Nanopass Attribute Grammars

Nathan Ringo
ringo025@umn.edu
University of Minnesota
USA

Lucas Kramer
krame505@umn.edu
University of Minnesota
USA

Eric Van Wyk
evw@umn.edu
University of Minnesota
USA

Abstract

Compilers for feature-rich languages are complex; they perform many analyses and optimizations, and often lower complex language constructs into simpler ones. The nanopass compiler architecture manages this complexity by specifying the compiler as a sequence of many small transformations, over slightly different, but clearly defined, versions of the language that each perform a single straightforward action. This avoids errors that arise from attempting to solve multiple problems at once and allows for testing at each step.

Attribute grammars are ill-suited for this architecture, primarily because they cannot identify the many versions of the language in a non-repetitive and type-safe way. We present a formulation of attribute grammars that addresses these concerns, called nanopass attribute grammars, that (*i*) identifies a collection of all language constructs and analyses (attributes), (*ii*) concisely constructs specific (sub) languages from this set and transformations between them, and (*iii*) specifies compositions of transformations to form nanopass compilers. The collection of all features can be statically typed and individual languages can be checked for well-definedness and circularity. We evaluate the approach by implementing a significant subset of the Go programming language in a prototype nanopass attribute grammar system.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**.

Keywords: attribute grammars, compilers, nanopass compilers, software engineering

ACM Reference Format:
Nathan Ringo, Lucas Kramer, and Eric Van Wyk. 2023. Nanopass Attribute Grammars. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623514

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623514

1 Introduction

Modern programming languages require sophisticated compilers. Feature-rich languages have many constructs, and a compiler typically performs several semantic analyses, optimizations, and transformations on programs. It can also be difficult to observe the behavior of different aspects of the compiler and test the results.

One approach to dealing with these complexities is to design the compiler as a sequence of several (perhaps dozens) small, clearly-defined tasks on programs in clearly-identified versions of the language. Compilers with this design are called *nanopass compilers* [18]. These tasks may be quite simple syntactic transformations such as reducing all if-then constructs in an imperative language into if-then-else constructs in which the else-clause is a skip statement. Other *lowering* transformations may replace list comprehensions with higher-order function calls or replace loops with `goto`s. These simple steps transform programs into simpler and smaller versions of the language, each one known to *not* contain the constructs that are transformed away. Other transformations may be more complex, such as type-checking in order to annotate expressions with their types; yet others may perform optimizations such as common sub-expression elimination. Some passes condense the source language down to language variations more suitable for translation. For example, a compiler may replace expressions in which binary operators are nested, as in `x + (y * z)`, with a sequence of operations that only allow atomic expressions such as variables and value literals as arguments to binary operators, as in `let temp1 = y * z in x + temp1`. Eventually, the steps transform the program to code that can be translated directly to a low-level intermediate language or to assembly, since it contains goto-statements and simple expressions.

This approach has been used successfully in both educational [19] and industrial [9] contexts. The proponents of nanopass compilers claim several benefits. The primary one is that each step is small and easy to understand. Because various language versions are clearly specified as the input and output of different passes, one can ensure that, *e.g.*, certain constructs have in-fact been transformed away and need not be considered again. Each step is also more amenable to testing as the output of each step can be inspected.

This paper adapts attribute grammars (AGs) to the nanopass approach. AGs were first specified by Knuth [10] in 1968 and are a convenient formalism for specifying computations over syntax trees. They work by decorating tree nodes

with semantic information called attributes. *Synthesized* attributes propagate information up the tree, *e.g.* types on expressions. *Inherited* attributes propagate information down the tree, *e.g.* typing contexts on statements and expressions. Well-definedness analyses can ensure that all the equations needed to specify attribute values are present and are non-circular [10, 15, 16]. This provides a strong static exhaustiveness check that all language constructs have a specification for each semantic analysis. They have been extended over the years in a variety of ways, most prominently to support *higher-order* attributes [25] so that syntax trees may be passed as attributes, and *reference* [5] or *remote* [1] attributes that allow referring to remote nodes in the syntax tree.

Despite supporting a modular approach to language implementation, attribute grammars are not well-suited for the nanopass approach. The primary problem is that the context-free grammar in an attribute grammar defines a single complete language and there is no convenient mechanism in the formalism for defining a family of languages that differ in various ways such that trees in (different) input and output languages can both be safely constructed. One can, of course, define an AG for each of the dozens of different languages that arise in a nanopass compiler but this would involve the significant duplication of many grammar productions that appear in many of the language versions. The alternative is to abandon the well-definedness analyses and define equations for a particular task or transformation on only the relevant subset of productions for which that task occurs. But doing so is quite unsatisfactory.

The primary contribution of the paper is to close this gap between nanopass compilers and attribute grammars by providing a formulation of *nanopass attribute grammars* such that the various languages and their attributes can be both conveniently defined and checked for type-correctness, well-definedness, and absence of circular attribute specifications. A nanopass attribute grammar consists of 3 components:

1. $\mathcal{E}$: the collection of language elements. This is in the form of an attribute grammar from whose components different languages are to be constructed.
2. $\mathcal{L}$: the family of languages that are transformed between. A language $L \in \mathcal{L}$ is an AG that has a subset of the nonterminals, productions, attributes, etc. found in $\mathcal{E}$.
3. C: the composition of transformations into a nanopass compiler. This maps programs in the original language into some target form.

The collection of language elements $\mathcal{E}$ is statically type checked to ensure, for example, all productions are applied to the correct number of correctly-typed arguments. However, this specification may not be well-defined (some productions are intentionally missing equations for some attributes not relevant to them), and it is not meant to be used on its own.

Languages in $\mathcal{L}$ are attribute grammars and identify steps in the compilation process. They correspond to the different languages in a nanopass compiler. Terms in a language can be annotated with attribute values during construction so that they may be used, instead of recomputed, by the transformation. For example, a type-checking transformation will produce programs, if they are well-typed, that have annotations on expressions indicating their type. The annotations are just attributes that need not be computed but exist on the tree directly. Transformations are defined by *transform attributes* and correspond to passes in a nanopass compiler. The framework can be instantiated with different varieties of attribute grammars as well as different attribute evaluation mechanisms.

A second contribution is two mechanisms for *concisely* constructing the languages in $\mathcal{L}$ that maintains their type-correctness established in $\mathcal{E}$. The first specifies a language "from scratch" by identifying the productions and attribute occurrences on nonterminals to include; all other aspects, such as nonterminals in the grammar and equations for attributes are determined from the desired productions and occurrences. The second specifies a new language by extending an existing language by adding or removing components. Since each language in $\mathcal{L}$ will include different productions and attribute equations, the well-definedness and circularity analyses need to be performed on a per-language basis.

We also evaluate this notion of nanopass attribute grammars by implementing a prototype nanopass attribute grammar system and use it to implement a compiler for a significant subset of the Go programming language.[1]

Section 2 recalls the structure of attribute grammars before Section 3 provides the specification of nanopass approach to AGs and the $\mathcal{E}$, $\mathcal{L}$, and C components described above. Section 4 describes the prototype system realizing nanopass AGs and the Go compiler developed with it. Section 5 discusses related work; Section 6 discusses performance and attribute analysis, describes some future work, and concludes.

2 Background: Attribute Grammars

Attribute grammars are a formalism for defining the semantics of context free languages [10] by attributing semantic values to nodes in a syntax tree. An attribute grammar can be defined as a tuple $AG = (G, A, \Gamma_A, @, EQ)$ consisting of the context-free grammar G, attributes A, mappings of attributes to types (Γ_A) and to nonterminals on which they occur(@), and the set of attribute-defining equations EQ.

The grammar G is a tuple (NT, T, P, Γ_P, S). NT is a finite set of nonterminals and T is a finite set of terminals. T includes traditional token types with lexemes and primitive types, *e.g.* integers and strings. In some systems this includes

[1] Available at https://melt.cs.umn.edu and archived at https://doi.org/10.13020/h1qa-s993.

structured data such as lists or tuples. P is a finite set of production names and Γ_P is a total map from production names to their signatures. Signature elements are labeled so that equations can refer to the left and right-side elements using labels instead of positions. Thus, Γ_P maps P to signatures of the form $x_0 : X_0 ::= x_1 : \tau_1 \ldots x_n : \tau_n$ where $X_0 \in NT$ and each $\tau_i \in NT \cup T$ for $i \in \{1..n\}$. $S \in NT$ is the start symbol indicating the type of the root node of a program tree.

