Ensuring Non-interference of Composable Language Extensions

Ted Kaminski
Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
tedinski@cs.umn.edu

Eric Van Wyk
Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
evw@umn.edu

Abstract

Extensible language frameworks aim to allow independently-developed language extensions to be easily added to a host programming language. It should not require being a compiler expert, and the resulting compiler should "just work" as expected. Previous work has shown how specifications for parsing (based on context free grammars) and for semantic analysis (based on attribute grammars) can be automatically and reliably composed, ensuring that the resulting compiler does not terminate abnormally.

However, this work does not ensure that a property proven to hold for a language (or extended language) still holds when another extension is added, a problem we call *interference*. We present a solution to this problem using of a logical notion of *coherence*. We show that a useful class of language extensions, implemented as attribute grammars, preserve all coherent properties. If we also restrict extensions to only making use of coherent properties in establishing their correctness, then the correctness properties of each extension will hold when composed with other extensions. As a result, there can be no interference: each extension behaves as specified.

CCS Concepts • **Software and its engineering → Extensible languages**;

Keywords language extension composition, attribute grammars

ACM Reference Format:
Ted Kaminski and Eric Van Wyk. 2017. Ensuring Non-interference of Composable Language Extensions. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136023

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136023

1 Introduction

There exist many systems that allow language extensions to be developed, composed together with a host language, and used together to write programs. These include tools like JastAdd [4] and SugarJ [6]. However, most of these tools do not ensure composition will automatically be successful, and may require a compiler engineer to intervene. For example, newly introduced syntax from two extensions may result in ambiguities or errors in parser generation.

Our working hypothesis is that *reliable composition* is required for language extension to achieve practicality. It should not be possible for two extensions to conflict, and so the ordinary programmer, simply attempting to make use of them, will not need to ever debug their compiler. Copper [26] is a parser generator with a modular determinism analysis [23] that ensures extension syntax will compose without conflict (i.e. yield a deterministic LALR(1) parser). Silver [11, 24] is an attribute grammar-based [14] language with forwarding [25] that comes with a modular well-definedness analysis [12] that ensures extension semantics (in the form of an attribute grammar) will compose without conflict (i.e. the composed attribute grammar has no missing equations). Together, these analyses ensure composition will be successful and the compiler will not terminate abnormally, but they do not guarantee that it behaves as expected.

Each of these analyses restricts the range of possible extensions, however. These restrictions cannot be too onerous: we need to be able to produce useful extensions, and we need the user experience to be like that of native language features. In this paper, we wish to solve this final source of conflict, and ensure the composed compiler behaves as expected, and do so without seriously compromising the capability of language extensions.

1.1 The Problem of Interference

The problem of interference between extensions arises because the developers of independent extensions (E_A and E_B) are unable to examine the composition of these two extensions with a host language H. In the notation of Erdweg et al. [5] this is $H \triangleleft (E_A \uplus_\emptyset E_B)$, where $\uplus_\emptyset$ denotes automatic extension composition and $\triangleleft$ denotes extending a host language. Each artifact for H, $H \triangleleft E_A$, and $H \triangleleft E_B$ are language specifications about which their developers can reason or

```
nonterminal Expr;
syn eval :: Bool occurs on Expr;

production or
e::Expr ::= l::Expr  r::Expr
{
  e.eval = l.eval || r.eval;
}
production literal
e::Expr ::= b::Bool
{
  e.eval = b;
}

production ors
e::Expr ::= es::[Expr]
{
  forwards to
    foldr(or, literal(false), es);
}
```

```
syn tainted :: Bool occurs on Expr;

production taint
e::Expr ::= x::Expr
{
  e.tainted = true;
  forwards to x;
}
aspect or
e::Expr ::= l::Expr  r::Expr
{
  e.tainted =
    l.tainted || r.tainted;
}
aspect literal
e::Expr ::= b::Bool
{
  e.tainted = false;
}
```

```
syn transform :: Expr occurs on Expr;

production id
e::Expr ::= x::Expr
{
  forwards to x.transform;
}
aspect or
e::Expr ::= l::Expr  r::Expr
{
  e.transform =
    or(l.transform, r.transform);
}
aspect literal
e::Expr ::= b::Bool
{
  e.transform = literal(b);
}
```

Figure 1. A simple example of interference. Left: a small host language H. Middle: E_A. Right: E_B.

write tests. It may not be possible to construct any flawed trees in these languages, but we may be able to do so for the composed language. Indeed, it may be difficult to precisely identify what "flaw" means without a semantics for the composed language specifying how the two extensions should interact. Worse still, having found a tree that reveals an apparent flaw in the composed compiler, there may not be any obvious way to fix it. Both extensions may seem correct in isolation, and the flaw may be the result of an unfortunate interaction that seems the fault of neither or seems to require "glue code" to fix. When extensions are developed independently, the developers may simply point their fingers at each other, resolving nothing.

