Toward Compiler Implementation Correctness Proofs

LAURIAN M. CHIRICA
California Polytechnic State University
and
DAVID F. MARTIN
University of California, Los Angeles

An aspect of the interaction between compiler theory and practice is addressed. Presented is a technique for the syntax-directed specification of compilers together with a method for proving the correctness of their parse-driven implementations. The subject matter is presented in an order-algebraic framework; while not strictly necessary, this approach imposes beneficial structure and modularity on the resulting specifications and implementation correctness proofs. Compilers are specified using an order-algebraic definition of attribute grammars. A practical class of compiler implementations is considered, consisting of those driven by LR(k) or LL(k) parsers which cause a sequence of translation routine activations to modify a suitably initialized collection of data structures (called a translation environment). The implementation correctness criterion consists of appropriately comparing, for each source program, the corresponding object program (contained in the final translation environment) produced by the compiler implementation to the object program dictated by the compiler specification. Provided that suitable intermediate assertions (called *translation invariants*) are supplied, the program consisting of the (parse-induced) sequence of translation routine activations can be proven partially correct via standard inductive assertion methods.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.4 [**Programming Languages**]: Processors—*compilers, parsing*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, pre- and post conditions, specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics, denotational semantics*

General Terms: Languages, Verification

Additional Key Words and Phrases: Attribute grammar, compiler correctness, compiler implementation, compiler specification, program correctness, program specification

1. INTRODUCTION

Many problems of proving the correctness of an implementation with respect to a specification can be reduced to proving the commutativity of the kind of diagram in Figure 1. The function *implement* maps a specification into its corresponding concrete realization, and *compare* is a function that suitably

This research was sponsored in part by NSF grant MCS-78-18918.
Authors' addresses: L. M. Chirica, Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407; D.F. Martin, Computer Science Department, University of California, Los Angeles, CA 90024.

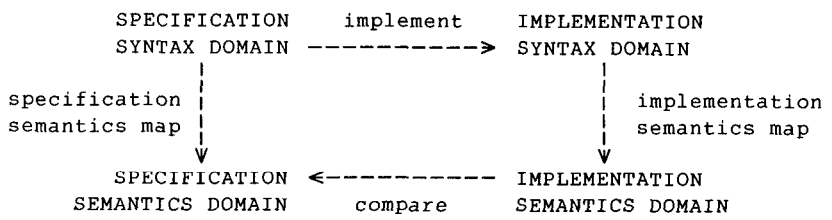

Fig. 1. Implementation correctness.

compares the semantic representations of corresponding specifications and implementations. In many computer science applications the domains at the corners of the diagram are many-sorted algebras [8] with the same signature Σ, and the arrows are Σ-homomorphisms. *SPECIFICATION-SYNTAX-DOMAIN* is an initial Σ-algebra, and thus *implement* and *specification-semantics-map* are uniquely determined [8]. Commutativity of the diagram is then guaranteed by finding homomorhisms *implementation-semantics-map* and *compare*.

Two important instances of (the algebraic form of) the diagram in Figure 1 that arise in the context of the compiler correctness problem are shown in Figures 2 and 3. Figure 2 represents the correctness of a compiler *specification* by comparing the semantics of corresponding source and object programs; see [5, 6, 15–18, 19]. The commutativity of this kind of diagram represents what is most often called "compiler correctness" in the literature. Here a compiler specification is a *function* that specifies the object program corresponding to each source program; this function says *what* the compiler is to do, but not *how* it is to be implemented. More complete discussions, with examples, of the algebraic approach to the compiler specification correctness problem can be found in [6] and [19].

On the other hand, Figure 3 represents the correctness of a compiler *implementation* by comparing corresponding object programs generated by the compiler specification and implementation; cf. [6, 18]. This facet of compiler correctness is almost never treated in the literature. Here the actual translation *program* that is executed under the control of the syntax analysis of each source program is given; this is a model of *how* the compiler is implemented to generate object code. The composition of Figures 2 and 3 represents the relationship between the two major tasks, specification and implementation, that compiler writers typically encounter; these equally important tasks represent the compiler construction problem at different levels of abstraction.

In this paper we focus exclusively on the compiler implementation correctness problem as represented in Figure 3. Our goal is to provide a systematic methodology for compiler specification, together with a corresponding characterization of a large and practical class of compiler implementations, toward the development of a criterion for proving the correctness (with respect to the specification) of the programs that comprise these compiler implementations.

Our approach is *analytical*; that is, given a compiler specification and implementation, we develop a means to determine whether the implementation is

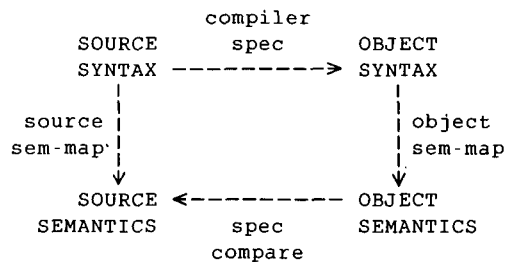

Fig. 2. Compiler specification correctness.

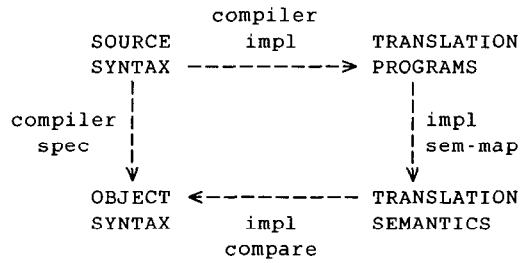

Fig. 3. Compiler implementation correctness.

correct with respect to the specification. This is in contrast to a *synthetic* approach, in which a correct compiler implementation is automatically generated from a given compiler specification; in this case only the implementation generator need be proven correct. If the implementation language's control structures and data types are sufficiently similar to those of the compiler specification, then such automation might be possible. If not, as is generally the case, the prospects for automation are much more remote. The current practical state of the art is the simultaneous development of an implementation and its correctness proof from a specification, a process that is largely analytical.

Research similar to ours has been done by Polak [18]. A brief comparison of our work and his appears in the Conclusion (Section 8).

This paper is organized as follows. Notation and definitions are given in Section 2. An order-algebraic method for compiler specification is given in Section 3. In Section 4 we present a treatment of a practical class of parse-driven compiler implementations and their semantic characterization. A compiler implementation correctness criterion and proof methodology are given in Section 5, culminating in the MAIN THEOREM. A preliminary relational formulation evolves into one that uses "surface semantics" [12] as represented by partial correctness assertions [10], so that conventional (and even mechanized) partial correctness proofs can be performed. Example translation routines and translation invariants are given in Section 6, in order to give concrete substance to the concepts of Sections 4 and 5. Example proofs that illustrate our techniques and results are given in Section 7, followed by discussion and conclusions in Section 8.

2. NOTATION AND DEFINITIONS

If x and y are sequences, xy and $x \bullet y$ denote their concatenation; depending on context, both ϵ and $\langle\ \rangle$ denote the empty sequence. If $G = \langle V_N, V_T, P, Z \rangle$ is a context-free grammar (CFG), we assume a set $\{\pi_p \mid p \in P\}$ of *production labels* disjoint from any other symbols we use. To avoid cumbersome indices on the π_p, it is convenient to arbitrarily number the productions 1, 2, ... and to use the production labels $\pi_1, \pi_2, \ldots$. It is also convenient to permit an infinite number of productions denoted by production schemes (with corresponding production labels) such as $\pi_{[x]} = \mathrm{id} \rightarrow x$, $x \in \mathrm{Id}$; this is particularly useful for displaying the interface between the CFG G and "offline" syntactic processes such as lexical analysis which recognize infinite token classes such as identifiers (illustrated here) and numeric constants. The pth production of G is written $\pi_p = A \rightarrow a_0 B_1 a_1 \ldots B_r a_r$, where π_p is its label, $A, B_1, \ldots, B_r \in V_N$, and $a_0, \ldots, a_r \in V_T^*$. In general r depends upon p, but we omit denoting this dependency when no confusion results. $L(G)$ denotes the language generated by G. In an attribute grammar, $\alpha.A$ denotes attribute α of (nonterminal) A; terminal symbols are not permitted to have attributes.

Let J be an index set. A *J-indexed family* of sets A is denoted by $\{A_j\}_{j \in J}$ or simply $\{A_j\}$ when J is understood. J-indexed families of functions and relations (between J-indexed families of sets) are denoted $\{f_j: A_j \rightarrow B_j\}$ and $\{R_j \subseteq A_j \times B_j\}$. Operations on J-indexed families distribute over the families (e.g., $A \times B = \{A_j \times B_j\}$, and $A \subseteq B$ iff for each $j \in J$, $A_j \subseteq B_j$. When $R \subseteq A \times B$ is an indexed family of (binary) relations, $\langle x, y \rangle \in R$ is often written $R(x, y)$. If $w = j_1 \ldots j_n \in J^*$, A^w abbreviates $A_{j_1} \times \cdots \times A_{j_n}$; $A^\epsilon = \{\epsilon\}$. If $x = \langle x_1, \ldots, x_n \rangle \in A^w$ and $f: A \rightarrow B$, then $f^w(x)$ abbreviates $\langle f_{j_1}(x_1), \ldots, f_{j_n}(x_n) \rangle$, and similarly for relations.

Definitions

Let S be any set, called a set of *sorts*. An *S-sorted signature* is a $(S^* \times S)$-indexed family $\{\Sigma_{w,s}\}$ of sets of *operator symbols*. If Σ is an S-sorted signature, a *Σ-algebra* A consists of an S-indexed family $\{A_s\}$ of sets (called the *carriers of A*), and for each $\sigma \in \Sigma_{w,s}$, a function $\sigma_A: A^w \rightarrow A_s$ (called an *operation* of A). When $w = \epsilon$, $\sigma_A \in A_s$ will as usual be regarded as a constant (of sort s). In the following definitions, A and B are Σ-algebras. B is a *subalgebra* of A if $B \subseteq A$ and for each $\sigma \in \Sigma_{w,s}$, $y \in B^w$ implies $\sigma_A(y) \in B_s$ (i.e., the carriers of B are closed under the operations of A). The *product algebra* $C = A \times B$ of A and B is a Σ-algebra with carriers $A \times B$, and for each $\sigma \in \Sigma_{w,s}$, operations $\sigma_C: A^w \times B^w \rightarrow A_s \times B_s$ such that $\sigma_C(x, y) = \langle \sigma_A(x), \sigma_B(y) \rangle$.

$h: A \rightarrow B$ is a *Σ-homomorphism* if for all $\langle w, s \rangle \in (S^* \times S)$, $\sigma \in \Sigma_{w,s}$, $x \in A^w$: $h_s(\sigma_A(x)) = \sigma_B(h^w(x))$. $R \subseteq A \times B$ is a *weak Σ-homomorphism* from A to B if R is a subalgebra of $A \times B$. Equivalently, R is a weak homomorphism from A to B if for all $\langle w, s \rangle \in S^* \times S$, $\sigma \in \Sigma_{w,s}$, $\langle x, y \rangle \in A^w \times B^w$: $R^w(x, y) \supset R_s(\sigma_A(x), \sigma_B(y))$. Homomorphisms (weak homomorphisms) are viewed as "structure-preserving" functions (relations). A weak Σ-homomorphism R is a Σ-homomorphism iff all the relations in R are functions. The composition of (weak) Σ-homomorphisms is a (weak) Σ-homomorphism. A is *initial* in the class of Σ-algebras if for any Σ-algebra B there exists a unique homomorphism from A to B. For each signature Σ, there exists an initial Σ-algebra T_Σ.

