase in $S$ can be mapped to $Tx'$ using a generalized homomorphism. Again without concerning ourselves with the details of $Tx'$ we note that $\mathit{impl}'(\mathit{contents}_y \mathrel{\succ\!\!-} z.(-z)! \mathrel{\succ\!\!-} \mathit{update}_x) = \mathit{contents}_y \rightarrow \mathit{flip}^1 \rightarrow z.(z! \rightarrow -) \rightarrow \mathit{flip}^1 \rightarrow \mathit{update}_x$. $Tx'$ has representations for the free variables in $S$. Mosses treats a set of equivalence defining equations for $Tx'$ as rewrite rules which can be used to rewrite the phrases in $Tx'$ that are images (via $\mathit{sem} \circ \mathit{impl}'$) of phrases in $L$ as phrases in $T$. We will refer to this rewriting as $\mathit{impl}''\colon Tx' \rightarrow T$; note that $\mathit{impl}''$ is a partial map since some phrases in $Tx'$ (those that are not images of constructs in $L$) can not be expressed in $T$. Thus, $\mathit{impl} = \mathit{impl}' \circ \mathit{impl}''$. These rewritten phrases are also phrases in $Tx'$ since $T \subset Tx'$.

Although $Tx'$ is not as independent of the target language $T$ as the specification languages $L_F$ and $L_I$ are of $L_M$ we can still consider $Tx'$ to be a specification language. In our notation, let $L_{Tx'} = \langle \mathcal{A}^{sem}_{Tx'}, \mathcal{A}^{syn}_{Tx'} = Tx', \mathcal{L}_{Tx'} \rangle$. We can define the evaluation mapping $\mathcal{E}_{Tx'}\colon \mathcal{A}^{syn}_{Tx'} \rightarrow \mathcal{A}^{sem}_{Tx'}$ to perform the rewriting done by $\mathit{impl}''$ before mapping to the semantic values in $\mathcal{A}^{sem}_{Tx'}$ and $\mathcal{L}_{Tx'}\colon \mathcal{A}^{sem}_{Tx'} \rightarrow \mathcal{A}^{syn}_{Tx'}$ such that $\forall \alpha \in \mathcal{A}^{sem}_{Tx'}, \mathcal{E}_{Tx'}(\mathcal{L}_{Tx'}(\alpha)) = \alpha$. Defined in this way, $\mathit{impl}'' = \mathcal{E}_{Tx'} \circ \mathcal{L}_{Tx'}$. We also define the other languages $L$, $S$ and $T$ as $\Sigma$-languages as expected ($L_L = \langle \mathcal{A}^{sem}_{L}, \mathcal{A}^{syn}_{L} = L, \mathcal{L}_L \rangle, L_S = \langle \mathcal{A}^{sem}_{S}, \mathcal{A}^{syn}_{S} = S, \mathcal{L}_S \rangle$ and $L_T = \langle \mathcal{A}^{sem}_{T}, \mathcal{A}^{syn}_{T} = T, \mathcal{L}_T \rangle$) and place the mappings defined above into the commutative diagram in Figure 12. We have omitted the intermediate hybrid languages from this diagram which would sit between $L_L$ and $L_S$ and also between $L_S$ and $L_{Tx'}$. (This same omission was made in Figure 2 but not in Figure 3.) We see that as before *sem* embeds $L$ into $S$ and $\mathit{impl}'$ embeds $S$ into $Tx'$. Similarly $\mathit{em}_{sem}$ embeds $\mathcal{A}^{sem}_{L}$ into $\mathcal{A}^{sem}_{S}$ and $\mathit{impl}'_{sem}$ embeds $\mathcal{A}^{sem}_{S}$ into $\mathcal{A}^{sem}_{Tx'}$. The mapping $\mathit{im}_{syn}$ is the partial identity mapping which injects elements of $Tx'$ which also exist in $T$ into $T$. The semantic version $\mathit{im}_{sem}$ is also a partial identity mapping. Since $\mathit{impl}'$ may generate elements of $Tx'$ which are not in $T$ but can be rewritten as elements of $T$ by $\mathit{impl}''$ and $\mathit{impl}'' = \mathcal{E}_{Tx'} \circ \mathcal{L}_{Tx'}$, we can define *impl* as $\mathit{impl} = \mathit{impl}' \circ \mathcal{E}_{Tx'} \circ \mathcal{L}_{Tx'} \circ \mathit{im}_{syn}$ or $\mathit{impl} = \mathit{impl}' \circ \mathcal{E}_{Tx'} \circ \mathit{im}_{sem} \circ \mathcal{L}_T$. Thus, *comp* can be constructed from the composition of the mappings we've seen as $\mathit{comp} = \mathit{sem} \circ \mathit{impl}' \circ\ \mathcal{E}_{Tx'} \circ \mathcal{L}_{Tx'} \circ \mathit{im}_{syn}$