Consider Figure 1, showing the abbreviated Silver attribute grammar specification for two extensions to a very primitive Boolean expression language. The host consists of `or` and `literal` constructs, as well as the "syntactic sugar" `ors` that expands to an equivalent tree by folding up the list of expressions `es` using `or` and `literal`. Equations defining attributes can be written for forwarding productions, but they are not required because forwarding automatically introduces "copy equations" that use the value from the tree they "forward to" instead. Thus, the synthesized attribute `eval` only needs equations for `or` and `literal` productions as the value of `eval` on `ors` is copied from folded-up forwards-to tree. When language extensions introduce new productions on host nonterminals, they must having forwarding equations, as we see in the figure. Extension E_A describes a simple sort of "tainting" analysis, introducing both a new attribute (`tainted`) and a new production (`taint`). The attribute gets its meaning by defining equations for each production using `aspects`. Extension E_B introduces a simple identity tree transformation, including the attribute that computes it (`transform`) and a production that invokes it (`id`). Although this latter extension might seem useless, it is actually the simplest form of tree transformation, which are generally quite useful.

Because `transform` is only defined on host language productions (and is unaware of other extensions like E_A and so cannot handle `taint` except via forwarding) this attribute has the effect of replacing forwarding productions with what they forward to. In the figure, we see that `taint` production of E_A simply forwards to whatever expression it wraps. Thus, the analysis's success depends on their `tainted` attribute being evaluated on the forwarding production. However, this means there is trouble for a composed language tree:

```
id(or(literal(false), taint(t))).tainted
```

Here, `id` (which should do nothing) will essentially transform away the reference to `taint`, leaving the `tainted` analysis nothing to discover, even though it should be discovering a tainted subexpression in this case.

Each extension works as expected in isolation: no tree exhibits any misbehavior. The trouble here is not *just* that there is a problem with the composed language that is undetected in isolation. We also have no guidance on how this problem should be resolved. Did E_A err in forwarding to its wrapped expression for the taint annotation syntax? Did E_B err in transforming away the other extension's syntax? Both? Something else? For extensible languages to be useful for mainstream software development this problem needs a solution. After all, the space of potential interference grows exponentially with each new extension in the ecosystem. Interference can easily start off seeming like a non-issue, and then suddenly seem like an insurmountable problem.

Outline As contributions, we show:

- How a class of properties proven for an attribute grammar are automatically preserved when composed with other extensions, using a logical tool we call *coherence*.
- That coherence assigns blame for interference.
- How to generalize on coherence, easing restrictions enough for extensions to feel native.

- That non-interference can be policed using ordinary testing, without verification, and we apply this technique to a real compiler as evaluation.

In Section 2, we describe some background on how to reason about attribute grammars. In Section 3, we define coherence, in Section 4, we define a non-interfering extension in terms of coherence, and then in Section 5, we see how to show that extensions are non-interfering. In Section 6, we show how we can relax our approach to showing extension is non-interfering, with the goal of allowing them to feel like native language features. In Section 7, we show how interference violations can be tested for, and in Section 8 we apply this to a real compiler, before concluding.