Given a CFG $G = \langle V_N, V_T, P, Z \rangle$, we define a V_N-sorted signature G using G's productions via $G_{w,A} = \{\pi_p \mid \pi_p = A \rightarrow x \in P \text{ and } w = \text{nt}(x)\}$, where $\text{nt}(x)$ denotes the embedded string of nonterminals in x [7, 8]. The initial G-algebra T_G is the usual word algebra [8], whose carriers $T_{G,A}$ can be viewed as representing A-rooted derivation trees on (grammar) G, and whose operations π_p: $T_G^w \rightarrow T_{G,A}$ (where $\pi_p \in G_{w,A}$) can be viewed as derivation tree constructors. Note that we ambiguously denoted the $(\pi_p)_{T_G}$ of T_G by the corresponding operation symbol π_p; this causes no harm in the sequel.

3. COMPILER SPECIFICATION

In this section we indicate the nature of and briefly illustrate the compiler specification technique used in our method for proving the correctness of compiler implementations. We adopt attribute grammars (AGs) [11] as a means of compiler specification. This is a reasonable choice, since AGs have been and are still being used as the basis for several compiler-generator systems (e.g., [3]). Moreover, since there is wide latitude in choosing the attribute value domains of an AG, the resulting compiler specification can be as abstract or as concrete as desired.

For typographical brevity, we adopt the following abbreviations for the components of Figure 3 in the remainder of this paper:

T_G = SOURCE-SYNTAX	c-spc = compiler-spec
O-PROG = OBJECT-SYNTAX	c-imp = compiler-impl
T-PROG = TRANSLATION-PROGRAMS	**D** = impl-sem-map
$[U \rightarrow U]$ = TRANSLATION-SEMANTICS	R = impl-compare

Our contribution to compiler specification and to the role it plays in proofs is to convert an AG compiler specification into algebraic form by using the order-algebraic definition of AGs given in [7]. The result is a G-algebra O-PROG with operations θ_p (one per production). Once this object code algebra O-PROG is obtained, the compiler specification c-spc: $T_G \rightarrow$ O-PROG (a G-homomorphism) is uniquely determined: if

$$\pi_p = A \rightarrow a_0 B_1 a_1 \ldots B_r a_r, \quad r \geq 0$$

is the pth production, then

$$\text{c-spc}_A(\pi_p(t_1, \ldots, t_r)) = \theta_p(\text{c-spc}_{B_1}(t_1), \ldots, \text{c-spc}_{B_r}(t_r)). \tag{1}$$

Alternatively, an algebraic compiler specification can be written directly, as a denotational semantics [9]. Whichever technique is employed is clearly a matter of taste; for large specifications it is often helpful to first write the specification as an AG (using both inherited and synthesized attributes), followed by its straightforward conversion to (purely synthesized) algebraic form. The remainder of this section consists of the presentation and discussion of examples intended to illustrate the specification technique.

Suppose it is desired to specify a compiler that translates programs in a block-structured source language into object programs which are sequences of instructions that contain absolute addresses and are loaded into absolute locations 1, 2, ..., k. We write an AG whose underlying grammar is LR(1), that is, a grammar intended to be used for parsing. Four representative fragments of this

AG are given here; the complete AG and its algebraic counterpart are given in Appendix A.

The AG specified by our example translator generates object programs (sequences of target machine instructions) from a domain O-CODE. The carriers of the algebra O-PROG form a V_N-indexed family of sets, in which $\text{O-PROG}_{\text{prog}} = \text{O-CODE}$, where "prog" is the distinguished (start) symbol of G. Our example translator uses rudimentary symbol tables from a domain $E = \text{Id}^*$, where Id is the domain of identifiers. The position of an identifier in a symbol table $e \in E$ is the relative location allocated to that identifier in the target machine data storage. The domain E also has associated functions that concatenate symbol tables (•), give the length of a symbol table (lg), give the position of a given identifier in a given symbol table (loc), and indicate whether or not a given identifier is present in a given symbol table (found). The following attributes are used in our example AG:

Synthesized Attributes

z: O-CODE; object code;
d: E; local symbol table defined by a declaration;
w: Id; an identifier;
v: Z; (integer) value of a constant.

Inherited Attributes

n: N; starting location of object code in program storage;
e: E; global symbol table (defined by global declarations).

The first AG fragment defines the translation of programs:

$$\begin{aligned} \pi_1 = \text{prog} \rightarrow \text{block} \qquad & n.\text{block} = 2 \\ & e.\text{block} = \langle\rangle \\ & z.\text{prog} = \langle(\text{START})\rangle \bullet z.\text{block} \bullet \langle(\text{HALT})\rangle \end{aligned}$$

Syntactically, a source program is a block; n.block is the first (absolute) location in which the object code for "block" is to be stored, and e.block is the symbol table with respect to which "block" is to be translated. (START) and (HALT) are object instructions. Intuitively, this AG fragment specifies that the object program corresponding to "prog" consists of a (START) instruction followed by the object code corresponding to "block", and ends with a (HALT) instruction. Further, n.block = 2 specifies that the first program location of the block's object code is location 2 (the (START) instruction will occupy location 1), and e.block = $\langle\rangle$ specifies that because the block is outermost, its translation "sees" no global declarations and thus the symbol table is empty.

This AG fragment contributes to the definition of the G-homomorphism c-spc: $T_G \rightarrow$ O-PROG via

$$\text{c-spc}_{\text{prog}}(\pi_1(t)) = \theta_1(\text{c-spc}_{\text{block}}(t)), \tag{1.1}$$

where

$$\theta_1(f) = \langle(\text{START})\rangle \bullet f(2, \langle\rangle) \bullet \langle(\text{HALT})\rangle,$$

f: $N \times E \rightarrow$ O-CODE is a function such that $f(n, e)$ is the block's object code (beginning at location n and translated with respect to symbol table e), and $\pi_1(t)$

and t are program and block derivation trees. The functions θ_p are the operations of the G-algebra O-PROG. Details of this conversion from AG to algebraic specification are given in [7].

The second AG fragment deals with the addition of another declaration to the local symbol table associated with the head of a block:

$$\pi_4 = \text{dcl} \rightarrow \text{dcl}_1, \text{id}$$
$$d.\text{dcl} = \textbf{if found } (w.\text{id}, d.\text{dcl}_1) \textbf{ then } d.\text{dcl}_1 \textbf{ else } d.\text{dcl}_1 \bullet \langle w.\text{id} \rangle$$
$$z.\text{dcl} = \textbf{if found } (w.\text{id}, d.\text{dcl}_1) \textbf{ then } z.\text{dcl}_1 \textbf{ else } z.\text{dcl}_1 \bullet \langle (\text{NEW}) \rangle$$

This fragment produces a pair of items: a new local symbol table (d.dcl) and augmented object code (z.dcl). If the newly declared identifier (w.id) is found in the local symbol table ($d.\text{dcl}_1$) formed thus far, then this new declaration is ignored; otherwise the new identifier is added to (the end of) the local symbol table and a (NEW) instruction is emitted onto the end of the object program fragment generated thus far for the declarations local to this block ($z.\text{dcl}_1$). The (NEW) instruction performs run-time storage allocation. Clearly, a policy of "forgiving" duplicate local declarations has been adopted here; it is just as easy to regard this as an error (see the fourth example fragment below).

This fragment contributes to c-spc via

$$\text{c-spc}_{\text{dcl}}(\pi_4(t_1, t_2)) = \theta_4(\text{c-spc}_{\text{dcl}}(t_1), \text{c-spc}_{\text{id}}(t_2)), \tag{1.4}$$

where

$$\theta_4(\langle d, z \rangle, w) = \textbf{if found}(w, d)\textbf{then } \langle d, z \rangle \textbf{ else } \langle d \bullet \langle w \rangle, z \bullet \langle (\text{NEW}) \rangle \rangle.$$

The predicate "found" is defined by

$$\text{found}(x, d) \leftrightarrow \text{loc}(x, d) > 0,$$

where loc: $\text{Id} \times E \rightarrow N$ is characterized by

$$\text{loc}(x, \langle\rangle) = 0,$$
$$\text{loc}(x, \langle x_1 \rangle) = 1 \text{ iff } x \text{ is the same identifier as } x_1,$$
$$\text{loc}(x, d \bullet e) = \textbf{if } \text{loc}(x, e) = 0 \textbf{ then } \text{loc}(x, d) \textbf{ else } (\text{lg}(d) + \text{loc}(x, e)).$$

The third example AG fragment deals with the generation of internal transfers of control required to translate iteration statements:

$$\pi_{10} = \text{stm} \rightarrow \text{test DO body END}$$
$$n.\text{test} = n.\text{stm}$$
$$e.\text{test} = e.\text{stm}$$
$$n.\text{body} = n.\text{stm} + \text{lg}(z.\text{test}) + 1$$
$$e.\text{body} = e.\text{stm}$$
$$z.\text{stm} = z.\text{test} \bullet \langle (\text{JPF}, k_2) \rangle \bullet z.\text{body} \bullet \langle (\text{JMP}, n.\text{stm}) \rangle$$
$$\textbf{where } k_2 = n.\text{stm} + \text{lg}(z.\text{test}) + \text{lg}(z.\text{body}) + 2.$$

In the above, the object code for an iteration statement consists of the code for its test expression followed by a conditional jump (if false) to the first instruction (at address k_2) following this iteration statement, followed by the code for its body, and finally an unconditional jump back to the first instruction (at address n.stm) of this statement. Since the conditional jump is "forward", our (one-pass)

implementation will be required to "backpatch" its branch address k_2. The function "lg" computes the length (in machine instructions) of an object code sequence.

This fragment contributes to c-spc via

$$\text{c-spc}_{\text{stm}}(\pi_{10}(t_1, t_2)) = \theta_{10}(\text{c-spc}_{\text{test}}(t_1), \text{c-spc}_{\text{body}}(t_2)), \tag{1.10}$$

where

$$\theta_{10}(f, g)(n, e) = f(n, e) \bullet \langle(\text{JPF}, k_2)\rangle \bullet g(k_1, e) \bullet \langle(\text{JMP}, n)\rangle$$

and

$$k_1 = n + \text{lg}(f(n, e)) + 1$$
$$k_2 = k_1 + \text{lg}(g(k_1, e)) + 1.$$

The fourth (and final) example AG fragment deals with expressions that are simply identifiers:

π_{20} = pri → id
  z.pri = **if** found(w.id, e.pri) **then** $\langle(\text{LD}, \text{loc}(w.\text{id}, e.\text{pri}))\rangle$
  else $\langle(\text{ERR}, 20)\rangle$.