Attributes are specified by the set $A = A_I \cup A_S$ which can be partitioned into disjoint sets of inherited (I) and synthesized (S) attributes. Γ_A is a total map from attribute names to types in T, $\Gamma_A \subseteq ((A_I \cup A_S) \rightarrow T)$. The *occurs-on* relation $@ \subseteq A \times NT$ specifies which nonterminals an attribute decorates; $(a, X) \in @$ (written $a@X$ as shorthand) indicates that attribute a occurs on X. Note that S has no inherited attributes: $\forall a \in A, (a@S) \implies a \notin A_I$.

Equations, $EQ = \bigcup_{p \in P} EQ_p$, indicate how values of attributes are determined; each is associated with a production p in P. Each has the form $x.a = e$ where $a \in A$, x is label on the production, and e is an expression defining the value of $x.a$. We require that for any production p, no two equations in EQ_p have the same left hand side. Different attribute grammar systems put different requirements on the constructs in e, but generally, e is an expression that can refer to the values of attributes on the signature elements of p and construct and manipulate these values.

Since Knuth's original specification [10] attribute grammars have been extended in many ways. One variety extends the types that attributes may take (the range of Γ_A) and constructs in equations accordingly. Higher-order attributes [25] dramatically increase the usefulness of AGs by allowing trees to be passed around as attributes and then supplied with inherited attributes so that synthesized attributes can then be computed on them and accessed. These add NT to the range of Γ_A. Reference [5] and remote [1] attributes extend Γ_A with pointers (references) to decorated tree nodes somewhere else in the syntax tree. A common use is to allow variable uses in a program to have a reference attribute pointing to their declarations so that information such as the variable's type can be accessed on the remote declaration node.

Another form of extension provides means for more easily moving values up and down the tree. Kastens and Waite [8] alleviate so-called copy-rules for propagating information down the tree and described other mechanism for collecting information, such as diagnostic messages, up the tree. Variations on these are now common in AG systems.

An important aspect of attribute grammars, and many of their extensions, are static analyses to identify and validate the flow of information through different attributes. A *well-definedness* analysis in many systems determines, for each synthesized attribute, which inherited attributes may be needed to compute its value (sometimes called *flow-types* [17]) in order to determine if all required equations are present. This information can also be used to define a circularity analysis to check for cycles in attribute dependencies. These analyses were provided in Knuth's original formulation and are typically extended to accommodate new features, such as higher-order attributes [25].

There also a variety of mechanisms for computing the values for attributes on a tree. Ordered attribute grammars [7] and an extension of them [22] determine an order, applicable for all possible trees, for computing attributes. In contrast, the commonly-used demand-driven approach treats AGs similarly to lazy functional programs and computes attributes only as they are needed. Other approaches embed AGs in existing languages, often lazy functional ones, to write the specification directly as programs in those languages [12, 21]. Circular attribute grammars allow attribute dependencies to be circular as long as they are well-founded, providing a convenient means for specifying fix-point algorithms [3, 13].

This discussion of attribute grammars and their different variations is necessarily incomplete. In principle, these, and others, fit into the nanopass attribute grammar formalism presented below. That framework can be instantiated with different types of attributes and evaluation schemes.

3 Nanopass Attribute Grammars

In this section we describe the nanopass attribute grammar formalism: the language elements $\mathcal{E}$, the family of languages $\mathcal{L}$, and the compositions C. Sections 3.1 - 3.2 specify $\mathcal{E}$ by extending the formalism specified in Section 2, discuss what is required for it to be *well-formed*, and provide its type-checking rules. Sections 3.3 - 3.5 specify $\mathcal{L}$ and transform attributes and discuss the *language checking* process performed on each language to ensure that it is well-defined and that its transformations produce terms using only the productions in their target language. Section 3.6 describes how transformations are composed to construct a nanopass compiler.

3.1 Language elements: $\mathcal{E}$

The first part of a nanopass AG specification, denoted $\mathcal{E}$, is a collection of attribute grammar elements $(G, A, \Gamma_A, @, EQ)$ as above from which different languages will be constructed. This is extended in two ways.

First, we add a new kind of attribute, transform attributes, denoted A_T. The A component of $\mathcal{E}$ is now $A = A_I \cup (A_S \cup A_T)$. These are essentially higher-order synthesized attributes for defining transformations between languages in $\mathcal{L}$ by equations also in EQ. Transform attributes always have the same type as the nonterminal they're computed on, so they are not (and need not be) included in Γ_A. These are discussed further in Section 3.4.

The second addition is *annotations*. These annotate the tree with semantic values supplied when the tree is constructed instead of being computed during attribute evaluation. To simplify the formalism, these are specified as

attributes whose equations are ignored when it is demanded; instead, the value is the one provided when the tree was built. Note that Γ_A still contains the types of annotations.

To support annotations, expressions that appear on the right-hand-side of equations can supply them to trees that they construct. If we notate building a tree from production p with children t_1 and t_2 as $p(t_1, t_2)$, we might notate building the same tree while supplying annotation a with the value of the expression e as $p(t_1, t_2, a = e)$.

Annotations might be used to store, for example, the types of expressions. After the types have been computed (as an attribute), we will still continue to transform the program in the process of compiling it. To avoid needing to re-type-check the tree at any time the types of expressions might be needed later on, we can make the attribute an annotation.

As discussed in Section 3.3, each language in $\mathcal{L}$ will indicate which attributes in $A_I \cup A_S$ are to be treated as annotations and thus be predefined on trees in the language.

Due to how close $\mathcal{E}$ is to a standard AG specification, existing attribute evaluation strategies can work with only minor tweaks to support annotations. An attribute evaluation strategy that is aware of languages can take advantage of this to remove dependencies between equations that might otherwise exist.

If $\mathcal{E}$ satisfies the requirements mentioned above and for AG in Section 2, it is said to be *well-formed*; this is the first requirement the specifications must satisfy. Note that unlike a traditional attribute grammar specification, $\mathcal{E}$ is not expected to be well-defined. After some transformations have been applied some language constructs will have been translated into other more fundamental forms and thus no longer appear in the programs. Since later passes won't need to handle constructs that won't be present, we don't need equations on those productions for attributes that are only used after the production is eliminated.

3.2 Type checking language elements $\mathcal{E}$

Even though we cannot check well-definedness on a collection of elements $\mathcal{E}$, we can still check that they are well-typed, given some language of expressions that may appear on the right-hand side of an equation. Type checking can be done once on the language elements $\mathcal{E}$ and type-correctness will preserved for languages in $\mathcal{L}$ when they are constructed using one of the two methods described in Section 3.3. Some rules for a reasonably standard type system adapted for a NAG system are shown in Figure 1.

The type-correctness of a well-formed $\mathcal{E}$ is satisfied when for all productions $p \in P, P \in AG$, all equations $x.a = e$ in $EQ_p, EQ_p \in AG$ type check, as indicated by the judgment

$$p \vdash (x.a = e) \textbf{ Tok}$$

This judgment in turn refers to a traditional typing judgment for expressions,

$$p \vdash e : \tau$$

In both cases, $p \in P$ acts as a context, providing the types of children. The components of AG are also ambiently present as the global context and referred to by names used above. Thus typing contexts such as Γ_A and Γ_P, the occurs-on relation @, and other aspects of AG can be used in the type checking rules. Synthesized and inherited equations have typical typing rules, T-INH-EQ and T-SYN-EQ, ensuring that attributes occur properly and the type of the expression matches that of the attribute. Following these are 3 sample rules for typing expressions. Of more interest are transform attributes; their equations are typed by T-TRANSFORM-EQ, which ensures that the type of the equation's right-hand side matches the production's left-hand side. Their access is typed similarly by T-TRANSFORM-ACCESS.

The T-PROD rule for constructing trees is more general than a typical one, since it needs to handle annotations. Note that type-checking does not check that only the annotations that should occur do occur. In $\mathcal{E}$ we do not know whether an attribute will be treated as an annotation or as an attribute to be computed. This check happens in the language-correctness checks described in Section 3.5.