2 Reasoning About Language Extensions

We will tackle this problem by considering the verification of language extensions. Our model has each artifact (the host language and each extension) come with an associated set of correctness properties. For example, with the `tainted` extension, we might wish to show a simple property:

$$\forall t.\ \texttt{taint}(t).\texttt{tainted} = \texttt{true}$$

More generally, the host language H will have a set of properties $\mathscr{P}(H) = \{P_1^H, P_2^H, \ldots\}$, and proofs that each of these properties holds, i.e. proofs of $\forall t \in H.\ P_i^H(t)$ for each i. Similarly, an extension E_A will have a set of properties $\mathscr{P}(E_A) = \{P_1^A, P_2^A, \ldots\}$, and proofs of the form $\forall t \in H \triangleleft E_A.\ P_i^A(t)$. Note that now quantification ranges over the extended language. An extension will also contain all properties the host language contains, so $\mathscr{P}(E_A) \supseteq \mathscr{P}(H)$. This means the extension must provide proofs of $\forall t \in H \triangleleft E_A.\ P_i^H(t)$ (i.e. proofs of host properties but quantified over extended language trees.)

Our approach to eliminating interference is for all of these properties to remain true of a composed language. That is, given n language extensions, we want:

$$\forall i \in [1 \ldots n], P \in \mathscr{P}(E_i), t \in H \triangleleft \{E_1 \uplus_\emptyset \cdots \uplus_\emptyset E_n\}.\ P(t)$$

When the correctness properties for each extension are proven to hold for the composed compiler, we have eliminated the possibility of interference sneaking into the composed compiler. Each extension behaves as specified.

This must be accomplished *modularly*: we need to enforce something on each individual extension in isolation, which ultimately achieves this goal. We call this modular restriction *noninterfering*, and we want to show:

Theorem 2.1 (Modular non-interference)**.**

$$\begin{aligned}&\left(\forall i \in [1,n].\ \mathit{noninterfering}(H \triangleleft E_i)\right) \implies \\ &\quad \forall i \in [1,n], P_j^i \in \mathscr{P}(E_i).\ (\forall t \in H \triangleleft E_i.\ P_j^i(t)) \implies \\ &\qquad \forall t \in H \triangleleft \{E_1 \uplus_\emptyset \cdots \uplus_\emptyset E_n\}.\ P_j^i(t)\end{aligned}$$

We can note right away that this goal gives us a precise notion of blame, despite not yet having a definition for *noninterfering*. If an extension is non-interfering, then correctness proofs for different extensions can be automatically extended over it. If a bug exists in the final, composed artifact, then either one extension is interfering (that is, *noninterfering* does not hold of it), or one extension's specification was incorrect. Both of these possibilities can be observed in isolation from other extensions. Thus, even if an end-user is not able to diagnose a problem, the extension developers involved will not be conflicted about who to blame for causing interference.

2.1 Reasoning About Attribute Grammars

Definition 2.2 (Properties)**.** A property over a tree is a logical proposition with a free (tree-typed) variable. For the purposes of this paper, we restrict the proposition to:

1. simple logical connectives (and/or/implies),
2. quantification over non-tree types (such as numbers),
3. the use of inductively-defined relations over trees (such as equality), and
4. the use of attribute evaluation on trees (which we call *evaluation relations*).

The results we later show of properties of this sort can be generalized (e.g. to permit quantification over multiple trees, see Kaminski [10, Chapter 6]), and we further conjecture they can be generalized to any propositions. Note that evaluation relations *are* inductive relations, but the former come from the equations written in the attribute grammar, and the later are part of a verification, and so we distinguish them. Examples of properties and relations appear in Figure 2.

Note that we talk about properties P that range over one language, and then conflate them to also apply to an extended language. For instance, the claim that $\mathscr{P}(E_A) \supseteq \mathscr{P}(H)$ seems like a type error, since the properties range over different languages ($H \triangleleft E_A$ and H alone respectively). We will fix this issue in Section 3, but point out this imprecision now.

Also note that we abuse notation slightly by using a *language* as the bound of quantification. This is a shorthand for indicating which productions may appear in the tree. Properties actually quantify over *decorated trees*: trees rooted in a particular nonterminal, that have been supplied with an initial set of inherited attributes, and thus attributes can be evaluated. When discussing properties from an abstract language H, lacking any particular nonterminals, we just indicate which language the nonterminal should be from.