If the identifier is found in the current symbol table, then a load instruction containing its address in data storage is emitted; otherwise, an (compile-time) error has occurred, and an "error instruction" (which contains an error code) is emitted. This manner of dealing with compile-time errors is unusual, but provides a simple means of recovering from errors, and defers reporting them until run time. Other methods of recovering from and reporting compile-time errors are certainly possible; we adopt this one for simple illustrative purposes.

This fragment contributes to c-spc via

$$\text{c-spc}_{\text{pri}}(\pi_{20}(t)) = \theta_{20}(\text{c-spc}_{\text{id}}(t)), \tag{1.20}$$

where

$$\theta_{20}(w)(n, e) = \textbf{if found}(w, e) \textbf{ then } \langle(\text{LD}, \text{loc}(w, e))\rangle \textbf{ else } \langle(\text{ERR}, 20)\rangle.$$

4. COMPILER IMPLEMENTATION

A compiler implementation must accurately carry out the translation defined by a specification, like those described in Section 3. In this section we formalize a particular, but quite general and practical, class of syntax-directed compiler implementations. These compiler implementations are those driven by deterministic SHIFT-REDUCE (S-R) parsers [1] (e.g., LR(k) parsers) that perform translation (code generation) operations interleaved with parsing operations. Such compiler implementations are called *postfix translators* [1] because there is exactly one translation operation c_p associated with each production π_p, and c_p is performed when and only when the entire right part of π_p is recognized during parsing, just prior to the corresponding REDUCE parsing operation. The c_p, called *translation routines* (TRs), are programs in some language L (not necessarily the same as the source language being compiled). Each TR $c_p(\mathbf{u})$ modifies the same vector of (global) variables $\mathbf{u}$, whose value ranges over a domain U. The variables in $\mathbf{u}$ are the data structures that comprise a *translation environment*

(TE) which contains not only an object code array, but also auxiliary structures such as a symbol table, translation stack, etc. Many compilers have been implemented this way, including the EULER [20], XPL [14], and PLM [13] compilers, to mention a few. Specific examples of TRs in our compiler will be given in Section 6.

S-R parsers deduce the canonical (reversed rightmost) derivation associated with each syntactically valid input. Such a derivation is uniquely represented by the string of production labels corresponding to the sequence of REDUCE operations (and hence TR activations) performed during the parse of that input; this is true of an input derived from any single nonterminal of the grammar. If $\pi_a \pi_b \cdots \pi_z$ represents a canonical derivation, then $c_a; c_b; \ldots; c_z$ is the corresponding L-program (a TR sequence) executed during the parse, where ";" denotes L-program composition. After the TE is suitably initialized, this L-program is executed, one TR per REDUCE, under the direction of the S-R parser. When parsing is complete, the final value of the TE contains the object code corresponding to the parsed source program. In practice, the final TE can supply other items as well, such as a symbol table to be used by a symbolic debugging system.

T-PROG is a G-algebra of such TR sequences, with operations μ_p for each production π_p. The G-homomorphism c-imp: $T_G \rightarrow$ T-PROG is (uniquely) determined from

$$\text{c-imp}_A(\pi_p(t_1, \ldots, t_r)) = \mu_p(\text{c-imp}_{B_1}(t_1), \ldots, \text{c-imp}_{B_r}(t_r)), \tag{2}$$

where

$$\mu_p(d_1, \ldots, d_r) = d_1; \ldots; d_r; c_p.$$

If $t \in T_{G,A}$ is a source program fragment, then c-imp$_A(t)$ is the TR sequence executed under the direction of the S-R parse of t.

Let **D** be a denotational semantics [9] for L such that the semantics of an L-program is a continuous function from U to U. $[U \rightarrow U]$ is easily made into a G-algebra with carriers $[U \rightarrow U]_A = [U \rightarrow U]$ for all $A \in V_N$ and operations ξ_p. **D** is extended to a G-homomorphism **D**: T-PROG $\rightarrow [U \rightarrow U]$ via

$$\mathbf{D}_A[\![\mu_p(d_1, \ldots, d_r)]\!] = \xi_p(\mathbf{D}_{B_1}[\![d_1]\!], \ldots, \mathbf{D}_{B_r}[\![d_r]\!]), \tag{3}$$

where

$$\xi_p(g_1, \ldots, g_r) = F_p \circ g_r \circ \cdots \circ g_1,$$

F_p: $U \rightarrow U$ is the denotational semantics of TR c_p, and "$\circ$" denotes function composition.

The above compiler implementation model can still be used even if the underlying grammars are LL(k) (and thus use top-down parsing) with TR activations occurring anywhere in the right part of productions, not just at their ends. This is accomplished by using the idea of an *output-simulated input grammar* [4], in which special nonterminal "markers" are inserted into the right parts of productions at points (other than just at the end) where TRs are to be called. See [4] or [1] (Section 7.11) for more complete discussions of this technique.

5. IMPLEMENTATION CORRECTNESS CRITERION

The correctness of a compiler implementation c-imp with respect to a compiler specification c-spc is expressed by the commutativity of Figure 3. Having obtained the G-homomorphisms c-spc, c-imp, and **D** of Sections 3 and 4, the commutativity of Figure 3 is guaranteed (by the initiality of T_G) if a G-homomorphism R: $[U \rightarrow U] \rightarrow$ O-PROG can be found.

The correctness problem can be simplified by noting that it is sufficient to find a homomorphism c-gen: T-PROG $\rightarrow$ O-PROG which bypasses the lower right portion of Figure 3, in particular the semantics function **D**. In fact, since our primary concern is that the specification of the compilation of entire source programs (as opposed to program fragments) be correctly implemented, it is sufficient that c-gen be a *weak* homomorphism such that $\text{c-gen}_{\text{prog}}$ is a *function*. We shall indeed adopt this criterion; it is often more convenient (and sometimes necessary) to have the c-gen_A, $A \neq \text{prog}$, be relations instead of functions. This is an example of using weaker, but more convenient, auxiliary (structural) inductive hypotheses that support the proof of a stronger goal (that requires $\text{c-gen}_{\text{prog}}$ to be a function), namely for all $t \in T_{G,\text{prog}}$, $\text{c-gen}_{\text{prog}}(\text{c-imp}_{\text{prog}}(t)) = \text{c-spc}_{\text{prog}}(t)$.

Each c-gen_A expresses an invariant condition that must hold whenever an A-tree $t \in T_{G,A}$ is translated. Basically, c-gen_A must at least state that the L-program $\text{c-imp}_A(t)$, executed while parsing the source program corresponding to t, must produce in the TE the same object program $\text{c-spc}_A(t)$ required by the compiler specification; c-gen_A must in general make other specifications as well. For example, let $\mathbf{u} \in U$ be a TE and let C and k be variables in $\mathbf{u}$, where C is an array containing an object program in its first k elements $C[1{:}k]$. If "prog" is the start symbol of G, then for each $\tau \in \text{T-PROG}_{\text{prog}}$ and $\gamma \in \text{O-PROG}_{\text{prog}}$,

$$\text{c-gen}_{\text{prog}}(\tau, \gamma) \leftrightarrow (\forall \mathbf{u})(I_0(\mathbf{u}) \supset \mathbf{abs}(\mathbf{D}_{\text{prog}}[\![\tau]\!](\mathbf{u})) = \gamma), \tag{4}$$

where I_0 is a predicate on U that gives the conditions that each initial TE must satisfy prior to compilation, and **abs** is an abstraction function that extracts the value of $C[1{:}k]$ from the value of $\mathbf{u}$ and converts it into a corresponding O-CODE sequence. Note the explicit appearance of **D** in (4). If there exists at least one value of $\mathbf{u}$ for which $I_0(\mathbf{u})$ is true, then it can be readily shown that $\text{c-gen}_{\text{prog}}$ is a *function* from $\text{T-PROG}_{\text{prog}}$ to $\text{O-PROG}_{\text{prog}}$. The (pathological) case where $I_0(\mathbf{u})$ is false for every value of $\mathbf{u}$ is of no practical interest; nothing can be proven about the correctness of the implementation.

To show that c-gen is a weak G-homomorphism, we must prove that for each production $\pi_p = A \rightarrow a_0 B_1 \cdots B_r a_r$, each $\tau_i \in \text{T-PROG}_{B_i}$ and $\gamma_i \in \text{O-PROG}_{B_i}$, $1 \leq i \leq r$,

$$\bigwedge_{1 \leq i \leq r} \text{c-gen}_{B_i}(\tau_i, \gamma_i) \supset \text{c-gen}_A(\mu_p(\tau_1, \ldots, \tau_r), \theta_p(\gamma_1, \ldots, \gamma_r)). \tag{5}$$

Equation (5) comprises the cases of a proof by structural induction over the source language syntax T_G. In principle, the relation c-gen provides the necessary information required to prove compiler implementation correctness via eq. (5). In practice, however, this style of specification requires the inconvenient explicit use of the semantics function **D**, in order to convert L-program text τ into the corresponding function $\mathbf{D}[\![\tau]\!]$ that transforms the value of the TE $\mathbf{u}$, as in (4).

This explicit use of **D** can be eliminated by using "surface semantics" [12], in which the role of **D** is implicit. This style of semantics, called "axiomatic" semantics [10], replaces explicit use of **D** with a logic of L-programs [2] containing "partial correctness formulas" $P\{\text{stm}\}Q$, where "stm" is an L-program and P and Q are formulas in an assertion language associated with L. The formulas P and Q are often called *input-output assertions*; P is called a *precondition* and Q a *postcondition* in $P\{\text{stm}\}Q$. In effect, **D** is a *model* for the resulting proof system.

Toward this end, eq. (5) can be developed into a more useful form by defining, for each $A \in V_N$, $\tau \in \text{T-PROG}_A$, and $\gamma \in \text{O-PROG}_A$, predicates $Q_A(\gamma)$ on $U \times U$ that are used instead of the c-gen$_A$ via

$$\text{c-gen}_A(\tau, \gamma) \leftrightarrow (\forall \mathbf{u}_0)(Q_A(\gamma)(\mathbf{u}_0, \mathbf{D}_A[\![\tau]\!](\mathbf{u}_0))). \tag{6}$$

The right-hand side of (6) is the usual interpretation of the partial correctness formula $\mathbf{u} = \mathbf{u}_0\{\tau(\mathbf{u})\}Q_A(\gamma)(\mathbf{u}_0, \mathbf{u})$. The predicates $Q_A(\gamma)$ are called *translation invariants*; they play a role similar to that played by *inductive assertions* in proofs of correctness for ordinary programs. Intuitively, $Q_A(\gamma)(\mathbf{u}_0, \mathbf{u})$ describes the TE $\mathbf{u}$ (represented by $\mathbf{D}_A[\![\tau]\!](\mathbf{u}_0)$) obtained just after executing (assuming normal termination) a translation sequence $\tau \in \text{T-PROG}_A$, given that $\mathbf{u}_0$ is the value of the TE just prior to the execution of this translation sequence. The specification $\gamma \in \text{O-PROG}_A$ is used in this description so that the compiler-generated object code contained in $\mathbf{u}$ can be related to the specified object code γ. Typically, Q_A states that for any syntactic unit of type $A \in V_N$, the object code generated by the execution of τ is the same as the code prescribed by the specification γ. Additionally, Q_A describes the expected effect of τ on the rest of the TE, such as the fact that certain TE components are modified in a specific way or are not modified at all. For example, the translation invariant counterpart of c-gen$_{\text{prog}}$ is (see eq. (11.prog) of Section 6 for a detailed explanation)

$$Q_{\text{prog}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (I_0(\mathbf{u}_0) \supset C[1{:}k] = \gamma).$$

Although the definition of the translation invariants is the most creative part of the correctness proof, our experience indicates that they are considerably easier to find than typical inductive assertions. Most errors in the formulation of the Q_A are omissions in stating that the values of certain TE components are unchanged after the execution of a translation sequence. Example translation invariants are given in Section 6, and a complete set in Appendix B.