$$\begin{tikzcd}
\mathcal{A}_L^{sem} \arrow[r, "\mathcal{L}_L"] \arrow[d, "sem_{sem}"] & \mathcal{A}_L^{syn} \arrow[r, "\mathcal{E}_L"] \arrow[d, "sem"] & \mathcal{A}_L^{sem} \arrow[d, "sem_{sem}"] \\
\mathcal{A}_S^{sem} \arrow[d, "impl'_{sem}"] & \mathcal{A}_S^{syn} \arrow[l, "\mathcal{E}_S"'] \arrow[d, "impl'"] & \mathcal{A}_S^{sem} \arrow[l, "\mathcal{L}_S"'] \arrow[d, "impl'_{sem}"] \\
\mathcal{A}_{Tx'}^{sem} \arrow[r, "\mathcal{L}_{Tx'}"] \arrow[d, "im_{sem}"] & \mathcal{A}_{Tx'}^{syn} \arrow[r, "\mathcal{E}_{Tx'}"] \arrow[d, "im_{syn}"] & \mathcal{A}_{Tx'}^{sem} \arrow[d, "im_{sem}"] \\
\mathcal{A}_T^{sem} & \mathcal{A}_T^{syn} \arrow[l, "\mathcal{E}_T"'] & \mathcal{A}_T^{sem} \arrow[l, "\mathcal{L}_T"']
\end{tikzcd}$$

Fig. 12. Mosses's compiler with a specification language.

or $comp = sem \circ \ impl' \circ \mathcal{E}_{Tx'} \circ im_{sem} \circ \mathcal{L}_T$.

Of interest here is that we have used a specification language $L_{Tx'}$ to implement the (partial) mapping $impl\colon S \to T$. The deficiency in $T$ is that some elements, the free variables, of $S$ can not be represented in $T$. Even though the elements of $S$ we are interested in mapping to $T$ may contain free variables they do have a representation in $T$, because the free variables are bound in these elements of $S$. The problem is that this prevents the specification of the mapping *impl* as a generalized homomorphism. We thus use the specification language $L_{Tx'}$ to define *impl′* as a generalized homomorphism and its evaluation function $\mathcal{E}_{Tx'}$ to rewrite elements of $Tx'$ into elements of $T$.

### *6.2 Specification languages in other frameworks*

#### *6.2.1 Attribute Grammars.*

Specification languages within attribute grammars have a slightly different form than in algebraic compilers. Algebraic compilers rely on an explicit definition of the target language and use target language operations for writing derived operations. These operations thus provide a starting point for adding specification language features. Attribute grammars, to their detriment, make no explicit mention of the target language and thus do not have a set of target language operations to provide as a starting point for writing semantic functions for defining attribute values. Instead, they provide a single general purpose language for writing semantic functions. This language doesn't suffer the expressiveness problems we saw above, but it does lock the user into a single "specification language" for defining attribute values. We have thus argued [19] that a choice of domain specific specification languages in attribute grammars is also desirable for many of the same reasons as they are beneficial in algebraic compilers.