We also assume that properties are proven by induction on those decorated trees. Critically, this is not the same as induction on the undecorated tree. A node in the decorated tree not only has a child decorated node for each child of the production, but also the forwarded-to tree. Consequently, induction on decorated trees means the induction hypothesis for a production allows us to assume the property holds of the forward tree, just like any other child. This is valid so long as expansion of forwarding equations terminates, which we will simply assume for our purposes.

```
nonterminal RepMin;
syn min :: Integer occurs on RepMin;

production inner e::RepMin ::= x::RepMin y::RepMin
{ e.min = min(x.min,y.min);  }

production leaf e::RepMin ::= x::Integer
{ e.min = x;  }
```

Relevant relations:
  lte_all : *RepMin* → *Integer* → *Prop*
  lte_all(*inner*(*x*, *y*), *m*) :- lte_all(*x*, *m*) ∧ lte_all(*y*, *m*)
  lte_all(*leaf*(*x*), *m*) :- *m* <= *x*
  exists : *RepMin* → *Integer* → *Prop*
  exists(*inner*(*x*, *y*), *m*) :- exists(*x*, *m*) ∨ exists(*y*, *m*)
  exists(*leaf*(*x*), *m*) :- *x* = *m*
Relevant property: min_correctness:
  ∀*t*, *m*. *t*.min = *m* → lte_all(*t*, *m*) ∧ exists(*t*, *m*)
Proof: By induction on the decorated tree *t*.

Figure 2. Example of inductive relations and properties to show of an attribute grammar-defined language.

Finally, we pay special attention to inductively-defined relations over trees. These are relations defined by case analysis on the production at the root of a tree. Examples of this appear in Figure 2, which is based on Bird [2]. We define new relations `lte_all` and `exists` which exact different requirements on `leaf` and `inner` nodes. (We use the reverse implies notation (`:-`) to better show how each production maps to its consequents.) The `min_correctness` property also makes use of the evaluation relation for `min`, which we write using the attribute grammar notation (*t*.`min` = *m*).

2.2 Extending Proofs to Extended Languages

If we have a fully verified host language, and we introduced a language extension, what is involved in modifying the verification to accommodate the extension? Note that we are not yet considering the much harder problem of composing extensions together, just the process involved in building a single extension. In Figure 3, we describe a simple extension to our RepMin example.

Our minimal host language only defines the `min` attribute. Here we add in the usual `rep` transformation (replace leaf values with the global minimum, preserving structure), as well as the new production `three`, which merely serves as an example of a new syntactic construct. For the verification, we have a new property describing `rep`. This extension has the following effects on the verification:

- The host language's relations need extending to new forwarding productions. In the upper right, we show how to extend `lte_all` and `exists` to handle `three`.
- Any proofs of properties from the host language will have new cases to show for new productions. The proof of `min_correctness` will require a new case for `three`, though we do not show this in the figure.
- Extensions may introduce new properties and relations defined inductively on nonterminals of the host language, along with new proofs showing new properties about trees. (Such as the definition of `eq_all` and proof of the associated property `rep_all_equal`.) These can be properties about the unextended language that the extension wishes to make use of, but the host language had not shown itself. (No such property is present in this example, but in principle the `min_correctness` property may not have been in the host language, and could be present here instead.)
- For new nonterminals introduced by the extension (not a part of this example), there may also be new properties and new proofs.

```
production three
l::RepMin ::= x::RepMin y::RepMin z::RepMin
{ l.min = min(z.min, min(x.min, y.min));
  forwards to inner(inner(x, y), z);  }

syn rep :: RepMin;
inh gmin :: Integer;

aspect inner  e::RepMin ::= x::RepMin y::RepMin
{ e.rep = inner(x.rep, y.rep);
  x.gmin = e.gmin;
  y.gmin = e.gmin;   }

aspect leaf  e::RepMin ::= x::Integer
{ e.rep = leaf(e.gmin);  }
```

Extensions to existing relations:
  lte_all(*three*(*x*, *y*, *z*), *m*) :-
    lte_all(*x*, *m*) ∧ lte_all(*y*, *m*) ∧ lte_all(*z*, *m*)
  exists(*three*(*x*, *y*, *z*), *v*) :-
    exists(*x*, *v*) ∨ exists(*y*, *v*) ∨ exists(*z*, *v*)
New relations:
  eq_all(*three*(*x*, *y*, *z*), *m*) :-
    eq_all(*x*, *m*) ∧ eq_all(*y*, *m*) ∧ eq_all(*z*, *m*)
  eq_all(*inner*(*x*, *y*), *m*) :- eq_all(*x*, *m*) ∧ eq_all(*y*, *m*)
  eq_all(*leaf*(*x*), *m*) :- *x* = *m*
New property: rep_all_equal:
  ∀*t*, *m*, *r*. *t*.global_min = *m* → *t*.rep = *r* → eq_all(*r*, *m*)
New Proof: By induction on the decorated tree *t*.