Substituting (6) into (5) yields the implementation correctness criterion: For all

$$\pi_p = A \rightarrow a_0 B_1 \cdots B_r a_r,$$
$$\tau_i \in \text{T-PROG}_{B_i}, \quad \text{and} \quad \gamma_i \in \text{O-PROG}_{B_i}, \quad i = 1, 2, \ldots, r,$$
$$\bigwedge_{1 \le i \le r}(\forall \mathbf{u}_0)(Q_{B_i}(\gamma_i)(\mathbf{u}_0, \mathbf{D}_{B_i}[\![\tau_i]\!](\mathbf{u}_0))) \supset (\forall \mathbf{u}_0)(Q_A(\gamma)(\mathbf{u}_0, \mathbf{D}_A[\![\tau]\!](\mathbf{u}_0))), \tag{7}$$

where

$$\tau = \mu_p(\tau_1, \ldots, \tau_r) \quad \text{and} \quad \gamma = \theta_p(\gamma_1, \ldots, \gamma_r).$$

MAIN THEOREM. (*Compiler Implementation Correctness*). *c-imp is a correct implementation of c-spc if for each $A \in V_N$ there exists a translation invariant Q_A (with Q_{prog} chosen such that the represented c-gen$_{prog}$ is a* function) *such that for*

each production $\pi_p = A \rightarrow a_0 B_1 \cdots B_r a_r$, *each* $\gamma_i \in O\text{-}PROG_{B_i}$, $i = 1, \ldots, r$, *and* $\gamma = \theta_p(\gamma_1, \ldots, \gamma_r)$, *the following assertions hold:*

$$(Q_{B_1}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \,\&\, \cdots \,\&\, Q_{B_r}(\gamma_r)(\mathbf{u}_{r-1}, \mathbf{u}))\{c_p(\mathbf{u})\}Q_A(\gamma)(\mathbf{u}_0, \mathbf{u}) \quad (r > 0) \tag{8a}$$

$$\mathbf{u} = \mathbf{u}_0\{c_p(\mathbf{u})\}Q_A(\theta_p)(\mathbf{u}_0, \mathbf{u}) \quad (r = 0) \tag{8b}$$

where $\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{r-1}$, *and* $\mathbf{u}$ *are distinct variables over* U *and* $\mathbf{u}_1, \ldots, \mathbf{u}_{r-1}$ *are "fresh" variables that do not occur in any of the translation routines or in any of the* $Q_{B_i}(\gamma_i)$. *Equation (8b) represents (8a) when* $r = 0$.

PROOF. The proof of (8a) will be given; the proof of (8b) is similar but much simpler. The criterion for implementation correctness is expressed in eq. (7). Assume the antecedent of (7). From eq. (2), $\tau = \mu_p(\tau_1, \ldots, \tau_r) = \tau_1; \ldots; \tau_r; c_p$. Using the distinct variables $\mathbf{u}_0, \ldots, \mathbf{u}_{r-1}$, we let $\mathbf{u}_{i-1}$ denote the input, and $\mathbf{u}_i$ the output, of τ_i, where $\mathbf{u}_i = \mathbf{D}_{B_i}[\![\tau_i]\!](\mathbf{u}_{i-1})$, $i = 1, 2, \ldots, r-1$. Also, let $\mathbf{u}_{r-1}$ denote the input, and $\mathbf{v} = \mathbf{D}_{B_r}[\![\tau_r]\!] \circ \cdots \circ \mathbf{D}_{B_1}[\![\tau_1]\!](\mathbf{u}_0)$ the output, of τ_r. Thus the antecedent of (7) implies that

$$(\forall \mathbf{u}_0)((\exists \mathbf{u}_1) \cdots (\exists \mathbf{u}_{r-1})(Q_{B_1}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \,\&\, \cdots \,\&\, Q_{B_r}(\gamma_r)(\mathbf{u}_{r-1}, \mathbf{v}))). \tag{9}$$

In order that (9) imply the consequent of (7), it is sufficient to prove that

$$(\forall \mathbf{u}_0)((\exists \mathbf{u}_1) \cdots (\exists \mathbf{u}_{r-1})(Q_{B_1}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \,\&\, \cdots \,\&\, Q_{B_r}(\gamma_r)(\mathbf{u}_{r-1}, \mathbf{v})) \supset Q_A(\gamma)(\mathbf{u}_0, \mathbf{D}_A[\![\tau]\!](\mathbf{u}_0))),$$

which is logically equivalent to

$$(\forall \mathbf{u}_0)(\forall \mathbf{u}_1) \cdots (\forall \mathbf{u}_{r-1})(Q_{B_1}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \,\&\, \cdots \,\&\, Q_{B_r}(\gamma_r)(\mathbf{u}_{r-1}, \mathbf{v}) \supset Q_A(\gamma)(\mathbf{u}_0, \mathbf{D}_A[\![\tau]\!](\mathbf{u}_0))). \tag{10}$$

Since (from eq. (3)) $\mathbf{D}_A[\![\tau]\!](\mathbf{u}_0) = F_p(\mathbf{v})$, (10) is the interpretation of

$$(Q_{B_1}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \,\&\, \cdots \,\&\, Q_{B_r}(\gamma_r)(\mathbf{u}_{r-1}, \mathbf{u}))\{c_p(\mathbf{u})\}Q_A(\gamma)(\mathbf{u}_0, \mathbf{u}),$$

whose proof is sufficient for implementation correctness. □

Equation (8) states the input-output assertions (in terms of the translation invariants Q_A) which, if satisfied by the TRs c_p, ensure a correct compiler implementation.

6. EXAMPLE TRANSLATION ROUTINES AND TRANSLATION INVARIANTS

In this section we give concrete substance to some of the concepts discussed and developed in Sections 4 and 5, by presenting examples of translation routines (TRs) and translation invariants (TIs) associated with the AG fragments given in Section 3.

The translation environment (TE) associated with our example postfix compiler is a collection of data structures whose names, types, and uses are given below:

C: O-CODE	indexed sequence (array) of target machine instructions containing the object code generated during compilation;
k: N	index to C which points to the instruction most recently generated;

M: E, E = Id*	indexed sequence of identifiers, used as a symbol table;
m: N	index to M which points to the most recently declared active identifier;
V: ($N \cup$ Id)*	auxiliary translation stack of object program addresses and identifiers;
v: N	index to V which points to the "top" element of V;
b, pos: N	auxiliary variables.

Thus in our example,

$$\mathbf{u} = \langle C, k, M, m, V, v, b, pos \rangle, \quad \text{and}$$
$$U = \text{O-CODE} \times N \times E \times N \times (N \cup \text{Id})^* \times N \times N \times N.$$

The example TRs given below are written in an ALGOL-like language L, which needs no formal definition. Similarly, the assertion language associated with L is so straightforward that it also requires almost no explanation. If A is an indexed sequence (array) of values and k is a suitable index, then $A[k]$ denotes the kth element of A and $A[k_1\text{: } k_2]$, $k_1 \le k_2$, denotes the "slice" (subarray) of A containing elements k_1 through k_2. We adopt the convention that arrays are potentially infinite and indexed from 1 and that $A[1{:}k]$ denotes the "interesting" contents of A; $k = 0$ denotes that A is "empty." When $k_1 \le i \le k_2$, $A[k_1\text{: } k_2]\{a/i\}$ denotes an array slice which is the same as $A[k_1\text{: } k_2]$, except that $A[i] = a$. In the assertions that appear in the TIs and correctness proofs, various instances x_0, x_1, x_2, ... of the same variable x are used to denote the value of x at different program points; x_0 often denotes an "initial" value of x.

The example TRs and TIs given below represent only selected fragments of our example compiler; the complete version is given in Appendix B.

The first example fragment deals with entire programs:

π_1 = prog $\longrightarrow$ block

```
c1: k := k+1; C[k] := (HALT)
```

c_1 is the final TR executed during any parse; it simply adds a (HALT) instruction to the end of the object program. The associated translation invariant is

$$Q_{\text{prog}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow I_0(\mathbf{u}_0) \supset C[1{:}k] = \gamma \qquad \text{(11.prog)}$$

where

$$I_0(\mathbf{u}_0) \leftrightarrow k_0 = 1 \;\&\; m_0 = 0 \;\&\; v_0 = 0 \;\&\; C_0[1] = \text{(START)}$$

defines an initial value that the TE must have prior to initiating parsing. Equation (11.prog) states that for any object program $\gamma \in \text{O-PROG}_{\text{prog}}$, if the TE is properly initialized, then the object program $C[1{:}k]$ in the final TE $\mathbf{u}$ is exactly the same as the object program γ required by the compiler specification. Note that C and k are components of the final TE $\mathbf{u}$, and that $C[1{:}k]$ is the interesting slice of the final value of the object code array.

The second fragment treats new declarations:

$\pi_4 = \text{dcl} \rightarrow \text{dcl}_1, \text{id}$

```
c4: SEARCH(m-b+1, m);
  IF pos = 0
  THEN k := k+1; C[k] := (NEW);
       m := m+1; M[m] := V[v];
       b := b+1
  ELSE SKIP    /* ignore duplicate declaration */
  FI;
  v := v-1    /* pop V stack */
```

This TR checks whether the newly declared identifier has been previously declared in the current block. If so, this new declaration is ignored (without comment); if not, then the new identifier is entered into the symbol table M and a (NEW) instruction is emitted into the object code array C. SEARCH(i, j) is an L-program segment (a subroutine or an inline "macro") that searches for the rightmost occurrence of the newly declared identifier in the symbol table slice $M[m - b + 1{:}m]$, which is that portion of M pertinent to the current block. The newly declared identifier is assumed to be on the top of the V stack, i.e., $V[v] = \text{id}$. SEARCH is characterized by

$$(i = i_0 \,\&\, j = j_0 \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \,\&\, b = b_0)$$
$$\{\text{SEARCH}(i, j)\}$$
$$(pos = \text{loc}(V[v], M[i{:}j]) \,\&\, i = i_0 \,\&\, j = j_0 \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \,\&\, b = b_0).$$

The associated TI is

$$Q_{\text{dcl}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (M[1{:}m] = M_0[1{:}m_0] \bullet (\gamma)_1 \,\&\, m = m_0 + b \,\&\, C[1{:}k] = C_0[1{:}k_0] \bullet (\gamma)_2 \,\&\, V[1{:}v] = V_0[1{:}v_0] \bullet \langle b_0 \rangle), \tag{11.dcl}$$

where $\gamma \in E \times \text{O-CODE}$ and $(\gamma)_i$, $i = 1, 2$, denotes the ith component of γ.