3.3 Languages: $\mathcal{L}$

A nanopass AG also consists of a *family of languages*, and *transformations* between them, $\mathcal{L}$. Each language uses a subset of the grammatical and semantical features found in $\mathcal{E}$. They will use some productions and some attributes to define a language with only the desired syntax and semantics. We discuss two convenient and concise mechanisms to identify what these languages consist of so that the type-correctness established once on $\mathcal{E}$ is maintained on each language and need not be checked again.

3.3.1 Languages.

The productions, attributes, associated equations, etc., are all specified in $\mathcal{E}$ and may (or may not) be used in different languages and thus languages have (nearly) the same structure as the collection of language elements $\mathcal{E}$. We will superscript language elements by the language names and also superscript the overarching language components elements by $\mathcal{E}$. When the context is clear we will drop these superscripts.

A language $L \in \mathcal{L}$ is a 6-tuple containing:

- $G^L = (NT^L, T^L, P^L, \Gamma_P^L, S^L)$ where $NT^L \subseteq NT^{\mathcal{E}}$, $T^L = T^{\mathcal{E}}$, $P^L \subseteq P^{\mathcal{E}}$, $\Gamma_P^L \subseteq \Gamma_P^{\mathcal{E}}$, and $S^L = S^{\mathcal{E}}$.
- $A^L = A_I^L \cup (A_S^L \cup A_T^L)$, where $A^L \subseteq A^{\mathcal{E}}$
- $@^L \subseteq @^{\mathcal{E}}$
- $\Gamma_A^L \subseteq \Gamma_A^{\mathcal{E}}$
- $EQ^L = \cup_{p \in P^L} EQ_p^L$ where $EQ_p^P \subseteq EQ^{\mathcal{E}}{}_p$
- $@_A^L \subseteq @^{\mathcal{E}}$ where $@^L \cap @_A^L = \varnothing$

The first five components correspond directly to their equivalents in $\mathcal{E}$. Note that terminals and primitive values T^L and T are not scoped to a particular language and are

T-INH-EQ
$$\frac{a \in A_I \qquad a@X \qquad \Gamma_P(p) = (\ldots ::= \ldots\ x : X\ \ldots) \qquad p \vdash e : \Gamma_A(a)}{p \vdash (x.a = e)\ \textbf{Tok}}$$

T-SYN-EQ
$$\frac{a \in A_S \qquad a@X \qquad \Gamma_P(p) = (x : X ::= \ldots) \qquad p \vdash e : \Gamma_A(a)}{p \vdash (x.a = e)\ \textbf{Tok}}$$

T-LHS
$$\frac{p \in P \qquad \Gamma_P(p) = (x : X ::= \ldots)}{p \vdash x : X}$$

T-RHS
$$\frac{p \in P \qquad \Gamma_P(p) = (\ldots ::= \ldots\ x : \tau\ \ldots)}{p \vdash x : \tau}$$

T-INHSYN-ACCESS
$$\frac{a \in (A_I \cup A_S) \qquad a@X \qquad p \vdash e : X \qquad \Gamma_A(a) = \tau}{p \vdash e.a : \tau}$$

T-TRANSFORM-EQ
$$\frac{a \in A_T \qquad a@X \qquad \Gamma_P(p) = (x : X ::= \ldots) \qquad p \vdash e : X}{p \vdash (x.a = e)\ \textbf{Tok}}$$

T-TRANSFORM-ACCESS
$$\frac{a \in A_T \qquad a@X \qquad p \vdash e : X}{p \vdash e.a : X}$$

T-PROD
$$\frac{\Gamma_P(p') = (x_0 : X_0 ::= x_1 : \tau_1\ \ldots\ x_m : \tau_m) \qquad \underset{1 \le i \le n}{\forall} p \vdash e_i : \tau_i \qquad \underset{m < i \le n}{\forall} a_i@X \qquad \underset{m < i \le n}{\forall} \Gamma_A(a_i) = \tau_i}{p \vdash p'(e_1, \ldots, e_m, a_{m+1} = e_{m+1}, \ldots, a_n = e_n) : X_0}$$

Figure 1. Typing rules for $\mathcal{E}$.

available in all languages. This is done to simplify the presentation, but could be accommodated without much effort. Note that in all languages, the start symbol is the same $S^{\mathcal{E}}$.

The sixth component, $@_A^L$, describes the annotations that are present on each nonterminal. Values for these are provided when the L tree is constructed. Recall the use-case of saving the results of type-checking expressions. We assume the computed type is ty, the nonterminal for expressions is E, the language in which type-checking is performed is L_0, and the language in which the ty attribute is an annotation is L_1. In this case, both $ty \in A_S^{L_0}$ and $ty \in A_S^{L_1}$, but while $ty\ @^{L_0}\ E$, instead $ty\ @_A^{L_1}\ E$.

Identifying languages. A language is simply a subset of $\mathcal{E}$ along with an indication of annotations $@_A$. Identifying a specific language $L \in \mathcal{L}$ by enumerating all of the elements would be quite tedious and also open to errors from leaving out required components. For example, L might not be well-formed if the production signature map Γ_P^L does not have a signature for a production in P^L. It might also not be type-correct if in the expression for an equation in EQ^L references an attribute that is not found in A^L, $@^L$, or Γ_A^L. To avoid these problems we provide two mechanisms for specifying languages that are both concise and also result in type correct languages.

The first mechanism identifies a language L *directly*. It requires only the enumeration of production names (P^L) that are to be used and the desired occurrences of attributes ($@^L$) and annotations ($@_A^L$) to be used. All other elements of L can be inferred from these. For each production name $p \in P^L$, we include its signature in Γ_P^L. Any nonterminal appearing in Γ_P^L is added to NT^L, and $S^L = S^{\mathcal{E}}$. Thus G^L is well-formed. From the attribute and annotation occurrence relation elements identified we populate the sets of attribute A_I^L, A_S^L, and A_T^L. Also, any nonterminal X in $@^L$ or in $@_A^L$ is added to NT^L if not already there. Similarly, appropriate equations are selected for attributes in $@^L$. The equation $x.a = e$ from $EQ_P^{\mathcal{E}}$ is included in EQ_p^L when $p \in P^L$ and $a \in A^L$. Equations for annotations are *not* included in EQ^L. Γ_A^L is Γ_A restricted to attributes in A_S^L and A_I^L.

Why design things in this way? Recall the two types of expression productions discussed in Section 1: one allowed (nested) expressions as children; the other allowed only atomic children of variables and literals. The source language for the transformation that rewrites nested expressions into atomic ones needs to identify only the complex expression productions. These are the expressions that may be used to construct a tree that is input to this transformation. We would not want to include the atomic expression productions in this collection because that would indicate that they could also be used to form input terms. Doing so would not allow us to ensure that input and output languages are of the proper form. The *language checking* process described in Section 3.5 will ensure that expressions only generate trees in the appropriate language and that attributes access are in fact in the language. This check, along with ensuring that all the required equations are present, is done in the language-checking process. Specifically, see the discussion of the language-checking rule L-PROD there.

The second mechanism for identifying a language L does so by *extension*. It uses an *extends* mechanism that creates a new language by identifying elements to add to, or remove from, an existing language. Formally, a language L identified this way is specified as:

- a language L', perhaps also defined as an extension,
- a set of production names P^{L+} to add to $P^{L'}$
- a set of production names P^{L-} to remove from $P^{L'}$
- a set of occurrences $@^{L+}$ to add to $@^{L'}$

- a set of occurrences $@^{L-}$ to remove from $@^{L'}$
- a set of occurrences $@^{L+}_A$ to add to $@^{L'}_A$
- a set of occurrences $@^{L-}_A$ to remove from $@^{L'}_A$

From these, we can compute:

- $P^L = (P^{L'} \cup P^{L+}) - P^{L-}$
- $@^L = (@^{L'} \cup @^{L+}) - @^{L-}$
- $@^L_A = (@^{L'}_A \cup @^{L+}_A) - @^{L-}_A$

From here, the other elements of L are inferred using the same process as described above for creating language directly from a set of productions and occurrences. Similarly, the resulting language is well-formed and well-typed in the same manner.

3.4 Transform attributes

Transform attributes play a key role in nanopass attribute grammars, as they define the transformations from one language into the next. A transform attribute $a_T \in A_T$ is defined as having a source language $\Gamma_S(a_T) = L_S$ and a target language $\Gamma_T(a_T) = L_T$. Transform attributes have equations similar to those used for higher-order attributes.