Figure 3. An example of an extension, showing extension productions, equations, attributes, relations, proofs.

In isolation, the extension developers are capable of manually extending the definitions and proofs and introducing the necessary new ones, as we have in this example. Extension is not the problem. The challenges arise from composition of independent extensions. As soon as we have two independent extensions, each developer no longer knows about the other's new productions, nor do they know what new properties other extensions might be relying on. There is

also what seems to be a creative component: specifications themselves have gaps, not just proofs. When a property is introduced independently from a new production, we do not have a complete specification. Somehow, we need to be able to complete these automatically.

3 Coherence

We solve the problem of completing both specifications and proofs by introducing the concept of *coherence*. This allows us to write modular proofs about extensions, and be confident they hold for a language composed with other extensions. To start with, coherence gives us a mechanism to automatically complete the gaps in specifications that arise as a result of composition of multiple language extensions.

Coherence can be thought of as requiring the "semantic equivalence of forwarding"—a phrase used even in the original paper on forwarding [25], but has been without a precise definition until now. Properties that are not equally true of a forwarding tree and what it forwards to are considered to be incoherent.

Definition 3.1 (Coherent properties)**.** A property P over a language L is coherent if

$$\forall t \in L.\ P(t) \iff P(t.forward)$$

Where $t.forward$ is defined as the tree that the root production of t forwards to (or if the root is non-forwarding, then $t.forward = t$.) In other words, coherence requires consistency between the properties that hold of a tree rooted in a forwarding production and the tree it forwards to.

Coherence gives us an automatic and natural means of dealing with the gaps in specifications that arise as a result of composition. These gaps are created because relations defined over nonterminals require new cases to handle any new forwarding productions from other extensions. Because any coherent property should be equally true of a tree and what it forwards to, this gives us an automatic method of determining what should hold of a new forwarding production for an existing relation. Unary relations are just one kind of property, and so a unary relation is coherent according to the above definition of a coherent property. So let us begin with unary relations:

Definition 3.2 (Coherent extension of unary relations)**.** For a coherent unary relation R over a language H, we can automatically extend this relation over an extended language $H \triangleleft E$. We leave the definition of $R(t)$ alone for all cases where t is a non-forwarding production, and for each forwarding production define it as being equivalent to $R(t.forward)$. Since we assume termination, this is well-defined because recursive expansion of $R(t.forward)$ will ultimately terminate in non-forwarding productions from the host language, a restricted subset of cases where we already have a complete definition of R. The resulting relation is still coherent, if all of its dependent relations are still coherent.

As an example[1], one of the missing cases for a relation that we manually filled in back in Figure 3 was for the `three` production, which forwarded to `inner`, and so we would have the following coherent extension:

$$\texttt{lte\_all}(three(x, y, z), m)\ \texttt{:-}$$
$$\texttt{lte\_all}(\texttt{inner}(\texttt{inner}(x, y), z), m)$$

This defers the meaning of `lte_all` on `three` to the tree that `three` forwards to, instead of the extension having to manually specify it.

Coherent extension of unary relations can be generalized to n-ary relations, and likewise, we can generalize the notion of coherent property. Ultimately, we can show:

Theorem 3.3 (Coherent relations imply coherent properties)**.** *If every relation R in a property P is coherent, and P contains no quantifiers, then $\forall \overline{x}.\ P(\overline{x})$ is a coherent proposition.*

These proofs appear in Kaminski [10], though it remains future work to show this holds for arbitrary propositions (i.e. those that are not limited to leading universal quantifiers, as in this theorem). As a consequence of this theorem, coherent extension of properties is possible simply by coherently extending each relation involved. Thus, we are able to freely take any coherent property P over a language H, and sensibly speak of that "same" property (really its coherent extension) holding over an extended language $H \triangleleft E$.