The assumption that the newly declared identifier is on the top of V is expressed by the TI

$$Q_{\text{id}}(\theta_{[x]})(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (C[1{:}k] = C_0[1{:}k_0] \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \bullet \langle x \rangle \,\&\, b = b_0). \tag{11.id}$$

This TI states that the offline process that obtains the identifier x from the input stream pushes it onto the V stack, and leaves the rest of the TE unmodified.

The third example fragement deals with iteration statements:

$\pi_{10} = \text{stm} \rightarrow \text{test DO body END}$

```
c10: C[V[v]] := (JPF, k+2);    /* backpatch object code */
     k := k + 1; C[k] := (JMP, V[v-1]);    /* jump to loop
                                              entry point */
     v := v-2    /* pop saved program addresses from V */
```

When this TR is executed, it is assumed that the object code for the test and the body of the iteration statement have already been emitted into the object code array C. In addition, it is assumed that the object code address that should

contain the conditional jump instruction following the test code is on the top of V (in $V[v]$), and that the object code address of the first instruction of the test code is next down in V (in $V[v-1]$). The conditional jump instruction is backpatched into the (already generated) object code, and the unconditional jump to the head of the loop is emitted onto the end of the object code. Once these instructions are emitted, the two object code addresses are no longer needed, and are popped from V. The associated TI is

$$\begin{aligned} Q_{\text{stm}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (&C[1{:}k] \\ &= C_0[1{:}k_0] \bullet \gamma(k_0 + 1, M_0[1{:}m_0]) \;\&\; k_0 < k \\ &\;\; \&\; M[1{:}m] = M_0[1{:}m_0] \;\&\; V[1{:}v] = V_0[1{:}v_0] \;\&\; b = b_0), \end{aligned} \tag{11.stm}$$

where $\gamma \in \text{O-PROG}_{\text{stm}} = N \times E \rightarrow \text{O-CODE}$; $\gamma(k, M)$ denotes object code generated starting at program location k, with respect to the environment represented by symbol table M.

The final example fragment considers expressions that are just identifiers:

$\pi_{20} = \text{pri} \rightarrow \text{id}$

```
c20: SEARCH(1, m);
     k := k + 1;
     IF pos > 0
     THEN C[k] := (LD, pos)
     ELSE C[k] := (ERR, 20)
     FI;
     v := v-1
```

A search for the rightmost occurrence of the identifier (on the top of V) in the entire symbol table is made. If found, then a load instruction is emitted; otherwise, an error instruction is emitted. In either case the identifier is popped from V. The associated TI is

$$\begin{aligned} Q_{\text{pri}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (&C[1{:}k] = C_0[1{:}k_0] \bullet \gamma(k_0 + 1, M_0[1{:}m_0]) \;\&\; k_0 < k \\ &\;\; \&\; M[1{:}m] = M_0[1{:}m_0] \;\&\; V[1{:}v] = V_0[1{:}v_0] \;\&\; b = b_0), \end{aligned} \tag{11.pri}$$

which is similar to Q_{stm}.

7. EXAMPLE PROOFS

In this section we present a few representative cases of the proof that our example compiler implementation is correct with respect to the specification of the example compiler. The proofs are straightforward; we include them primarily to convey their style and organization. Although the proofs can be done quite formally, we give informal proofs so that the presentation does not become unnecessarily ponderous.

Equation (8) is the correctness criterion; the implementation correctness proof consists of several cases, one for each production. Three representative cases are considered in detail; eq. (8) for each case is displayed in a vertical format that makes its proof convenient to discuss and follow. The format for each case (for

the pth production $\pi_p = A \rightarrow a_0 B_1 \cdots B_r a_r$) is

$$\begin{array}{l} Q_{B_1}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \\ \hline \vdots \\ \hline Q_{B_r}(\gamma_r)(\mathbf{u}_{r-1}, \mathbf{u}) \\ \hline \{c_p(\mathbf{u})\} \\ \hline Q_A(\theta_p(\gamma_1, \ldots, \gamma_r))(\mathbf{u}_0, \mathbf{u}) \end{array}$$

Case 1. prog $\rightarrow$ block. We must prove that for all $\gamma \in \text{O-PROG}_{\text{block}}$,

$$Q_{\text{block}}(\gamma)(\mathbf{u}_0, \mathbf{u})\{c_1(\mathbf{u})\}Q_{\text{prog}}(\theta_1(\gamma))(\mathbf{u}_0, \mathbf{u}).$$

Noting that

$$Q_{\text{prog}}(\theta_1(\gamma))(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (I_0(\mathbf{u}_0) \supset C[1{:}k] = \theta_1(\gamma)),$$

we may instead prove that

$$(Q_{\text{block}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \;\&\; I_0(\mathbf{u}_0))\{c_1(\mathbf{u})\}C[1{:}k] = \theta_1(\gamma),$$

where

$$\theta_1(\gamma) = \langle(\text{START})\rangle \bullet \gamma(2, \langle\rangle) \bullet \langle(\text{HALT})\rangle.$$

Using these facts, the proof of Case 1 takes the form

(a1) $C[1{:}k] = C_0[1{:}k_0] \bullet \gamma(k_0 + 1, M_0[1{:}m_0]) \;\&\; k > k_0$
(b1) $M[1{:}m] = M_0[1{:}m_0]$
(c1) $V[1{:}v] = V_0[1{:}v_0] \;\&\; b = b_0$
(d1) $k_0 = 1 \;\&\; v_0 = 0 \;\&\; m_0 = 0 \;\&\; C_0[1] = (\text{START})$

```
================================
c1:     {k := k+1; C[k] := (HALT)}
================================
```

(e1) $C[1{:}k] = \langle(\text{START})\rangle \bullet \gamma(2, \langle\rangle) \bullet \langle(\text{HALT})\rangle.$

Combining (d1) and (a1) and noting that $M_0[1{:}0] = \langle\rangle$ and $C_0[1{:}1] = \langle C_0[1]\rangle = \langle(\text{START})\rangle$, reduces (a1) to

(f1) $C[1{:}k] = \langle(\text{START})\rangle \bullet \gamma(2, \langle\rangle) \;\&\; k > 1.$

It is now clear that c_1 is partially correct with respect to (f1) and (e1).

Case 4. dcl $\rightarrow$ dcl, id. We must prove that for all $\gamma \in \text{O-PROG}_{\text{dcl}}$ and $x \in \text{Id}$,

$$(Q_{\text{dcl}}(\gamma)(\mathbf{u}_0, \mathbf{u}_1) \;\&\; Q_{\text{id}}(\theta_{[x]})(\mathbf{u}_1, \mathbf{u}))\{c_4(\mathbf{u})\}Q_{\text{dcl}}(\theta_4(\gamma, \theta_{[x]}))(\mathbf{u}_0, \mathbf{u}).$$

Let $\gamma = \langle d, z\rangle$. The proof of Case 4 takes the form

(a4) $C_1[1{:}k_1] = C_0[1{:}k_0] \bullet z$
(b4) $M_1[1{:}m_1] = M_0[1{:}m_0] \bullet d \;\&\; m_1 = m_0 + b_1$
(c4) $V_1[1{:}v_1] = V_0[1{:}v_0] \bullet \langle b_0\rangle$

(d4) $C[1{:}k] = C_1[1{:}k_1]$
(e4) $M[1{:}m] = M_1[1{:}m_1]$
(f4) $V[1{:}v] = V_1[1{:}v_1] \bullet \langle x\rangle$
(g4) $b = b_1$

==============================

```
c4:     {SEARCH(m-b+1, m);
         IF   pos = 0
         THEN k := k+1; C[k] := (NEW);
              m := m+1; M[m] := V[v];
              b := b+1
         ELSE SKIP  /* ignore duplicate declaration */
         FI;
         v := v-1  /* pop V stack */}
```

==============================

(h4) $C[1{:}k] = C_0[1{:}k_0] \bullet \theta_4(\gamma, \theta_{[x]})_2$
(i4) $M[1{:}m] = M_0[1{:}m_0] \bullet \theta_4(\gamma, \theta_{[x]})_1 \;\&\; m = m_0 + b$
(j4) $V[1{:}v] = V_0[1{:}v_0] \bullet \langle b_0\rangle$

To begin with, (1.4) and the above definition of $\langle d, z\rangle$ yield

$$\theta_4(\gamma, \theta_{[x]}) = \textbf{if } \text{found}(d, x) \textbf{ then } \langle d, z\rangle \textbf{ else } \langle d \bullet \langle x\rangle, z \bullet \langle(\text{NEW})\rangle\rangle$$

and thus (h4) and (i4) can be replaced by

(k4) $C[1{:}k] = C_0[1{:}k_0] \bullet \textbf{if } \text{found}\,(x, d) \textbf{ then } z \textbf{ else } (z \bullet \langle(\text{NEW})\rangle)$
(m4) $M[1{:}m]= M_0[1{:}m_0] \bullet \textbf{if } \text{found}\,(x, d) \textbf{ then } d \textbf{ else } (d \bullet \langle x\rangle)$
$\quad\& \; m = m_0 + b.$

Equations (a4)–(g4) can be combined and replaced by

(n4) $C[1{:}k] = C_0[1{:}k_0] \bullet z$
(p4) $M[1{:}m] = M_0[1{:}m_0] \bullet d \;\&\; d = M[m_0 + 1{:}m] \;\&\; m = m_0 + b$
(q4) $V[1{:}v] = V_0[1{:}v_0] \bullet \langle b_0\rangle \bullet \langle x\rangle$

We can characterize the effect of the call to SEARCH in c_4 (which, by (p4), is SEARCH(m_0 + 1, m)) by

(r4) $pos = \text{loc}(V[v], M[m_0{:}m]) = \text{loc}(x, d),$

and then prove the rest of c_4 (after the call to SEARCH) partially correct with respect to (n4) & (p4) & (q4) & (r4) and (k4) & (m4). This is easily shown by

recalling that found(x, d) iff loc$(x, d) > 0$ (Section 3), and then considering the cases $pos = 0$ and $pos > 0$. □

Case 10. stm → test DO body END. We must prove that for all $\gamma_1 \in$ O-PROG$_{\text{test}}$ and $\gamma_2 \in$ O-PROG$_{\text{body}}$,

$$(Q_{\text{test}}(\gamma_1)(\mathbf{u}_0, \mathbf{u}_1) \;\&\; Q_{\text{body}}(\gamma_2)(\mathbf{u}_1, \mathbf{u}))\{c_{10}(\mathbf{u})\}Q_{\text{stm}}(\theta_{10}(\gamma_1, \gamma_2))(\mathbf{u}_0, \mathbf{u}).$$