For many transformations, the computation for many productions in the language is to simply apply the transformation to the child trees and re-build the tree with the same production and the transformed child trees. It would be quite inconvenient to write these directly and thus they are inferred when an explicit equation is not provided for a production. Consider a production *selector* for selecting a field from a record with signature *e*:*Expr* ::= *l*:*Expr r*:*String* with no explicit equation for the transformation a_T. When the target language has no annotations, the generated equation would be $e.a_T = selector(l.a_T, r)$. The same production is used, and the a_T attribute is computed on each child of nonterminal type. Children that are not of nonterminal type are passed as-is.

If the production has annotations in the target language, the generated equation copies them over from the attributes and annotations in the source language. The a_T attribute is recursively applied if the annotation has a nonterminal type — this is the same process as performed on the children. For example, if in the target language of a_T, *Expr* has two annotations, *isInLambda*:*Boolean* and *type*:*Type*, the generated equation would be:

$$e.a_T = selector(l.a_T, r, isInLambda = e.isInLambda, type = e.type.a_T)$$

Note that the $e.type.a_T$ call would require that there are no non-annotation inherited attributes on the *Type* nonterminal in a_T's source language, since additional attribute equations are not supplied here.

In the general case, the generated equation for a production p with signature $e{:}X_e ::= x_0{:}X_0\ x_1{:}X_1\ \ldots$ would be $e.a_T = p(x_0[.a_T], ..., a_0 = this.a_0[.a_T], ...)$, where the $[.a_T]$ represents the attribute only being demanded on children, attributes, and annotations of nonterminal type.

Note that this equation is generated only if:

- $p \in P^{L_T}$,
- $@^{L_S} \cup @^{L_S}_A \subseteq @^{L_T}_A$, and
- $a@^{L_S}_A X \implies \nexists a_I \in A_I$ such that $a_I@^{L_S}\Gamma_A(a)$

If these conditions are not met, the programmer is required to explicitly provide an equation. Consider the transformation replacing *if c then s* with *if c then s else skip*. Since the if-then production is not in the target language an explicit equation is required. Likewise, if the attributes and annotations in the source are not enough to define the annotations in the target then equations are required. Finally, as noted, we require that annotations on the source language do not need inherited attributes on that type.

The process of generating an equation for a transform attribute always results in a well-typed equation. If the rest of $\mathcal{E}$ passed type-checking, we know the entire specification is well-typed, and that evaluation of attributes will not result in runtime type errors.

3.5 Language checking

A language $L \in \mathcal{L}$ is language-correct if every equation in EQ^L evaluates to a term in the appropriate language. Recall that synthesized and inherited attributes produce terms in the same language as the language of the term the attributes are computed on, while transform attributes produce terms in the attribute's target language.

If $\mathcal{E}$ is type correct, then the languages $L \in \mathcal{L}$ will, if constructed using the methods described above, also be type correct and will not experience typical evaluation-time type errors. Some concerns however must be checked on the individual languages and their transform attributes. These checks ensure that a language L is *language-correct* and that the evaluation-time errors listed below do not occur.

- L is missing an equation for an attributes in $A^L_S \cup A^L_I$ for production in P^L that may be demanded during evaluation.
- L computes a tree for a higher-order attribute in A^L_S or A^L_I that is not in the language of L. That is, a production not in P^L is used in the tree.
- Similarly, L computes a tree for a transform attribute $a \in A_T$ with target language L_T ($\Gamma_T(a) = L_T$) that is not in the language of L_T.
- L has an equation for a synthesized attribute $a \in A^L_S$ for production p but also defines a as an annotation in an application of the production p, as in $p(..., a = ...)$.
- L has an equation that demands the value of an inherited attribute on a tree that does not have a parent tree that can provide the defining equations. This could happen when accessing a synthesized or transform attribute (that depends on an inherited attribute) on

a tree *stored as a higher-order attribute* since that tree has no parent.

The language-correctness of a language L is indicated by the judgment $L \vdash \textbf{Lok}$. By definition, $L \vdash \textbf{Lok}$ is established if for all nonterminals $X \in NT^L$, for all productions $p \in P^L$ with X on the left hand side, and for all attributes $a \in (A_S \cup A_I)$ where $a@^L X$ the following conditions hold:

- $(x.a = e) \in EQ_p^L$: ensuring that the attribute grammar is complete and no equations are missing, and
- $p, L \vdash (x.a = e)$ **Lok**, ensuring L is language-correct.

Completeness is a conservative analysis and it is often more convenient to require that only the equations that would ever be needed in an attribute evaluation are present. In Section 6 we discuss why this conservative analysis is less of a concern in nanopass attribute grammars than in traditional higher-order attribute grammars. This second requirement could be replaced by a less-conservative analysis if desired.

This language-correctness analysis assumes that type-checking has already been performed. It is specified as a collection of inference rules, as was done with type checking. It is organized as a pair of judgments, one for equations and one for expressions:

$$p, L \vdash (x.a = e)\ \textbf{Lok} \quad \text{and} \quad p, L, L_T \vdash e\ \textbf{Lok}$$

These judgments are parameterized by languages. L is the language of the tree the attribute is being computed on. It is constant throughout the analysis. L_T is the language the expression computing a tree should evaluate to: the source language L for inherited and synthesized attributes and the target language L_T for transform attributes. Some of the rules establishing these judgments are given in Figure 2.

The rule L-INHSYN-EQ for synthesized and inherited equations ensures that the expression of the equation will compute a tree in the source language L. The rule L-TRANSFORM-EQ for transform equations checks the expression e using the target language L_T to ensure that trees used in a_T are in L_T. A node and its children are always in the same language as the attribute is being computed on. This allows accessing attributes that are defined on the source language.

Accessing a synthesized or inherited attribute (or an annotation) preserves the language of the term it is computed on, since we would expect the results of analyses on a term to be in the term's own language. It also requires that the attribute be present in the language. To ensure all attribute accesses are well-defined, the access may be on a name from the signature (L-INHSYN-SIG-ACCESS) where, due to the completeness check, all attributes in the language have equations defined. The access could also be of an annotation on any expression (L-ANNO-ACCESS), since the annotation must have been supplied when the term was constructed. A synthesized attribute can also be accessed on an arbitrary expression (L-SYN-NODEPS-ACCESS), for example on a higher-order attribute,

L-INHSYN-EQ
$$\frac{a \in (A_I^L \cup A_S^L) \qquad p, L, L \vdash e\ \textbf{Lok}}{p, L \vdash (x.a = e)\ \textbf{Lok}}$$

L-TRANSFORM-EQ
$$\frac{a \in A_T^L \qquad p, L, L_T \vdash e\ \textbf{Lok} \qquad \Gamma_S^L(a) = L \qquad \Gamma_T^L(a) = L_T}{p, L \vdash (x.a = e)\ \textbf{Lok}}$$

L-LHS
$$\frac{\Gamma_P^L(p) = (x{:}X ::= \ldots)}{p, L, L \vdash x\ \textbf{Lok}}$$

L-RHS
$$\frac{\Gamma_P^L(p) = (\ldots ::= \ldots\ x{:}\tau\ \ldots)}{p, L, L \vdash x\ \textbf{Lok}}$$

L-INHSYN-SIG-ACCESS
$$\frac{x{:}X \in \Gamma_P^L(p) \qquad a \in (A_I^L \cup A_S^L) \qquad a@^L X}{p, L, L_T \vdash x.a\ \textbf{Lok}}$$

L-ANNO-ACCESS
$$\frac{a \in (A_I^L \cup A_S^L) \qquad a@_A^L X \qquad p \vdash e{:}X \qquad p, L, L_T \vdash e\ \textbf{Lok}}{p, L, L_T \vdash e.a\ \textbf{Lok}}$$

L-SYN-NODEPS-ACCESS
$$\frac{a \in A_S^L \qquad a@^L X \qquad \underset{a'@^L X}{\forall} a' \notin A_I^L \qquad p \vdash e{:}X \qquad p, L, L_T \vdash e\ \textbf{Lok}}{p, L, L_T \vdash e.a\ \textbf{Lok}}$$

L-TRANSFORM-SIG-ACCESS
$$\frac{a \in A_T \qquad a@^L X \qquad x{:}X \in \Gamma_P^L(p) \qquad \Gamma_S(a)^L = L \qquad \Gamma_T(a)^L = L_T}{p, L, L_T \vdash x.a\ \textbf{Lok}}$$