### *6.2.2 Action Semantics.*

At first glance, the idea of specification languages presented here is reminiscent of *facets* in Action Semantics [38,39]. A facet provides “action combinators” whose focus is on processing at most one kind of information at a time. For example, a *declarative* facet is used for processing scope information and an *imperative* facet is used for processing information such as bindings and values of storage cells. These are thus similar to domain specific specification languages [19] whose goal is to provide a specification language for an algebraic compiler specific to a particular domain of language processing such as type checking or optimization. These were discussed in Section 5.3. Facets however are also very similar to *aspects* from aspect-oriented programming [40] for the domain of language processing. Aspects allow a programmer to “cross cut” the conventional program structures, in this case abstract syntax trees, to specify semantic information for a particular concern, perhaps the “environment”, in one place or module without inter-mixing specifications for other semantic concerns. Thus facets are also similar to the aspects in aspect-oriented compilers [41].

Action semantics and algebraic compilers can be seen as striving to reach the same goal of providing a mechanism for easily specifying provably correct compilers but by beginning from different starting points. Mosses states [39] that the primary design philosophy behind action semantics was to provide a *pragmatic* methodology for specifying language semantics that would scale up to realistic programming languages and avoid many of the problems of denotational semantics. In fact, there are action semantics specifications for Pascal [42] and the Standard ML ‘bare’ language [43]. Palsberg [44–46] has proved the correctness of a compiler generator which he designed and implemented that accepts action semantics descriptions of imperative languages and generates code for an abstract RISC machine. As Mosses states in [39], this may be a “first step” in developing tools which allow one to prove the correctness of a generated compiler which is specified in a notation (action semantics) which is easy to read. Algebraic compilers, however, begin with a philosophy that aims to prove the correctness of compilers. The notion of correctness is usually defined in terms of commuting diagrams [35]. Our specification languages here are aimed to make it easier to write algebraic compilers and hopefully this work is a contribution toward moving algebraic compilers in a more pragmatic direction. Thus the two methodologies could be said to be heading toward the same goal, but from different starting points.

### *6.2.3 Rewriting Logic.*

The rewriting logic system [47] of the Maude [48] language provides very general and powerful mechanisms for implementing logics as well as a semantic framework for specifying languages and systems. Many different models of computation can be unified using rewriting logic. The semantics of functional (specification) languages can be implemented via rewriting [49] in which different functional evaluation strategies, either strict or non-strict (lazy), can be specified by changing the rewriting strategy. Imperative languages can also be implemented via rewriting when the rewrite rules corresponds to state transitions and the rewritten term represents the program's state. Maude also provides a module system for specifying rewrite and equation theories. In the case of algebraic compilers, specification languages could be composed with target languages where both are specified in term of rewriting logics. There are, however, no restrictions on the implementation techniques one chooses for the specification languages in our framework. Thus, we have a bit more freedom in that everything does not have to be specified as a rewriting logic. Nevertheless, it would be an interesting experiment to embed the notions presented here into a rewriting logic framework.

### *6.3 Domain Specific Languages*

Above we mentioned domain specific specification languages and here we discuss how our use of domain specific specification languages compares with the other work on domain specific languages (DSLs) in general. We mention only a few different approaches to DSLs here, as represented by Hudak [50], Swierstra [51] and Neighbors [52].

In [50], Hudak discusses the importance of using domain specific *embedded* languages (DSEL) in building large software systems. In this approach, new domain specific features can be added to a language by providing definitions for these constructs from existing constructs in the language. Similar techniques are also discussed by Swierstra et al. in [51]. Both of these approaches make use of higher order functions in Haskell [32]. These techniques have been used to build embedded domain specific languages for parser generation, animation, table formatting and language processing to mention just a few. We believe these technique provide powerful and convenient mechanisms for raising the level of abstraction in one's programs and have used these ideas in a prototype for an "intentional programming" system [53] briefly mentioned below in Section 7.

Building on existing language features in this manner is certainly possible in our algebraic compiler model as well. We saw an example of this in how the *filter* and *for each* operations for sets were built using existing set operations like *get_one* and *get_rest*. Our main goal, in the realm of algebraic compilers however, is to provide specification language constructs that have a functionality not present in the target language and thus these new constructs can not be built on top of existing target language operators. In the imperative specification language, for example, the *while* operator could not have been implemented on top of the existing target language set operations.