The proof of Case 10 takes the form

(a10) $C_1[1{:}k_1 - 1] = C_0[1{:}k_0] \bullet \gamma_1(k_0 + 1, M_0[1{:}m_0])$
(b10) $M_1[1{:}m_1] = M_0[1{:}m_0]$
(c10) $V_1[1{:}v_1] = V_0[1{:}v_0] \bullet \langle k_0 + 1\rangle \bullet \langle k_1\rangle$
(d10) $k_0 < k_1 - 1 \;\&\; b_1 = b_0$

(e10) $C[1{:}k] = C_1[1{:}k_1] \bullet \gamma_2(k_1 + 1, M_1[1{:}m_1])$
(f10) $M[1{:}m] = M_1[1{:}m_1]$
(g10) $V[1{:}v] = V_1[1{:}v_1]$
(h10) $k_1 < k \;\&\; b = b_1$

==========

```
c10:    {C[V[v]] := (JPF, k+2);  /* backpatch */
         k := k+1; C[k] := (JMP, V[v-1]);  /* loop back */
         v := v-2}
```

==========

(i10) $C[1{:}k] = C_0[1{:}k_0] \bullet \theta_{10}(\gamma_1, \gamma_2)(k_0 + 1, M_0[1{:}m_0])$
(j10) $M[1{:}m] = M_0[1{:}m_0]$
(k10) $V[1{:}v] = V_0[1{:}v_0]$
(l10) $k_0 < k \;\&\; b = b_0$

Since c_{10} does not change the symbol table M, it is clear that (j10) follows from (b10) and (f10). It is also readily apparent that (l10) follows from (d10) and (h10) and the statement $k := k + 1$; (k10) follows from (c10) and (g10) and the statement $v := v - 2$. Thus it remains to show that (i10) is a consequence of (a10) through (h10) and the TR c_{10}. Let $e = M_0[1{:}k_0](= M_1[1{:}m_1] = M[1{:}m]$ before or after c_{10}). Back-substituting (i10) through c_{10} yields (with the help of (c10) and (g10), and noting that $k_1 < k + 1$ from (h10)):

(m10) $C[1{:}k + 1]\{(\text{JMP}, k_0 + 1)/k + 1\}\{(\text{JPF}, k + 2)/k_1\}$
$= C_0[1{:}k_0] \bullet \text{c-spc}_{\text{stm}}(\pi_{10}(t_1, t_2))(k_0 + 1, e).$

Equation (1.10) yields

(n10) $\theta_{10}(\gamma_1, \gamma_2)(k_0 + 1, e)$
$= \gamma_1(k_0 + 1, e) \bullet \langle(\text{JPF}, n_2)\rangle \bullet \gamma_2(n_1, e) \bullet \langle(\text{JMP}, k_0 + 1)\rangle$

where

$$n_1 = k_0 + \lg(\gamma_1(k_0 + 1, e)) + 2$$

and

$$n_2 = n_1 + \lg(\gamma_2(n_1, e)) + 1.$$

From (a10) and (e10) we can infer that

$$\lg(\gamma_1(k_0 + 1, e)) = k_1 - k_0 - 1$$

and

$$\lg(\gamma_2(k_1 + 1, e)) = k - k_1,$$

yielding from (m10) and (n10),

$$C[1{:}k+1]\{(\mathrm{JMP}, k_0+1)/k+1\}\{(\mathrm{JPF}, k+2)/k_1\} = C_0[1{:}k_0] \bullet \gamma_1(k_0+1, e) \bullet \langle(\mathrm{JPF}, k+2)\rangle \bullet \gamma_2(k_1+1, e) \bullet \langle(\mathrm{JMP}, k_0+1)\rangle,$$

which, since $k_0 + 1 < k_1 < k$ by (d10) and (h10), reduces to

$$\text{(o10)} \quad C[1{:}k]\{(\mathrm{JPF}, k+2)/k_1\} = C_0[1{:}k_0] \bullet \gamma_1(k_0+1, e) \bullet \langle(\mathrm{JPF}, k+2)\rangle \bullet \gamma_2(k_1+1, e).$$

(o10) is clearly implied by (a10) through (h10), completing the proof. □

This concludes the example proofs.

Structured Correctness Proofs

It is important to note that our proof method permits establishing the boundary between "primary" and "offline" syntactic processes at any syntactic level. In the proofs given in this paper, this boundary was drawn so that identifiers and constants were handled by "offline" processes (i.e., lexical analysis), whose effect on the translation implementation was characterized by the translation invariants Q_{id} and Q_{const}. Also note that it was not necessary to provide the TRs $c_{[x]}$, $x \in \mathrm{Id}$, and $c_{[n]}$, $n = 0, 1, 2, \ldots$; they were never used in the proofs, their effect being specified by Q_{id} and Q_{const}.

Thus our proof method supports a "structured" development, by successive refinement, of compiler implementations and proofs of their correctness. For example, the syntactic structure of expressions could remain unelaborated, their parsing and translation being carried out by an "offline" process whose net effect on the translation is specified by the translation invariant Q_{exp}. The implementation correctness proof could be carried out at this level of detail, and then later refined to consider the detailed syntactic structure of expressions. Note that this refinement would not require redoing any of the existing proof, but would only require adding translation invariants, TRs, and proof cases pertinent to the detailed structure of expressions. This is a natural consequence of the structural induction underlying the proof method.

8. CONCLUSION

The principal contributions of this paper are techniques for the (algebraic) specification of compilers and a practical class of compiler implementations, together with a methodology for proving the correctness of compiler implementations with respect to specifications. Our main theorem, although derived in the context of postfix compiler implementations, is actually a quite general theorem about the correctness of postfix transducers.

Our method of compiler specification is algebraic in nature, but uses attribute grammars as a means of obtaining an algebraic specification. Whether or not an

AG is used as a preliminary step toward an algebraic specification is largely a matter of personal taste; algebraic specifications can be written directly, using the style and methodology of denotational semantics. The preliminary use of AGs may be a helpful intuitive aid in the design of large compiler specifications.

The compiler implementations considered are very simple, but widely used in practice. These are the one-pass postfix translators concurrently driven by deterministic SHIFT-REDUCE parsers such as LR(k) parsers, causing the invocation of a sequence of translation routines, which operate upon a translation environment. Syntax-directed translators driven by LL(k) parsers can also be made to fit this translator model. Alternatively, the TRs could be invoked by first generating an abstract syntax tree under the direction of a concrete (LR(k) or LL(k)) parse and then executing one or more "tree walking" procedures over this tree, during which TRs are invoked. This method can be used to specify multipass compilers. The general constraint to be met is that the TR sequence invoked be specified within the framework of our proof method.

An important area for further investigation is whether our specification and proof techniques "scale up" with respect to larger size, more troublesome programming language features such as general gotos, forward references, and more complex backpatching [20], and the issue of multipass compilers. It is possible that machine-assisted program proof systems might help to manage the increase in detail that accompanies larger size.

A more speculative and difficult topic for further research is the derivation of the translation routines from the specifications supplied by the translation invariants. Equation (8) in Section 5 gives the input-output assertions, stated in terms of the translation invariants, that each TR must satisfy to ensure a correct compiler implementation. As an example, consider the proof of Case 10 in Section 7. Equations (a10)–(h10) are equivalent to the conjunction of

$$C[1{:}k] = C_0[1{:}k_0] \bullet \gamma_1(k_0 + 1, e) \bullet \langle ? \rangle \bullet \gamma_2(k_1 + 1, e), \tag{12.1}$$

where

$$k_1 = k_0 + \lg(\gamma_1(k_0 + 1, e)) + 1,$$

$$M[1{:}m] = M_0[1{:}m_0], \tag{12.2}$$

$$V[1{:}v] = V_0[1{:}v_0] \bullet \langle k_0 + 1 \rangle \bullet \langle k_1 \rangle, \tag{12.3}$$

$$k_0 < k_1 - 1 \;\&\; k_1 < k, \tag{12.4}$$

$$b = b_0, \tag{12.5}$$

where "?" denotes an undefined value. Similarly, eqs. (i10)–(l10) are equivalent to the conjunction of

$$C[1{:}k] = C_0[1{:}k_0] \bullet \gamma_1(k_0 + 1, e) \bullet \langle (\mathrm{JPF}, n_2) \rangle \bullet \gamma_2(n_1, e) \bullet \langle (\mathrm{JMP}, k_0 + 1) \rangle, \tag{13.1}$$

where

$$n_1 = k_0 + \lg(\gamma_1(k_0 + 1, e)) + 2$$

and

$$n_2 = n_1 + \lg(\gamma_2(n_1, e)) + 1,$$

(13.2) $M[1{:}m] = M_0[1{:}m_0]$
(13.3) $V[1{:}v] = V_0[1{:}v_0]$
(13.4) $k_0 < k$
(13.5) $b = b_0$

The TR c_{10} must be partially correct with respect to (12) and (13). It is clear that c_{10} need not modify M or b. From (12.1) and (13.1), $n_1 = k_1 + 1$. Just *before* executing c_{10}, $V[v] = k_1$, $V[v - 1] = k_0 + 1$, and $n_2 = k + 2$. Thus a comparison of (12.1) and (13.1) reveals that c_{10} must assign (JPF, n_2) = (JPF, $k + 2$) into $C[k_1] = C[V[v]]$. Also, in terms of the value of k just *before* executing c_{10}, $C[1{:}k]$ must be the same just before and after executing c_{10}, and (13.1) requires that c_{10} must assign (JMP, $k_0 + 1$) = (JMP, $V[v - 1]$) into $C[k + 1]$ and must then increment k by 1. The relative values of k_0, k_1, and k given by (12.4) and (13.4) validate the foregoing actions. Finally, (12.3) and (13.3) require c_{10} to remove the top 2 values from V (i.e., v must be decremented by 2). Therefore all of the above actions are summarized in

```
C[V[v]] := (JPF, k+2);
C[k+1] := (JMP, V[v-1]); k := k+1;
v := v-2
```

which is obviously equivalent to the given c_{10}.

Comparison to Related Work

The most significant published research related to ours is that of Polak [18]; this work is general and substantial, and is required reading for anyone seriously interested in compiler correctness. In the following discussion, we shall limit our remarks to a comparison of Polak's and our basic approaches to the compiler implementation correctness problem.

Polak first converts program text into abstract syntax trees, which are input (together with compile-time environments) into code generation functions. These abstract syntax trees are generally associated with major nonterminals (syntactic categories) in the source grammar (statements, expressions, etc.), and the associated code generation functions contain corresponding syntactic subcases (e.g., the code generation function for expressions contains cases for simple variables, constants, expressions involving arithmetic operators, etc.). These code generation functions, as well as the utility functions invoked by them, are given the same kind of pre-/postcondition specification that we use; the functions are proven partially correct with respect to their specifications.

There are differences, some inessential and others significant, between our and Polak's approach to the compiler implementation correctness problem. Polak applies code generation functions to abstract syntax trees, whereas we invoke, at the direction of the parser, translation routines to modify (by side effect) a suitably initialized global translation environment. The specifications of Polak's code generation functions correspond to our translation invariants; both kinds of specification must be supplied by the compiler implementer. These differences are inessential.