L-TRANSFORM-NODEPS-ACCESS
$$\frac{a \in A_T \qquad a@^L X \qquad \underset{a'@^L X}{\forall} a' \notin A_I^L \qquad \Gamma_S(a)^L = L_S \qquad \Gamma_T(a)^L = L_T \qquad p \vdash e{:}X \qquad p, L, L_S \vdash e\ \textbf{Lok}}{p, L, L_T \vdash e.a\ \textbf{Lok}}$$

L-PROD
$$\frac{p' \in P^{L_T} \qquad \Gamma_P(p') = (X ::= \ldots) \qquad \underset{0 \le i \le n}{\forall} p, L, L_T \vdash e_i\ \textbf{Lok} \qquad \{a_{m+1}, \ldots, a_n\} = \{a \mid a@_A^L X\}}{p, L, L_T \vdash p'(e_0, \ldots, e_m, a_{m+1} = e_{m+1}, \ldots, a_n = e_n)\ \textbf{Lok}}$$

L-PRIM
$$\frac{p \vdash e{:}\tau \qquad p, L, L_e \vdash e\ \textbf{Lok} \qquad \tau \notin NT \qquad L_e \in \mathcal{L}}{p, L, L_T \vdash e\ \textbf{Lok}}$$

L-IF
$$\frac{p, L, L_T \vdash c\ \textbf{Lok} \qquad p, L, L_T \vdash t\ \textbf{Lok} \qquad p, L, L_T \vdash e\ \textbf{Lok}}{p, L, L_T \vdash (\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e)\ \textbf{Lok}}$$

Figure 2. Some language correctness rules for $L \in \mathcal{L}$.

when there are no inherited attributes in the language occurring on the type, as the synthesized attribute cannot possibly depend on any inherited attributes.

Accessing a transform attribute produces a term in the transform attribute's target language, from a term in the transform attribute's source language. The rules L-TRANSFORM-SIG-ACCESS and L-TRANSFORM-NODEPS-ACCESS inspect the languages L and L_T in the context of the rules. Similar rules exist for transform attributes accessed as annotation ($a@^L_A X$) but this is rarely done.

Constructing a term with a production entails more checks. The rule L-PROD checks that the production is valid in the language L_T, that its subterms are valid in that language, and that it has exactly the set of annotations it should have to be in that language.

Most of the time, L and L_T are the same language in equations for inherited or synthesized attributes (L-INHSYN-EQ), since these attributes' values are in the same language as the tree they are computed on. In contrast, transform equations (L-TRANSFORM-EQ) require that L and L_T are the source and target languages of the transform attribute.

The last issue to address is in regards to primitive types in T. They don't belong to any language, so we want to leave them unconstrained with respect to languages. Consider the equation computing a term for a transform attribute a_T using production $p \in L_T$ that uses an Boolean annotation value computed on a tree in the source language ($x.a_S$):

$$x.a_T = p(ann = x.a_S)$$

Since the Boolean value is, by definition, in the language of L_T we do not care about the languages of the trees involved in computing it. We can add the L-PRIM rule to handle this case. Note that it leaves the target language for checking e to be unconstrained: L_e can be any language in $\mathcal{L}$. [2]

With L-PRIM in place, the rules for expressions of base type can be trivial, since they can belong to any language. The only feature of note is in the rule L-IF since the *then* and *else* branches of *if* expressions must have the same L_T as the expression unless they are base types (in which case the L-PRIM rule applies). We elide similar rules for other base type expressions such as addition or numeric literals.

3.6 Composition on nanopasses: C

A nanopass attribute grammar will also specify at least one composition of a desired set of transformations, typically to lower the source language down to a version that can be easily translated to some target language. For example, we may lower an imperative language with various control flow mechanisms and expressions over various types down to a version that only uses labels and goto-statements for control flow and expressions are transformed into assignment statement sequences that directly translate into low-level intermediate code similar to assembly language instructions.

A composition, in its most primitive form, is a sequence of transform attributes $(t_0, ..., t_n)$, where each $t_i \in A_T$. This implicitly identifies the source and target languages of the overall composition; the overall source language is $\Gamma_S(t_0)$, while the overall target language is $\Gamma_T(t_n)$. In practice, a composition may check the results of on transformation before performing the next one. For example, if type errors are found on a program in a typing transformation then those errors may be output and the compilation aborted.

If $\mathcal{E}$ is type correct and the languages are all language-correct, then we can check that the compositions are also type-correct. For a composition to be type-correct the following condition must hold:

$$\forall i, i \in \{1...n\}.\ \Gamma_S(t_i) = \Gamma_T(t_{i-1})$$

This ensures that the input tree for each transformation matches the source language of the transformation. Recall that the start symbol S has no inherited attributes and thus the computation of some transform attribute t_i does not need them to be specified. If additional information is needed by in the computation of a transform attribute, then that information can be supplied as an annotation.

The output of the simple composition form above is thus a tree in $\Gamma_T(t_n)$. We can then use this tree as input to the next step in the compilation, either to compute a textual representation of a program in a target language or use higher-order attributes to construct a tree in some other language.

Multiple compositions may be defined against a single nanopass attribute grammar. For example, a production compiler may wish to define multiple optimization levels that run different sets of passes.

4 Evaluation - a Nanopass Go Compiler

To evaluate the design of nanopass attribute grammars we have developed a prototype nanopass attribute grammar system and used it to implement several passes in a compiler for Go version 1.17 that generates x86_64 assembly language code. Go is a lexically-scoped, statically-typed, imperative language and, in many respects, has control-flow statements and expressions that one might expect. We note specific points of interest below as they become relevant.

In this section we describe several of the 32 languages and transformations between them. Section 4.1 describes the simple transformation that lowers for-loops into while-loops. Section 4.2 describes the more complicated transformation (and some of its predecessors) of lowering complex numbers into records with a real and imaginary field. This demonstrates how transformations lower not only the program syntax but also that of annotations. Section 4.3 describes the last language in the sequence; one that has been sufficiently lowered to enable a direct translation to assembly language.

[2] Although this rule is non-algorithmic, it can easily be conservatively approximated and is so implemented in our prototype system.

```
1  // for <clause> <body>
2  prod forStmt(clause: LoopClause,
3               body: Stmt): Stmt;
4
5  // for x != 0 { ... }
6  prod whileClause(expr: Expr): LoopClause;
7
8  // for i := 0; i < 10; i++ { ... }
9  prod forClause(init: Stmt, cond: Expr,
10                 post: Stmt): LoopClause;
11
12 // for k, v = range m { ... }
13 prod rangeClause(lhs: ExprList,
14                   rhs: Expr): LoopClause;
```

Figure 3. A subset of the productions describing loops.

Section A in the appendix lists and briefly describes all 32 languages and 34 passes (transformations).

4.1 Lowering for-style loops to while-style loops

Go supports both `for` loops that have the C-like structure of `for init; cond; post { body }` and while-style ones that have only a condition. We lower the former to the latter. Figure 3 shows, in the language of our prototype, that in the abstract syntax, both loops represented by a single generic `forStmt` production (line 2) that encapsulates all the looping constructs Go supports. Interestingly, they all use the keyword `for`, so this structure is not unreasonable. There are then various clauses that can be used with the `for` keyword, which are productions of the `LoopClause` nonterminal: while-loop style on line 6, C for-loop style on line 9, and a clause for ranging over key-value pairs on line 13. There are many other productions in $\mathcal{E}$ that we do not show, but they do have the expected form.

The meta-language syntax of the prototype essentially adds concrete syntax to the constructs in the formalism in Section 3. It does interleave elements of $\mathcal{E}$ and $\mathcal{L}$ but it should be straightforward to read. Productions are written in a functional style so that the left-hand-side nonterminal appears last and the right-hand-side elements are in parens.