Another use of domain specific languages is found in Draco [52]. Draco is both an approach to software engineering and a prototype implementing this approach that is broader in scope than the DESL techniques discussed above. It envisages a hierarchy of DSLs with general purpose languages at the bottom. High level specifications are written in a DSL appropriate to the task. These specifications are then refined, both manually and automatically, to more concrete DSLs until eventually a program in a general purpose language is generated. In Draco, there are several DSLs and the intent is to use the right one for the right part of the job. This is also our goal. In a proper algebraic language processing environment, one would find several domains specific specification languages for the domains of type checking, program analysis and code generation.

## 7 Conclusion

We have shown in this paper how specification languages can be used to address two types of deficiencies that are possible in target language algebras in the framework of algebraic compilers. In the main model checking example we saw that the target model language algebras $\mathcal{A}_M^{sem}$ and $\mathcal{A}_M^{syn}$ contained as elements the satisfiability sets of all of the CTL formulas in $\mathcal{A}_{ctl}^{syn}$ but that the operations of these target language algebras where insufficient to *compute* the satisfiability sets. The specification languages $L_F$ and $L_I$ were instantiated with $L_M$ to provide a language whose algebras contained the necessary operations to implement the model checker. This allowed us to keep the target language as it was originally defined and choose the types of additional computations (functional or imperative) that we wanted to use to specify the model checker.

We also saw in the discussion of Mosses's algebraic compiler model how specification languages could be used to address another deficiency in target languages: the inability to represent some components or sub expressions of elements of the source language we want to translate. This prevented us from implementing the translator directly as a (generalized) homomorphism. By treating $Tx'$ as a specification language we saw how the specification languages could be used in other constructive models of algebraic compilers.

It is our belief that using specification languages as we have defined them makes algebraic compilers significantly easier to specify. Common problems of target language expressibility, as illustrated by our examples, can be cleanly overcome using specification languages. Instead of extending a target language with additional operations or elements, we can choose to reuse existing specification languages which have the additional components required to specify the translation. This approach allows for a more modular specification of source and target languages and the mappings between them.

We are pursuing this work as part of an effort to find appropriate meta languages to be used for defining language constructs for the Intentional Programming (IP) [54,53,55] system which was until recently under development at Microsoft. IP is an extensible programming environment which allows programmers to define their own language constructs, called *intentions*, and add them to their programming environment. We are interested in exploring different specification languages for defining such intentions. Since the same domains of type checking, optimization, code generation, etc., are encountered in IP, domain specific specification languages will be useful in this system as well. They are especially important here since appropriate domain specific specification languages raise the level of abstraction in which the intention designer works and will thus make designing intentions a more reasonable process that experienced programmers could perform to create their own language extensions.

**Acknowledgments:** We would like to thank the anonymous reviewers for their helpful suggestions in improving this paper and Microsoft Research for funding this research.

## References

[1] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145, corrections in 5(2):95–96, 1971.

[2] T. Rus, Algebraic construction of compilers, Theoretical Computer Science 90 (1991) 271–308.

[3] J. Thatcher, E. Wagner, J. Wright, More on advice on structuring compilers and proving them correct, in: H. Maurer (Ed.), Automata, Languages and Programming Proceedings, ICALP'79, Vol. 71 of Lecture Notes in Computer Science, Springer-Verlag, 1979, pp. 596–615.

[4] P. Mosses, A constructive approach to compiler correctness, in: N. Jones (Ed.), Proc. Semantics-directed Compiler Generation Workshop, Vol. 94 of Lecture Notes in Computer Science, Springer-Verlag, 1980, pp. 189–210.

[5] P. Mosses, A constructive approach to compiler correctness, in: Proc. International Conference on Automata, Languages and Programming, Vol. 85 of Lecture Notes in Computer Science, Springer-Verlag, 1980, pp. 449–469.