Our approaches differ significantly in the nature of the code generation specifications. Polak's code generation specifications include the semantics of the object (i.e., target machine) language, whereas we use this semantics only in

the compiler *specification* correctness problem (Figure 2). We consider compiler specifications, and the correctness of implementations with respect to those specifications, to be concerned only with the translation of a representation of source program *syntax* into object program *text* (Figure 3). In our view, semantic correspondence between source and object programs is associated only with the compiler specification correctness problem. Clearly, this difference of viewpoint arises from the position of the boundary between compiler specification correctness and implementation correctness; our formulation of the compiler implementation correctness problem results in specifications and correctness proofs over simpler domains and theories (i.e., those concerned only with syntax). Both Polak's code generation specifications and our translation invariants must be general enough to deal with the syntactic subcases associated with each type of abstract syntax tree or (in our approach) nonterminal symbol of the source language grammar; our main theorem states how to derive, from the translation invariants, the specifications that each translation routine (associated with a production that deals with a syntactic subcase) must satisfy.

It is hoped that this research has achieved its goal of providing a step toward formalization of and reasoning about a familiar and proven class of compiler implementation techniques.

APPENDIX A. Example Compiler Specification

An example compiler specification is given in this appendix, both as an attribute grammar and as operations θ_p of the G algebra O-PROG which, together with the operations π_p of the syntactic algebra T_G, uniquely determine the compiler specification as a homomorphism from T_G to O-PROG.

Synthesized Attributes

z: O-CODE; object code
d: E; local symbol table defined by a local declaration, where E = Id*
w: Id; an identifier
v: Z; (integer) value of a constant

Inherited Attributes

n: N; starting location of object code in program storage
e: E; global symbol table (defined by global declarations)

Association of Attributes with Nonterminals. For each nonterminal A, let INH(A) and SYN(A) denote, respectively, the set of inherited and synthesized attributes of A.

$$\text{INH(prog)} = \{\} \qquad \text{SYN(prog)} = \{z\}$$

For A = block, body, stm, ifcl, tpart, epart, test, exp, term, pri, cond:

$$\text{INH}(A) = \{n, e\} \qquad \text{SYN}(A) = \{z\}$$

$$\begin{aligned} \text{INH(dcl)} &= \{\} & \text{SYN(dcl)} &= \{d, z\} \\ \text{INH(while)} &= \{\} & \text{SYN(while)} &= \{\} \\ \text{INH(id)} &= \{\} & \text{SYN(id)} &= \{w\} \\ \text{INH(const)} &= \{\} & \text{SYN(const)} &= \{v\} \end{aligned}$$

Carriers of the Algebra O-PROG

$$\text{O-PROG}_{\text{prog}} = \text{O-CODE}$$

For A = block, body, stm, ifcl, tpart, epart, test, exp, term, pri, cond:

$$\text{O-PROG}_A = N \times E \rightarrow \text{O-CODE}$$

$$\begin{aligned}\text{O-PROG}_{\text{dcl}} &= E \times \text{O-CODE}\\ \text{O-PROG}_{\text{while}} &= \{\text{WHILE}\}\\ \text{O-PROG}_{\text{id}} &= \text{Id}\\ \text{O-PROG}_{\text{const}} &= Z\end{aligned}$$

For a production $\pi_p = A \rightarrow a_0 B_1 \cdots B_r a_r$, the operation θ_p has type

$$\theta_p\colon \text{O-PROG}_{B_1} \times \cdots \times \text{O-PROG}_{B_r} \rightarrow \text{O-PROG}_A.$$

"•" is a polymorphic sequence concatenation operator.

Semantic Rules

π_1 = prog → block
n.block = 2
e.block = ⟨⟩
z.prog = ⟨(START)⟩ • z.block • ⟨(HALT)⟩
$\theta_1(f) = \langle(\text{START})\rangle \bullet f(2, \langle\rangle) \bullet \langle(\text{HALT})\rangle$

π_2 = block → BEGIN dcl; body END
n.body = n.block + lg(z.dcl)
e.body = e.block • d.dcl
z.block = z.dcl • z.body • ⟨(END, lg($d.dcl$))⟩
$\theta_2(\langle d, z\rangle, f)(n, e) = z \bullet f(n + \lg(z), e \bullet d) \bullet \langle(\text{END}, \lg(d))\rangle$

π_3 = dcl → NEW id
$d.dcl = \langle w.id\rangle$
$z.dcl = \langle(NEW)\rangle$
$\theta_3(w) = \langle\langle w\rangle, \langle(\text{NEW})\rangle\rangle$

π_4 = dcl → dcl_1, id
$d.dcl$ = **if** found ($w.id$, $d.dcl_1$) **then** $d.\text{dcl}_1$ **else** $d.\text{dcl}_1$ • $\langle w.id\rangle$
$z.dcl$ = **if** found ($w.id$, $d.dcl_1$) **then** $z.\text{dcl}_1$ **else** $z.\text{dcl}_1$ • ⟨(NEW)⟩
$\theta_4(\langle d, z\rangle, w)$ = **if** found (w, d) **then** $\langle d, z\rangle$ **else** $\langle d \bullet \langle w\rangle, z \bullet \langle(\text{NEW})\rangle\rangle$

π_5 = body → body_1; stm
$n.\text{body}_1 = n$.body
$e.\text{body}_1 = e$.body
n.stm = n.body + lg($z.\text{body}_1$)
e.stm = e.body
z.body = $z.\text{body}_1$ • z.stm
$\theta_5(f, g)(n, e) = f(n, e) \bullet g(n + \lg(f(n, e)), e)$

π_6 = body $\rightarrow$ stm
n.stm = n.body
e.stm = e.body
z.body = z.stm
$\theta_6(f) = f$

π_7 = stm $\rightarrow$ block
n.block = n.*stm*
e.block = e.stm
z.stm = z.block
$\theta_7(f) = f$

π_8 = stm $\rightarrow$ id := exp
n.exp = n.stm
e.exp = e.stm
z.stm = z.*exp* • **if** found (w.*id*, e.stm) **then** $\langle$(ST, loc(w.*id*, e.*stm*))$\rangle$
else $\langle$(ERR, 8)$\rangle$
$\theta_8(w, f)(n, e) = f(n, e)$ • **if** found (w, e) **then** $\langle$(ST, loc(w, e))$\rangle$
else $\langle$(ERR, 8)$\rangle$

π_9 = stm $\rightarrow$ ifcl tpart epart
n.ifcl = n.stm
e.ifcl = e.stm
n.tpart = n.stm + lg(z.ifcl) + 1
e.tpart = e.stm
n.epart = n.stm + lg(z.ifcl) + lg(z.tpart) + 2
e.epart = e.*stm*
z.stm = z.ifcl • $\langle$(JPF, k_2)$\rangle$ • z.tpart • $\langle$(JMP, k_3)$\rangle$ • z.epart
where k_2 = n.stm + lg(z.ifcl) + lg(z.tpart) + 2
and $k_3 = k_2$ + lg(z.epart)
$\theta_9(f, g, h)(n, e) = f(n, e)$ • $\langle$(JPF, k_2)$\rangle$ • $g(k_1, e)$ • $\langle$(JMP, k_3)$\rangle$ • $h(k_2, e)$
where $k_1 = n$ + lg($f(n, e)$) + 1
and $k_2 = k_1$ + lg($g(k_1, e)$) + 1
and $k_3 = k_2$ + lg($h(k_2, e)$)

π_{10} = stm $\rightarrow$ test DO body END
n.test = n.stm
e.test = e.stm
n.body = n.stm + lg(z.test) + 1
e.body = e.stm
z.stm = z.test • $\langle$(JPF, k_2)$\rangle$ • z.body • $\langle$(JMP, n.*stm*)$\rangle$
where k_2 = n.*stm* + *lg*(z.*test*) + *lg*(z.*body*) + 2
$\theta_{10}(f, g)(n, e) = f(n, e)$ • $\langle$(JPF, k_2)$\rangle$ • $g(k_1, e)$ • $\langle$(JMP, n)$\rangle$
where $k_1 = n$ + lg($f(n, e)$) + 1
and $k_2 = k_1$ + lg($g(k_1, e)$) + 1

π_{11} = ifcl → IF cond THEN
n.cond = n.ifcl
e.cond = e.ifcl
z.ifcl = z.cond
$\theta_{11}(f) = f$

π_{12} = tpart → body ELSE
Similar to π_{11}; omitted.
$\theta_{12}(f) = f$

π_{13} = epart → body FI
Similar to π_{11}.
$\theta_{13}(f) = f$

π_{14} = test → while cond
n.cond = n.test
e.cond = e.test
z.test = z.cond
$\theta_{14}(\text{WHILE}, f) = f$

The nonterminal "while" has no attributes; the "dummy" value WHILE is used.

π_{15} = while → WHILE
θ_{15} = WHILE

π_{16} = exp → exp_1 aop term
$n.\text{exp}_1 = n.\text{exp}$
$e.\text{exp}_1 = e.\text{exp}$
n.term = n.exp + lg($z.\text{exp}_1$)
e.term = e.exp
z.exp = $z.\text{exp}_1$ • z.term • ⟨(AOP)⟩
aop ∈ {+, −, ···} and AOP ∈ {ADD, SUB, ···}
$\theta_{16}(f, g)(n, e) = f(n, e) \bullet g(n + \lg(f(n, e)), e) \bullet \langle(\text{AOP})\rangle$

π_{17} = exp → term
n. term = n.exp
e.term = e.exp
z.exp = z.term
$\theta_{17}(f) = f$

π_{18} = term → term_1 mop pri
Similar to π_{16};
mop ∈ {*, /, ···} and MOP ∈ {MUL, DIV, ···}
$\theta_{18}(f, g)(n, e) = f(n, e) \bullet g(n + \lg(f(n, e)), e) \bullet \langle(\text{MOP})\rangle$

π_{19} = term → pri
Similar to π_{17}.
$\theta_{19}(f) = f$

π_{20} = pri $\rightarrow$ id
z.pri = **if** found (w.id, e.pri) **then** $\langle$(LD, loc(w.id, e.pri))$\rangle$
else $\langle$(ERR, 20)$\rangle$
$\theta_{20}(w)(n, e)$ = **if** found (w, e) **then** $\langle$(LD, loc(w, e))$\rangle$ **else** $\langle$(ERR, 20)$\rangle$

π_{21} = pri $\rightarrow$ const
z.pri = $\langle$(LDI, v.const)$\rangle$
$\theta_{21}(v)(n, e) = \langle$(LDI, v)$\rangle$

π_{22} = pri $\rightarrow$ (exp)
Similar to π_{17}.
$\theta_{22}(f) = f$

π_{23} = cond $\rightarrow$ $\exp_1$ rop $\exp_2$
Similar to π_{16};
rop $\in$ {=, /=, <, $\cdots$} and ROP $\in$ {EQ, NE, LT, $\cdots$}
$\theta_{23}(f, g)(n, e) = f(n, e) \bullet g(n + \lg(f(n, e)), e) \bullet \langle$(ROP)$\rangle$

$\pi_{[x]}$ = id $\rightarrow x$, $x \in$ Id
w.id = x
$\theta_{[x]} = x$

$\pi_{[n]}$ = const $\rightarrow n$, $n = 0, 1, 2, \cdots$ v.const = n
$\theta_{[n]} = n$

APPENDIX B. Translation Routines and Translation Invariants

Translation routines and translation invariants for the example compiler implementation are given in this appendix.