3.1 Examples of Incoherence

In light of this new notion of coherence, let us revisit the examples from Section 1.1. For `tainted` (from Figure 1) we have a problem with the `taint` production. Recall the simple example property we might wish to show, and let us choose an example instantiation:

t.`is_tainted` = `true` where t = `taint(literal(false))`

This property asserts the attribute should evaluate to true on t, but t forwards to just `literal(false)`, where the attribute evaluates to false. As a result, this property is incoherent.

Meanwhile, though we have not yet written a specification, we have no evidence of coherence problems with `id`. Indeed, intuitively, it is hard to imagine any: its production just forwards with no equations, so all attribute values are equal between the original and forwarded-to tree. So this begins to suggest an answer to our earlier questions about who is to blame for the interference problem.

3.2 Coherent Equality

The traditional notion of equality of two trees turns out to be incoherent. The trouble is that this notion refuses to see a forwarding production as "equal to" what it forwards to. We can introduce a new notion of tree equality to repair the problem. This essentially corresponds to writing down

[1] Already this example is not unary, but the second parameter is a primitive type, so it still fits with our limited notion of property.

an equality relation over non-forwarding productions, and relying on coherent extension to fill in the remaining cases:

Definition 3.4 (Coherent equality). $x =_N y$ (where $x, y : N$)

$$= \begin{cases} x_i =_M y_i & \text{if } x = p(\overline{x}),\ y = p(\overline{y}), \text{ and } x_i, y_i : M \\ x.forward =_N y & \text{if } x \text{ has forwarding root} \\ x =_N y.forward & \text{if } y \text{ has forwarding root} \end{cases}$$

That strict equality on trees is not coherent is actually an advantage we make use of later.

4 Non-interference Through Coherence

Coherence provides a way to complete gaps in specifications; this section describes how to complete proofs.

Definition 4.1 (Non-interference). A language extension is defined to be non-interfering (*noninterfering*$(H \triangleleft E)$) if:

1. The extension is coherent. That is, each property in its specification is a coherent property:

$$\forall P \in \mathscr{P}(E).\ \forall t \in H \triangleleft E.\ P(t) \iff P(t.forward)$$

2. The extension preserves all coherent properties. Or in other words, any true coherent property about the host is a true coherent property about the extended language (and vice versa):

$$\forall P.\ \big(\forall t \in H.\ P(t) \iff P(t.forward)\big) \implies \big(\forall t \in H.\ P(t)\big) \iff \big(\forall t \in H \triangleleft E.\ P(t)\big)$$

It is certainly possible to violate only one of these requirements. The developer of an interfering extension might simply leave out an incoherent property and this must be caught by requirement 2, *e.g.* if the `tainted` extension stated no properties about its behavior. Or the developer might state an "unreasonably strong" true incoherent property and thus must be caught by requirement 1, *e.g.* if the `id` extension tried to use incoherent equality. However, as we will see shortly, in practice these are almost always violated together.

4.1 Coherence Assures Modular Non-interference

With this definition, we can now see how to show Theorem 2.1, the definition for modular non-interference. Part 2 of the definition of *noninterfering* means we can extend proofs of coherent properties from the host to any extension. Let us work up from there.

First, we exploit an interesting property about composition of modularly well-defined attribute grammar modules. Only the set of modules included matters; the attribute grammar's behavior is never different for any "order" to how they get included. This is because the only possible side-effect of composing modules is the generation of copy equations in forwarding productions, and these do not vary. Likewise for the logical specification: coherent extension of a relation to a forwarding production can only be equivalent to what it forwards to, so no different order of composition could come to a different conclusion. This means the normal process for taking properties from H to an extended language $H \triangleleft E_1$ is exactly the same as for taking properties from $H \triangleleft E_1$ to a "rebased" language like $(H \triangleleft E_2) \triangleleft E_1$ (where we "reinterpret" E_1 onto a different host). In both cases, there are simply some new modules with some potential new forwarding productions, which we coherently extend relations over. Effectively we are able to "rebase" an extension onto an extended language, in the same manner and with the same properties as we extend a language. That the above is possible may be surprising, and so we wish to provide more concrete intuition.

Theorem 4.2 (Coherent extension of proofs). *Given a proof of a coherent property* $\forall t \in H.\ P(t)$*, and given a noninterfering* E*, then* $\forall t \in H \triangleleft E.\ P(t)$.