The lowering of C-style for-loops into while-style loops takes place after 18 previous passes, on language `L15`, and is shown in Figure 4. `L0`, the initial language in our nanopass compiler, contains the abstract syntax of Go and the intervening languages resolve names and perform type checking. The transform attribute `toL16` (line 6) produces programs without these kinds of loops in language `L16`. Language `L16` is specified by extending `L15` as shown on lines 1–5. It removes (`-=`) the `forClause` production from nonterminal `LoopClause`'s set of productions (line 2) and includes only (`:=`) occurrences of attributes `liftedInit` and `liftedPost` on `LoopClause` (line 3). These are defined as higher-order

```
1  lang L16 extends L15 {
2    LoopClause.prods -= {forClause},
3    LoopClause.attrs := {liftedInit,
4                         liftedPost}
5  }
6  transform attribute toL16 from L15 to L16;
7
8  syn liftedInit: Stmt;
9  syn liftedPost: Stmt;
10
11 aspect forClause {
12   this.liftedInit := init;
13   this.toL16 := whileClause(cond);
14   this.liftedPost := post;
15 }
16
17 aspect default LoopClause {
18   this.liftedInit := emptyStmt();
19   this.liftedPost := emptyStmt();
20 }
21
22 aspect forStmt {
23   this.toL16 := block {appendStmt(
24     clause.liftedInit,
25     forStmt(
26       clause.toL16,
27       appendStmt(
28         body.toL16,
29         clause.liftedPost))));     }
```

Figure 4. Lowering C-style for-loops to while-style loops.

synthesized attributes holding statements (lines 8–9). The `aspect` constructs associate equations with productions and, by convention, use `this` as the name of the constructed tree node. Names of argument trees are found in the production declarations in Figure 3. These attributes lift the `init` and `post` components out of a C-style for-loop (line 12, line 14) and are the empty statement, by `default`, on other `LoopClause` productions (lines 17–20). The `forClause` can then transform itself into an `whileClause` that uses the `cond` child as the loop condition (line 13). Finally, we define an equation on the `forStmt` production to actually put the `liftedInit` and `liftedPost` statements into place in the new while-style loop (lines 22–29). For all other `Stmt` productions, *e.g.* if-then-else statements, a default equation is generated, as discussed in Section 3.4, to apply the `toL16` transformation to its components and build the same tree with those transformed results. Thus, a statement of the form `for init; cond; post { body }` is lowered to one of the form `{ init; for cond { body; post }}`. Figure 4 contains the entirety of the code for this pass.

4.2 Lowering complex numbers

Go supports complex numbers with imaginary number literals, overloaded arithmetic operators, and supporting library functions for constructing and accessing complex numbers. Here we discuss a transformation that lowers them to structs containing two floating-point numbers. (For simplicity we assume complex numbers are 64 bits; the transformation can easily be made to also handle 128-bit complex numbers.) Interestingly, complex number types and functions are not built-in syntax, but instead are library functions whose names may be shadowed by programmer-declared names, thus complicating the lowering.

This process is done in a few steps, which we will illustrate on an example function shown in its original `L0` form at the top of Figure 5. These versions are the concrete syntax versions of the results of the various transformations, lightly formatted. The ability to easily inspect the results at each step is an advantage of the nanopass approach.

By language `L8`, (see second version in Figure 5) earlier passes have performed name resolution and renamed lexical variables, so the names of all types and functions are fully qualified (shown in here with ad hoc syntax not in `L0`; the `""` package is the "universe block" in Go, and contains all predeclared identifiers). Lexically-bound names are made unique, using the `$` notation to attach unique numbers.

The first step in lowering complex numbers occurs in `L9` where we lower imaginary number literals to calls to the `complex` function (line 2 of `L9` in Figure 5). In `L11` we have recognized calls to polymorphic standard library functions, including the `complex`, `real` (access of the real component), and `imag` (accessing the imaginary component) functions, and given them their own productions, of the same names, in the language's abstract syntax. This is indicated here by bolding the constructs' names on lines 2–3. In Go 1.17, users cannot define polymorphic functions, and polymorphic functions behave unlike other variables, so it makes type-checking simpler to recognize them as productions.

In `L12` type-checking is done. The specification of `L11` adds a synthesized attribute occurrence for types, `ty: Type`, to `Expr` and the corresponding equations are then also included. Equations for type-checking complex number constructs, such as the recently added `complex` production, are also included so that these expressions are appropriately typed.

Figure 6 defines `L12` and converts the `L11` attribute `ty` into an annotation (line 4) so that typing information is retained (in `toL12`) for use in the remaining passes. As of `L14`, the `Expr` nonterminal contains the productions `complex`, `real`, `imag`, as well as the arithmetic operators that are defined on complex numbers. After `L15`, a type annotation with its `isComplex64` attribute set to true indicates an operation over complex values.

We transform away language-level support for complex numbers when transforming from `L14` to `L15` in `toL15`, also

Language `L0`:

```
1  func f(x complex64) float32 {
2      y := x + 1.2i
3      return real(y) - imag(y)
4  }
```

Language `L8`:

```
1  func f(x$0 "".complex64) "".float32 {
2      y$1 := x$0 + 1.2i
3      return "".real(y$1) - "".imag(y$1)
4  }
```

Language `L9`:

```
1  func f(x$0 "".complex64) "".float32 {
2      y$1 := x$0 + "".complex(0.0, 1.2)
3      return "".real(y$1) - "".imag(y$1)
4  }
```

Language `L11`:

```
1  func f(x$0 "".complex64) "".float32 {
2      y$1 := x$0 + complex(0.0, 1.2)
3      return real(y$1) - imag(y$1)
4  }
```

Language `L15`:

```
1  func f(x$0 struct{ r "".float32,
2                     i "".float32 })
3        "".float32 {
4      var y$1 struct{ r "".float32,
5                      i "".float32 }
6      y$1 = "$builtins".AddComplex64(
7        x$0,
8        struct{ r "".float32,
9                 i "".float32 }{ 0.0, 1.2 })
10     return y$1.r - y$1.i
11 }
```

Figure 5. An example function, lowered from L_0 to L_{15}.

shown in Figure 6. To get started, `L14` inherits the string attribute `complexOpName` from `L12` (line 1), to occur on binary operators (line 6) to be the name of the built-in function corresponding to the operator (line 9) to which the operator will translate. (This is the empty string for operators that don't apply to complex numbers.) Though not shown, it also inherits an attribute `isComplex64` (line 2 on type nonterminals (line 5) to indicate if a `Type` is a complex type.

The target language, `L15` on line 12, eliminates the complex number productions (line 13) and the attributes used in `L14` (lines 14–15) as part of `toL15` (line 16).

One part of toL15 translates binary operators (lines 18–20) to a call to a runtime function (line 22) whose name is based on the complex operator name (line 25) if the type of the operator is complex (line 21). The operands are also lowered (lines 27–28). The type annotation ty is set to be the lowered version of the complex type (line 30). This transformation of complex types to struct types is not shown; it produces a struct with a real and imaginary field of type float.

If the type is not complex then the transformation is applied to the child expressions and annotations (lines 32–34).

Lastly, the complex productions complex (line 38), real (line 48) , and imag (not shown), are also lowered to, respectively, struct literals (lines 39–45, or the struct selector operator (lines 49–50).

Thus, we see, that over a handful of passes, complex numbers are lowered into structure types and expressions.

4.3 L31: Nearly assembly language

After all passes, the final target language L31 is simple enough to generate assembly straightforwardly in a single pass, although register allocation has not been done. The declaration nonterminal has productions for functions and methods with stack frame layouts, opaque type definitions, and global variables without initializers. The statement nonterminal has productions for sequencing statements, for conditional and unconditional gotos, assignments (to variables or struct fields), and expressions evaluated for effect. The expression nonterminal has no productions for nested expressions, only those with atomic operands: literals, references to variables (including functions), and references to methods. The remaining productions are for those atomic operands, function calls, closure creation, casting between types, calls to the memory allocator, and references to struct fields.