[6] E. Clarke, E. Emerson, A. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM TOPLAS 8 (2) (1986) 244–263.

[7] B. Steffen, Generating data flow analysis algorithms from modal specifications, Science of Computer Programming 21 (1993) 115–139.

[8] T. Rus, E. Van Wyk, Using model checking in a parallelizing compiler, Parallel Processing Letters 8 (4) (1998) 459–471.

[9] D. Lacey, O. de Moor, Imperative program transformation by rewriting, in: Proc. 10th International Conf. on Compiler Construction, Vol. 1113 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 52–68.

[10] T. Rus, E. Van Wyk, T. Halverson, Integrating temporal logics and model checking algorithms, Formal Methds in System Design 20 (3) (2002) 249–284.

[11] E. Clarke, O. Grumberg, D. Peled, Model Checking, M.I.T. Press, 1999.

[12] S. Kripke, Semantical analysis of modal logic i: Normal modal propositional calculi, Zeitschrift f. Math. Logik und Grundlagen d. Math. 9.

[13] A. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.

[14] P. Cohn, Universal Algebra, Reidel, London, 1981.

[15] T. Rus, Algebraic processing of programming languages, Theoretical Computer Science 199 (1) (1998) 105–143.

[16] T. Rus, E. Van Wyk, Algebraic implementation of model checking algorithms, in: Third AMAST Workshop on Real-Time Systems, Proceedings, 1996, pp. 267–279.

[17] P. Higgins, Algebras with a scheme of operators, Mathematische Nachrichten 27 (1963/64) 115–132.

[18] E. Van Wyk, Semantic processing by macro processors, Ph.D. thesis, The University of Iowa, Iowa City, Iowa, 52240 USA (July 1998).

[19] E. Van Wyk, Domain specific meta languages, in: ACM Symposium on Applied Computing, Vol. 2, Association of Computing Machinery, Como, Italy, 2000, pp. 799–803.

[20] E. Van Wyk, Meta languages in algebraic compilers, in: T. Rus (Ed.), Eighth International Conference on Algebraic Methodology and Software Technology, AMAST 2000, Proceedings, Vol. 1816 of Lecture Notes in Computer Science, Iowa City, Iowa, USA, 2000, pp. 119–134.

[21] M. McIlroy, Macro instruction extensions of compiler languages, Communications of the A.C.M. 3 (4) (1960) 214–220.

[22] B. Leavenworth, Syntax macros and extended translations, Communications of the ACM 9 (11) (1966) 790–793.

[23] T. J. Cheatham, The introduction of definitional facilities into higher level programming languages, in: AFIPS (Fall Joint Computer Conference, 29), 1966, pp. 623–637.

[24] D. Weise, R. Crew, Programmable syntax macros, ACM SIGPLAN Notices 28 (6).

[25] W. Maddox, Semantically-sensitive macroprocessing, Master's thesis, The University of California at Berkeley, Computer Science Division (EECS), Berkeley, CA 94720 (December 1989).

[26] J. Lee, Macro-processors as compiler code generators, Master's thesis, The University of Iowa, Iowa City, Iowa (1990).

[27] J. Knaack, An algebraic approach to language translation, Ph.D. thesis, The University of Iowa, Department of Computer Science, Iowa City, IA 52242 (December 1994).

[28] T. Rus, T. Halverson, E. Van Wyk, R. Kooima, An algebraic language processing environment, in: M. Johnson (Ed.), Sixth International Conference on Algebraic Methodology and Software Technology, AMAST '97, Proceedings, Vol. 1349 of Lecture Notes in Computer Science, Sydney, Australia, 1997, pp. 581–585.

[29] T. L. Halverson, Language development by component based tools, Ph.D. thesis, The University of Iowa, Iowa City, Iowa, 52240 USA (May 1999).

[30] R. Backhouse, P. Jansson, J. Jeuring, L. Meertens, Generic programming — an introduction, in: Proc. Third International Summer School on Advanced Functional Programming, Vol. 1608 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 28–115.

[31] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Java Series, Sun Microsystems, 1996.