Proof. While this is a direct consequence of the definition of *noninterfering*, we wish to show it in a slightly more constructive way, to build intuition. Take our given proof over H (which we assume proceeds by induction on decorated trees) and remove all cases for forwarding productions. We can then uniformly prove the subgoals $P(t)$ for each forwarding production in the same way. The originating module of the forwarding production meets *noninterfering*, and so requirement 2 ensures each forwarding production preserves all coherent properties. Since $P(t)$ is coherent property, we can apply coherence to change the goal to showing $P(t.forward)$. Because we are proceeding by induction on decorated trees, this goal is discharged by our induction hypothesis. □

This gives us a procedure for *how* it is that proofs can be preserved by extensions. And with that in mind, it becomes clear how we are able to "rebase" extensions, including their proofs. We restrict proofs to just the cases of non-forwarding productions, which extensions can never alter, and then uniformly extend them over any forwarding production, regardless of origin. This procedure is not affected by whether we are extending a language (going from H to $H \triangleleft E_1$) or rebasing a language extension (going from $H \triangleleft E_1$ to $(H \triangleleft E_2) \triangleleft E_1$). All these operations can do is add more forwarding productions, which we can safely handle so long as all extensions are non-interfering. And so we can do either.

Theorem 4.3 (Non-interference of two extensions). *Given* H*,* E_1*, and* E_2 *such that*

- *All are noninterfering,*
- $\forall P \in \mathscr{P}(H).\ \forall t \in H.\ P(t)$*,*
- $\forall P \in \mathscr{P}(E_1).\ \forall t \in H \triangleleft E_1.\ P(t)$*, and*
- $\forall P \in \mathscr{P}(E_2).\ \forall t \in H \triangleleft E_2.\ P(t)$

Then $\forall P \in \mathscr{P}(H) \cup \mathscr{P}(E_1) \cup \mathscr{P}(E_2).\ \forall t \in H \triangleleft (E_1 \uplus_\emptyset E_2).\ P(t)$.

For a proof of this theorem, we refer to Kaminski [10]. We can then prove Theorem 2.1 by repeated application of this theorem.

5 Showing Non-interference

The definition of *noninterfering* requires two properties hold of an extension. The first requires that our extension's verification properties are all coherent. The second requires that our extension does not violate any coherent properties about the host (known or unknown). Let us work through this process for the example extension of Figure 3.

To show we meet the first requirement, we must show our properties are coherent. Because properties are incoherent only if they involve incoherent relations, this task reduces to showing that the relations involved are coherent. This proof turns out to be quite simple in this case. For example, the inferred case `lte_all`$(inner(inner(x, y), z), m)$ expands in a syntax directed way into `lte_all`$(x, m) \wedge$ `lte_all`$(y, m) \wedge$ `lte_all`(z, m), which is exactly what we specified for `three`. This repeats quite easily for `exists` and `eq_all`.

It might seem that we are now done with this task, but we have to handle the last remaining subtlety. Properties are coherent if their constituent relations are coherent, but we have only looked at the inductively defined relations of the verification. *Evaluation relations can also introduce incoherence.* In this case, however, it is simple to show they do not, as the values computed are identical between trees.

By showing that the stated properties are coherent, we have done half of what we need to do justify this extension as non-interfering. The remaining half is showing that the extension preserves all coherent properties. In general, it is not necessary to concern ourselves with manually extending the proofs of properties we have inherited from the host language. Instead, we need to show that the extension preserves all coherent properties, not just those we have in mind. The ones we have in mind will of course be included as a consequence, since we have shown them to still be coherent as our first step.

But this is a difficult problem: how can we be sure any arbitrary coherent property is preserved when we introduce a new forwarding production?

5.1 An "Unreasonable" Approach to Enforcement

On the other hand, how could an extension invalidate a coherent property? Coherent extension of a property only preserves coherence if all relations used are still coherent. When the property evaluates synthesized attributes, the property depends on relations that may become incoherent with a new extension. This suggests a solution: we can ensure that evaluation relations stay coherent.

In Figure 4, we can see a visual depiction of the potential space of interference with two language extensions shown. For the purposes of this diagram, assume there are no forwarding productions within H. This figure depicts an extension E_A introducing a new forwarding production and synthesized attribute, while E_B introduces a new forwarding production only. The shaded regions are authoritative.