5 Related Work

Naturally there are many extensions to Knuth's original attribute grammars that are related to this work. Transform attributes in Section 3 are essentially a version of higher-order attributes [25] that are constrained to have the same nonterminal type as the nonterminal on which they occur. This restriction allows for the automatic generation of equations for productions that do not define them explicitly. Reference [5] and remote [1] attributes allow trees already decorated with attribute values (or references to them) to be passed as attributes. A common use case is to link uses of a variable back to the tree that declared it. In some sense, trees with attributes converted to annotations by a transform attribute are similar in that they come with values already decorating them. Trees generated in transform attributes are more restricted, however; we only use them to construct the tree of the program output from the transformation.

```
1  syn complexOpName: string;
2  syn isComplex64: bool;
3  lang L12 extends L11 {
4    Expr.annots += { ty },
5    Type.attrs  += { isComplex64 },
6    BinOp.attrs += { complexOpName }, ... }
7
8  aspect add {      // prod add(): BinOp;
9    this.complexOpName := "ComplexAdd";
10 } // similarly for all BinOp productions
11
12 lang L15 extends L14 {
13   Expr.prods  -= { complex, real, imag },
14   Type.attrs  -= { isComplex64 },
15   BinOp.attrs -= { complexOpName } }
16 transform attribute toL15 from L14 to L15;
17
18 // prod binOpExpr(lhs: Expr, op: BinOp,
19 //                rhs: Expr): Expr;
20 aspect binOpExpr { this.toL15 =
21     if this.ty.isComplex64 then
22       callExpr(
23         varExpr(
24           qname("$runtime",
25                 op.complexOpName),
26           ty=c64BinopType),
27         exprsCons(lhs.toL15,
28         exprsCons(rhs.toL15,
29         exprsNil())),
30         ty=this.ty.toL15)
31     else
32       binOpExpr(lhs.toL15, op.toL15,
33                 rhs.toL15,
34                 ty=this.ty.toL15);
35 }
36
37 // prod complex(r: Expr, i: Expr): Expr;
38 aspect complex { this.toL15 =
39     compositeExpr(c64Type,
40       elementsCons(fieldKey("r"),
41                    r.toL15,
42       elementsCons(fieldKey("i"),
43                    i.toL15,
44       elementsNil())),
45       ty=c64Type);   }
46
47 // prod real(c: Expr): Expr;
48 aspect real { this.toL15 =
49     selectorExpr(c.toL15, "r",
50                  ty=this.ty.toL15);  }
```

Figure 6. Lowering complex numbers in struct types and expressions.

One mechanism for transforming trees is the tree-rewriting mechanism [20] in JastAdd [2] that rewrites trees to eliminate certain syntactic forms. Another is forwarding [24] in Silver [23] used for similar purposes. These differ from transform attributes in that both of these processes are more local in nature and not used for transforming an entire program. Silver also has strategy attributes [11], in which strategies control the application of rewrite rules to transform trees in a more global manner, but the differentiation of different languages is not possible there. Perhaps most similar to transform attributes are attribute coupled grammars [4]. These were a precursor to higher order attributes and were used to link attribute grammars in a sequence to translate a tree through a series of different languages. This is essentially what transform attribute do, but again the differentiation of many languages from a common collection of language elements is not present in this formalism.

The LISA system previously explored [14] splitting an attribute grammar specification into separate languages, which are defined using an object-oriented-inspired inheritance mechanism similar to our framework's extension mechanism described in Section 3.3. In fact our prototype uses the same `extends` keyword as LISA. An important difference is that LISA could not identify and use two distinct languages in the same phase of evaluation. This is needed in computations carried out during our language transformations to isolate the safe construction of some trees in the source language and others in the target language of the transformation.

The other primary body of related work is that of nanopass compilers. The original design and implementation of nanopass compilers was done for the Scheme language for both educational [18] and industrial applications [9]. This work articulated the software engineering benefits of the approach, such as greater transparency into the working of the compiler and more direct means for testing. This is certainly useful in educational settings and Jeremy Siek's new textbook adopts this approach [19]. In that work, programs are represented as Scheme data structures that are essentially syntax trees. These lack the ability to structure computations over the tree as flexibly as attribute grammars allow. This concern was noted in the GitHub repository for the Scheme-to-C compiler.[3]

6 Discussion and Conclusion

A concern one might have about nanopass attribute grammars used production-grade compilers is runtime performance. Are the many small-scale passes much slower than a few larger-scale ones? We have not performed a rigorous evaluation of the overall performance; however, Keep and Dybvig note [9] that after rewriting the Chez Scheme compiler to use the nanopass framework, compile times remained within a factor of two despite improvements to code generation, including a slower register allocator.

Another possible concern is the traditional, and somewhat conservative, well-definedness analysis used in our formulation in Section 3. It requires equations for all synthesized attributes and inherited attributes that occur on nonterminals in a production's signature. This is overly conservative, as some equations may be written that are never actually demanded. More sophisticated analyses have been developed, such as one by Kaminski and Van Wyk [6] that checks for *effective completeness*; that is, all potentially demanded attributes have an equation. Performing this analysis and a traditional higher-order circularity analysis [25] for each language in $\mathcal{L}$ would be straightforward.

There are a number of aspects of future work that we are currently investigating. The first is the further development of the prototype nanopass AG system to make it more robust and extend it with more modern attribute grammar features such as some of those described in Sections 2 and 5. This will allow us to do a more complete evaluation of nanopass attributes grammars by applying it to more language. The current prototype is contains the features needed to experiment with the nanopass formalism described in Section 3 but it lacks many of the modern AG features that improve the usability and convenience of the paradigm.

Although, as described above, we have reason to believe that performance is not a significant problem there is one optimization that is appealing — fusing several independent passes into one to avoid an additional traversal over the tree.

Another potential extension is to specify a target language for all attributes, rather than just transform attributes. This would enable a notion of reference attributes using annotations, by defining the language of, *e.g.*, an environment attribute mapping names to definitions to have some attributes on definitions present as annotations. This may complicate language checking, as the source language of an attribute could no longer be determined from its target language.

To conclude, we have introduced nanopass attribute grammars, a formalization of their specification, and a prototype system used to define many aspects of a compiler for the Go programming language. The distinguishing feature of nanopass attribute grammars is the clear identification of many distinct, yet similar, languages drawn from the same set of language elements. This provides the linchpin on which the static *language checking* depends so that the attribute grammars for individual languages can be shown to be well-defined even when the entire collection of language elements in $\mathcal{E}$ will most likely not be. Perhaps equally important is the clarity of thought that this style of compiler design brings: one can think in terms of clearly defined languages, knowing what has, and has not, been translated away or had its structure change in some way. In a large complex software artifact such as a compiler this is a considerable benefit.

[3] https://github.com/akeep/scheme-to-c/blob/18f6cd26f/c.ss#L2576-L2578

A The passes in the Go nanopass compiler

The Go compiler we have designed consists of 32 different languages, `L0` to `L31` and 34 passes. Three passes, `renameVariables`, `makeDerefExplicit`, and `lowerSelectorMethodCalls`, have the same source and target language.

The last language has an attribute that outputs x86_64 assembly. Since there is no register allocator, this assembly is inefficient, but it is runnable. We have also not implemented the runtime support needed for concurrency.