[32] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, P. Wadler, Haskell 98, available at URL: `http://www.haskell.org` (February 1999).

[33] T. Rus, E. Van Wyk, Integrating temporal logics and model checking algorithms, in: Fourth AMAST Workshop on Real-Time Systems, Proceedings, Vol. 1231 of Lecture Notes in Compter Science, Springer-Verlag, 1997, pp. 95–110.

[34] F. Morris, Advice on structuring compilers and proving them correct, in: Proc. ACM Symposium on Principles of Programming Languages, Association of Computing Machinery, 1973, pp. 144–152.

[35] T. Janssen, Algebraic translations, correctness and algebraic compiler construction, Theoretical Computer Science 199 (1998) 25–56.

[36] M. Gaudel, Correctness proof of programming language translations, in: Formal Description of Programming Concepts - II, North-Holland, 1982, pp. 25–43.

[37] T. Rus, Algebraic definition of programming languages, in: Proc. AMiLP2000, Algebraic Methods in Language Processing, 2nd AMAST Workshop on Language Processing, 2000, pp. 223–232.

[38] P. Mosses, Action Semantics, no. 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[39] P. Mosses, Theory and practice of action semantics, in: Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Vol. 1113 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 37–61.

[40] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Aksit, S. Matsuoka (Eds.), ECOOP'97 Object–Oriented Programming, Vol. 1241 of Lecture Notes in Computer Science, 1997, pp. 220–242.

[41] O. de Moor, S. Peyton-Jones, E. Van Wyk, Aspect oriented compilers, in: K. Czarnacki, U. Eisenecker (Eds.), First International Symposium on Generative and Component-Based Software Engineering, Vol. 1799 of Lecture Notes in Computer Science, 1999, pp. 121–133.

[42] P. Mosses, D. Watt, Pascal action semantics, version 0.6, available at URL: `ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/MossesWattDRAFT.ps.Z` (Mar 1993).

[43] D. Watt, An action semantics of standard ml, in: Proc. Third Workshop on Math. Foundations of Programming Language Semantics, Vol. 298 of Lecture Notes in Computer Science, Springer-Verlag, 1988, pp. 572–598.

[44] J. Palsberg, Provably correct compiler generation, Ph.D. thesis, University of Aarhus (1992).

[45] J. Palsberg, An automatically generated and provably correct compiler for a subset of Ada., in: Proc. Fourth IEEE Int. Conf on Computer Languages, IEEE, 1992, pp. 117–126.

[46] J. Palsberg, A provably correct compiler generator, in: ESOP'92, Proc. European Symposium on Programming Languages, Vol. 582 of Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 418–434.

[47] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, Tech. Rep. SRI-CSL-93-05, SRI International, Computer Science Laboratory, to appear in D. Gabbay, editor, *Handbook of Philosophical Logic, Second Edition, Volume 6*, Kluwer Academic Publishers, 2001 (Aug. 1993).

[48] J. Meseguer, Rewriting logic and Maude: Concepts and applications, in: L. Bachmair (Ed.), Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10–12, 2000, Proceedings, Vol. 1833 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 1–26.

[49] S. Peyton-Jones, The implementation of functional programming languages, Prentice Hall, 1992.

[50] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys 28 (4es).

[51] S. Swierstra, P. Alcocer, J. Saraiva, Designing and implementing combinator languages, in: Proc. Third International Summer School on Advanced Functional Programming, Vol. 1608 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 150–206.

[52] J. Neighbors, The Draco approach to constructing software from reuasable components, IEEE Transactions on Software Engineering SE-10 (5) (1984) 564–574.

[53] E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, P. Kwiatkowski, Intentional programming: a host of language features, Tech. Rep. PRG-RR-01-21, Computing Laboratory, University of Oxford (2001).

[54] C. Simonyi, Intentional programming: Innovation in the legacy age, presented at IFIP Working group 2.1 workshop. (1996).

[55] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: Proc. 11th International Conf. on Compiler Construction, Lecture Notes in Computer Science, Springer-Verlag, 2002, To appear.