The translation environment consists of the data structures whose names, types, and uses are given below:

C: O-CODE	array of target machine instructions containing the object code generated during compilation;
k: N	index to C which points to the instruction most recently generated;
M: E	indexed sequence of identifiers, used as a symbol table, where E = Id*;
m: N	index to M which points to the most recently declared active identifier;
V: $(N \cup \text{Id})^*$	auxiliary translation stack of object program addresses, symbol table markers, and identifiers;
v: N	index to V which points to the "top" element of V;
b, pos: N	auxiliary variables.

In the translation invariants below, $\mathbf{u} \in U$, where

$$\mathbf{u} = \langle C, k, M, m, V, v, b, pos \rangle$$
$$U = \text{O-CODE} \times N \times E \times N \times (N \cup \text{Id})^* \times N \times N \times N.$$

A translation invariant Q_B is *similar to* a translation invariant Q_A if Q_B is the result of substituting all occurrences of A in Q_A by B.

π_1 = prog → block

```
c1: k := k+1; C[k] := (HALT)
```

$$Q_{\text{prog}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (I_0(\mathbf{u}_0) \supset C[1{:}k] = \gamma),$$
where $I_0(\mathbf{u}_0) \leftrightarrow (k_0 = 1 \;\&\; C_0[1] = (\text{START}) \;\&\; v_0 = 0 \;\&\; m_0 = 0)$.

π_2 = block → BEGIN dcl; body END

```
c2: k := k+1; C[k] := (END, b);
    m := m-b;
    b := V[v]; v := v-1
```

$$Q_{\text{block}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (M[1{:}m] = M_0[1{:}m_0] \;\&\; k_0 < k \;\&\; V[1{:}v] = V_0[1{:}v_0] \;\&\; b = b_0 \;\&\; C[1{:}k] = C_0[1{:}k_0] \bullet \gamma(k_0 + 1, M_0[1{:}m_0]))$$

π_3 = dcl → NEW id

```
c3: k := k+1; C[k] := (NEW);
    m := m+1; M[m] := V[v];
    V[v] := b; b := 1
```

π_4 = dcl → dcl, id

```
c4: SEARCH(m-b+1, m);
    IF pos=0
    THEN k := k+1; C[k] := (NEW);
         m := m+1; M[m] := V[v];
         b := b+1
    ELSE SKIP
    FI;
    v := v-1
```

$$Q_{\text{dcl}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (M[1{:}m] = M_0[1{:}m_0] \bullet (\gamma)_1 \;\&\; m = m_0 + b \;\&\; C[1{:}k] = C_0[1{:}k_0] \bullet (\gamma)_2 \;\&\; V[1{:}v] = V_0[1{:}v_0] \bullet \langle b_0 \rangle)$$

π_5 = body → body; stm

π_6 = body → stm

```
c5: c6: ;  /* empty routines */
```

Q_{body} is similar to Q_{block}.

π_7 = stm → block

```
c7: ; /* empty routine */
```

π_8 = stm → id := exp

```
c8: SEARCH(1, m);
    k := k+1; C[k] := IF pos > 0
                      THEN (ST, pos)
                      ELSE (ERR, 8)  /* undeclared id */
                      FI;
    v := v-1
```

π_9 = stm → ifcl tpart epart

```
c9: C[V[v]] := (JMP, k+1);  /* backpatch */
    C[V[v-1]] := (JPF, V[v]+1);  /* backpatch */
    v := v-2  /* pop pointers to backpatched areas */
```

π_{10} = stm → test DO body END

```
c10: C[V[v]] := (JPF, k+2);  /* backpatch */
     k := k+1; C[k] := (JMP, V[v-1]);  /* loop back */
     v := v-2  /* pop pointers */
```

Q_{stm} is similar to Q_{block}.

π_{11} = ifcl → IF cond THEN

π_{12} = tpart → body ELSE

```
c11: c12: k := k+1;  /* save space for jump */
          v := v+1; V[v] := k;  /* push pointer
                                   to saved space */
```

$$Q_{\text{ifcl}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (M[1{:}m] = M_0[1{:}m_0] \;\&\; k_0 < k-1 \;\&\; V[1{:}v] = V_0[1{:}v_0] \bullet \langle k \rangle \;\&\; b = b_0 \;\&\; C[1{:}k-1] = C_0[1{:}k_0] \bullet \gamma(k_0+1, M_0[1{:}m_0]))$$

Q_{tpart} is similar to Q_{ifcl}.

π_{13} = epart → body FI

```
c13: ;  /* empty */
```

Q_{epart} is similar to Q_{block}.

π_{14} = test → while cond

```
c14: k := k+1;  /* save space for jump */
     v := v+1; V[v] := k;  /* push pointer to saved space */
     /* c14 is the same as c11 and c12 */
```

$$Q_{\text{test}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (M[1{:}m] = M_0[1{:}m_0] \;\&\; k_0 < k-1 \;\&\; V[1{:}v] = V_0[1{:}v_0] \bullet \langle k_0+1 \rangle \bullet \langle k \rangle \;\&\; b = b_0 \;\&\; C[1{:}k-1] = C_0[1{:}k_0] \bullet \gamma(k_0+1, M_0[1{:}m_0]))$$

π_{15} = while → WHILE

```
c15: v := v+1; V[v] := k+1;
     /* push location of beginning of loop */
```

$$Q_{\text{while}}(\gamma)(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (C[1{:}k] = C_0[1{:}k_0] \;\&\; M[1{:}m] = M_0[1{:}m_0] \;\&\; V[1{:}v] = V_0[1{:}v_0] \bullet \langle k_0+1 \rangle \;\&\; b = b_0)$$

π_{16} = exp → exp aop term

```
c16: k := k+1; C[k] := (AOP)
```

π_{17} = exp → term

```
c17: ;  /* empty */
```

Q_{exp} is similar to Q_{block}.

π_{18} = term → term mop pri

```
c18: k := k+1; C[k] := (MOP)
```

π_{19} = term → pri

```
c19: ;  /* empty */
```

Q_{term} is similar to Q_{block}.

π_{20} = pri → id

```
c20: SEARCH(1, m);
     k := k+1; C[k] := IF pos > 0
                       THEN (LD, pos)
                       ELSE (ERR, 20)  /* undeclared id */
                       FI;
      v := v-1
```

π_{21} = pri → const

```
c21: k := k+1; C[k] := (LDI, V[v])
     v := v-1
```

π_{22} = pri → (exp)

```
c22; ;  /* empty */
```

Q_{pri} is similar to Q_{block}.

π_{23} = cond → exp rop exp

```
c23: k := k+1; C[k] := (ROP)
```

Q_{cond} is similar to Q_{block}.

$\pi_{[x]}$ = id → x, $x \in$ Id

```
c[x]: v := v+1; V[v] := x
```

$$Q_{\text{id}}(\theta_{[x]})(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (C[1{:}k] = C_0[1{:}k_0] \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \bullet \langle x \rangle \,\&\, b = b_0)$$

$\pi_{[n]}$ = const → n, $n = 0, 1, 2, \cdots$

```
c[n]: v := v+1; V[v] := n
```

$$Q_{\text{const}}(\theta_{[n]})(\mathbf{u}_0, \mathbf{u}) \leftrightarrow (C[1{:}k] = C_0[1{:}k_0] \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \bullet \langle n \rangle \,\&\, b = b_0)$$

SEARCH is characterized by

$$(i = i_0 \,\&\, j = j_0 \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \,\&\, b = b_0)$$
$$\{\text{SEARCH}(i, j)\}$$
$$(pos = \text{loc}(V[v], M[i{:}j]) \,\&\, i = i_0 \,\&\, j = j_0 \,\&\, M[1{:}m] = M_0[1{:}m_0] \,\&\, V[1{:}v] = V_0[1{:}v_0] \,\&\, b = b_0).$$

ACKNOWLEDGMENTS

The authors wish to sincerely thank the referees for their diligent and critical reading of this paper, which resulted in significant improvement of its presentation and technical accuracy.

REFERENCES

1. AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
2. APT, K. Ten years of Hoare's logic: A survey—Part I. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct. 1981), 431–483.
3. BOCHMANN, G. V., AND WARD, P. Compiler writing system for attribute grammars. *Comput. J. 21*, 5 (May 1978), 144–148.
4. BROSGOL, B. M. Deterministic translation grammars. TR 3-74, Center for Research in Computing Technology, Harvard Univ., 1974.

5. BURSTALL, R. M., AND LANDIN, P. J. Programs and their proofs: An algebraic approach. In *Machine Intelligence 4*, D. Michie, Ed., American Elsevier, New York, 1969, 17–43.
6. CHIRICA, L. M. Contributions to compiler correctness. UCLA-ENG-7697, UCLA Computer Science Dept., Oct. 1976.
7. CHIRICA, L. M., AND MARTIN, D. F. An order-algebraic definition of Knuthian semantics. *Math. Syst. Theor. 13* (1979), 1–27.
8. GOGUEN, J. A., ET AL. Initial algebra semantics and continuous algebras. *J. ACM 24*, 1 (Jan. 1977), 68–95.
9. GORDON, M. J. C. *The Denotational Description of Programming Languages*. Springer Verlag, New York, 1979.
10. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 19*, 10 (Oct. 1969), 576–580.
11. KNUTH D. E. Semantics of context-free languages. *Math. Syst. Theor. 2* (1968), 127–145.
12. LIGLER, G. Surface properties of programming language constructs. In *Proceedings IRIA Symposium on Proving and Improving Programs* (Arc-et-Senans, July, 1975), IRIA, 299–323.
13. MCCRACKEN, D. D. *A Guide to PL/M Programming for Microcomputer Applications*. Addison-Wesley, Reading, Mass., 1978.
14. MCKEEMAN, W. M., HORNING, J. J., AND WORTMAN D. B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
15. MILNER, R., AND WEYRAUCH, R. Proving compiler correctness in a mechanized logic. In *Machine Intelligence 7*, B. Meltzer and D. Michie, Eds, American Elsevier, New York, 1972, 51–70.
16. MORRIS, F. L. Correctness of programming languages—An algebraic approach. STAN-CS-72-303, Stanford Computer Science Dept., Aug. 1972.
17. MORRIS, F. L. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages* (Boston, Oct. 1973), ACM, New York, 144–152.
18. POLAK, W. *Compiler Specification and Verification. Lecture Notes in Computer Science, 124*, Springer Verlag, New York, 1981.
19. THATCHER, J., ET AL. More on advice on structuring compilers and proving them correct. *Theor. Comput. Sci. 15* (Sept. 1981), 223–249.
20. WIRTH, N., AND WEBER, H. EULER: A generalization of ALGOL and its formal definition. *Commun. ACM 9*, 2 (Feb. 1966), 89–99.

Received October 1982; revised January 1985; accepted August 1985