1. `giveImportsNames`, `L0` to `L1` – Rewrites imports like `import "example.com/foo"` to `import bar "example.com/foo"`. The latter form already exists in the CST, and this removes a special case in the next pass.
2. `fullyQualifyNames`, `L1` to `L2` – Rewrites variables that refer to imported declarations to refer to the package directly. For example, `Println` might become `"fmt".Println`, `foo.Bar` might become `"example.com/foo".Bar`, and `int32` might become `"".int32`.
3. `renameVariables`, `L2` to `L2` – Renames lexical variables to have globally unique names, so that variable shadowing doesn't become an issue. For example, `x` might become `x$45`.
4. `liftTypesAndConstants`, `L2` to `L3` – Lifts declarations of types and constants in local scopes to the global scope, and renames references to them to fit. (Due to `renameVariables`, we know there will not be any name conflicts.)
5. `expandTypeAliases`, `L3` to `L4` – Expands and removes type aliases. Note that this only applies to declarations like `type a = b`, not `type a b`.
6. `expandLists`, `L4` to `L5` – Rewrites function parameters, struct fields, etc. like `func foo(a, b "".int)` to `func foo(a "".int, b "".int)`.
7. `lowerIncDec`, `L5` to `L6` – Lowers increment and decrement statements to the corresponding assignment statements.
8. `labelLoops`, `L6` to `L7` – Adds labels to loops that lack them, and makes `break` and `continue` statements explicitly refer to their loop.
9. `normalizeInterfaces`, `L7` to `L8` – Sorts the methods of interface types to be in lexicographic order, and resolves any interface inclusions. This is useful for type-checking later.
10. `lowerImaginaryLits`, `L8` to `L9` – Lowers imaginary literals to calls to the `complex` function with a zero real part.
11. `recognizeMakeAndNew`, `L9` to `L10` – Recognizes the `make` and `new` constructs, and provides errors for uses of types as function call arguments other than those constructs.
12. `recognizePolyBuiltins`, `L10` to `L11` – Recognizes the other polymorphic built-in functions.
13. `typeCheck`, `L11` to `L12` – Adds a type annotation to expressions.
14. `makeDerefExplicit`, `L12` to `L12` – Adds uses of the dereference operator that were implicit in the source language. For example, `foo$7.Bar()` might be rewritten to `(*foo$7).Bar()`.
15. `lowerSelectorMethodCalls`, `L12` to `L12` – Lowers method calls that can be statically dispatched to direct calls, and references to those methods to lambdas. For example, `x$3.Foo(n$2)` might be rewritten to `("foo".MyStruct).Foo(x$3, n$2)`, and `y$4.Foo` might be rewritten to `func(n$1321 "".int) { ("foo".MyStruct).Foo(y$4, n$1321) }`.
16. `hoistVariableDecls`, `L12` to `L13` – Lifts variable declarations to the start of their nearest enclosing function.
17. `removeIfPreStmt`, `L13` to `L14` – Removes the "pre statement" from `if` statements. For example, rewrites `if n, err = "foo".bar(); n != nil {}` to `n, err = "foo".bar(); if n != nil {}`.
18. `lowerComplex` (to`L15`), `L14` to `L15` – Lowers calls to the complex-number-related built-in functions, and operators on complex numbers, to calls to runtime functions.
19. `lowerForLoops` (to`L16`), `L15` to `L16` – Lowers for-style loops to while-style loops.
20. `lowerForRangeLoops`, `L16` to `L17` – Lowers loops using the `range` construct to use indices on arrays, slices, and strings, and runtime functions on channels and maps.
21. `lowerIfThen`, `L17` to `L18` – Lowers if-thens without an else to if-then-elses.
22. `lowerSelect`, `L18` to `L19` – Lowers `select` statements to calls to a runtime function.
23. `lowerTypeSwitch`, `L19` to `L20` – Lowers `switch` statements on the runtime type of a value to a series of `if`s.
24. `lowerExprSwitch`, `L20` to `L21` – Lowers other `switch` statements to a series of `if`s, including their `fallthrough` statements.
25. `lowerControlFlow`, `L21` to `L22` – Lowers the remaining control-flow constructs (`break`, `continue`, `for`, `if`, `return`) to `goto`s and labels.
26. `lowerDefer`, `L22` to `L23` – Lowers the `defer` statement to a shadow stack and adds code to the exit blocks of functions to support `defer`.
27. `liftInitFunction`, `L23` to `L24` – Recognizes the definition of `init` functions and lifts their bodies to the `Package` nonterminal.
28. `inlineConstants`, `L24` to `L25` – Inlines references to `const`s and removes the `const`s' declarations.
29. `liftInitializers`, `L25` to `L26` – Lifts initializers for global variables and constants to the `init` function.

30. `flattenExprs`, `L26` to `L27` – Lowers complex expressions to simple ones; *e.g.* `"foo".f("foo".g())` might get lowered to `tmp$1322 = "foo".g(); "foo".f(tmp$1322)`.
31. `lowerCompositeExprs`, `L27` to `L28` – Lowers assignments of composite expressions to a series of assignments. For example, `x$22 = "foo".MyStruct{1,2}` might get lowered to `x$22.Foo = 1; x$22.Bar = 2`, and `xs$23 = []"".int{1,2,3}` might get lowered to `xs$23 = make([]"".int, 3); xs$23[0] = 1; ...`, and so on.
32. `lowerPolyBuiltins`, `L28` to `L29` – Transforms calls to the polymorphic built-in functions recognized by `recognizePolyBuiltins` to calls to runtime functions.
33. `layoutStackFrames`, `L29` to `L30` – Places the local variables declared in each function and method into a single struct, such that each function has exactly one local variable. A static link is also present, containing a pointer to the parent's stack frame struct, for lambdas' stack frames.
34. `lambdaLift`, `L30` to `L31` – Lifts lambda expressions to global functions that take an additional argument for their parent's stack frame and a call to a built-in function that accepts the global function pointer and the parent's stack frame pointer and constructs the closure. Globally-defined functions and methods also get a globally-defined "closure" (that ignores the parent's stack frame). References to globally defined functions and methods are changed to refer to these closure objects instead.

References

[1] John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687. https://doi.org/10.1145/1082036.1082042

[2] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)* (Montreal, Quebec, Canada). ACM, 1–18. https://doi.org/10.1145/1297027.1297029

[3] R. Farrow. 1986. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN Notices* 21, 7 (1986). https://doi.org/10.1145/13310.13320

[4] H. Ganzinger and R. Giegerich. 1984. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. 157–170. https://doi.org/10.1145/502874.502890

[5] Görel Hedin. 2000. Reference Attribute Grammars. *Informatica* 24, 3 (2000), 301–317.

[6] Ted Kaminski and Eric Van Wyk. 2012. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE) (Lecture Notes in Computer Science, Vol. 7745)*. Springer, 352–371. https://doi.org/10.1007/978-3-642-36089-3_20

[7] Uwe Kastens. 1980. Ordered attributed grammars. *Acta Informatica* 13 (1980), 229–256. Issue 3. https://doi.org/10.1007/BF00288644

[8] U. Kastens and W. M. Waite. 1994. Modularity and reusability in attribute grammars. *Acta Informatica* 31 (1994), 601–627. https://doi.org/10.1007/BF01177548

[9] Andrew W. Keep and R. Kent Dybvig. 2013. A Nanopass Framework for Commercial Compiler Development. *SIGPLAN Not.* 48, 9 (sep 2013), 343–350. https://doi.org/10.1145/2544174.2500618

[10] Donald E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511 Corrections in **5**(1971) pp. 95–96.

[11] Lucas Kramer and Eric Van Wyk. 2020. Strategic Tree Rewriting in Attribute Grammars. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Virtual, USA). 210–229. https://doi.org/10.1145/3426425.3426943

[12] José Nuno Macedo, Marcos Viera, and João Saraiva. 2022. Zipping Strategies and Attribute Grammars. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 13215)*. Springer, 112–132. https://doi.org/10.1007/978-3-030-99461-7_7

[13] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. https://doi.org/10.1016/j.scico.2005.06.005 Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA '03).

[14] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. 1998. The template and multiple inheritance approach into attribute grammars. In *Proceedings of the 1998 International Conference on Computer Languages*. 102–110. https://doi.org/10.1109/ICCL.1998.674161

[15] Jukka Paakki. 1995. Attribute Grammar Paradigms—a High-Level Methodology in Language Implementation. *Comput. Surveys* 27, 2 (June 1995), 196–255. https://doi.org/10.1145/210376.197409

[16] Michael Rodeh and Mooly Sagiv. 1999. Finding Circular Attributes in Attribute Grammars. *J. ACM* 46, 4 (July 1999), 556–ff. https://doi.org/10.1145/320211.320243

[17] Joao Saraiva. 1999. *Purely Functional Implementations of Attribute Grammars*. Ph. D. Dissertation. University of Utrecht.

[18] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A Nanopass Infrastructure for Compiler Education. *SIGPLAN Not.* 39, 9 (sep 2004), 201–212. https://doi.org/10.1145/1016848.1016878

[19] Jeremy G. Siek. 2023. *Essentials of Compilation: An Incremental Approach in Racket*. The MIT Press.

[20] Emma Söderberg and Görel Hedin. 2015. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures* 44 (2015), 3 – 23. https://doi.org/10.1016/j.cl.2015.08.008 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[21] S. D. Swierstra and P. R. Azero. 1998. *Attribute grammars in the functional style*. Springer US, Boston, MA, 180–193. https://doi.org/10.1007/978-0-387-35350-0_14

[22] L. Thomas van Binsbergen, Jeroen Bransen, and Atze Dijkstra. 2015. Linearly Ordered Attribute Grammars: With Automatic Augmenting Dependency Selection. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15)*. ACM, 49–60. https://doi.org/10.1145/2678015.2682543

[23] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

[24] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th Conference on Compiler Construction (CC) (Lecture Notes in Computer Science, Vol. 2304)*. Springer-Verlag, 128–142. https://doi.org/10.1007/3-540-45937-5_11

[25] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 131–145. https://doi.org/10.1145/73141.74830

Received 2023-07-07; accepted 2023-09-01