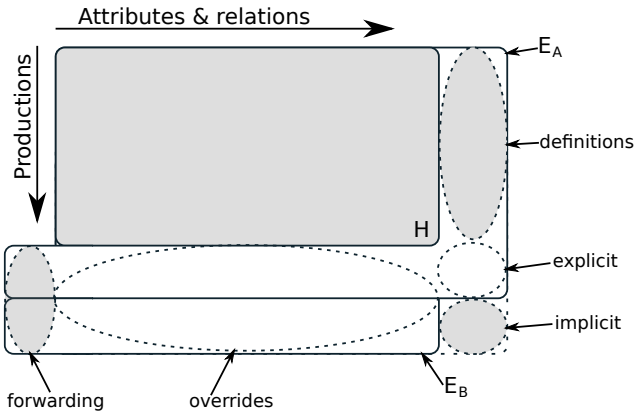

Figure 4. A table showing synthesized equations and relations on one axis, and productions on the other. E_A introduce both a new production and new attributes, while E_B introduces only a new production. Shaded regions cannot cause interference problems.

This includes the host (labeled H), the definitions of new attributes ("definitions") on just non-forwarding productions (where incoherence cannot possible arise), and forwarding equations ("forwarding"). Implicit copy equations (labeled "implicit") also obviously preserve coherence.

All of our problems stem from the other areas: labeled "overrides" and "explicit." These explicit equations on forwarding productions are a potential source of incoherence in evaluation relations. As a result, it is actually (mostly) possible to enforce non-interference syntactically: ban these equations on forwarding productions. The safe "implicit" region grows to cover the "explicit" and "overrides" spaces, leaving no possible source of interference. Or, more formally:

Theorem 5.1. *If there are no explicit equations on forwarding productions, then all evaluation relations are coherent.*

Theorem 5.2 (Coherent evaluation implies preservation of coherent properties)**.** *If all evaluation relations are coherent, then the extension preserves proofs of all coherent properties.*

Proofs appear in Kaminski [10], but the intuition here is straightforward: for a proof to fail on the extended language, there must be some forwarding production induction fails on, but all properties have had their coherence preserved, so it would also have to fail on some completely equivalent host language tree.

The "unreasonable" syntactic restriction is enough to ensure part 2 of non-interference, but it does not ensure part 1. It is always possible to state an incoherent property, no matter how well-behaved the extension's implementation. However, it does make it very difficult to write such an incoherent property.

Consider this: every test-case style property (i.e. something we could write as an expression in the language, even

introducing novel attributes) must automatically be coherent under this restriction. These kinds of properties do not involve any inductive relations except evaluation relations, so this claim is straightforward. This means an incoherent property must somehow be strongly stated: seemingly admitting the possibility of other extension's production existing, but then making claims about their behavior that should probably seem suspect, and getting away with it because (in isolation) there are no concrete counter examples. And so although this "unreasonable approach" does not ensure extensions do not violate part 1, it does make it harder.

6 Attribute Properties: More Useful Non-interference

In the previous section, we banned attribute equations on forwarding productions, in order to ensure evaluation relations were coherent. However, we typically want to place some equations on forwarding productions so that extensions "feel" like native language features. For example, extensions should generate error messages which refer to constructs in the extended program and not the generated code they forward to, as it the case with macro systems. This is why we called this approach "unreasonable."

6.1 Host Language Adaptation

Part of what makes the "unreasonable" approach unreasonable is that attribute values must be identical between the forwarding production and the tree it forwards to. This means, for example, that an attribute `errors` holding error messages on the forwarding tree must have the same value on the forwards-to tree. Thus we are unable to raise any error messages about an extension's custom syntax. The errors that can be raised are only those which the host language raises on the forwarded-to tree, which may be about generated code that does not appear in the written program.

One way to mitigate this problem is to make a minor change to the host language implementation. The `errors` attribute must always be the same, but if we have host language abstract syntax capable of raising arbitrary error messages, this would not be an issue. To do this, we introduce into the host language *error productions*, like that in Figure 5, that simply always raise the error messages they are provided with. Forwarding productions in extensions, instead of trying to override the value of the `errors` attribute (which is incoherent), can instead simply choose to forward to the error production when they needs to raise custom errors. We still demand strict equality on the part of the `errors` attribute, but it is no longer a problem.

6.2 Exploiting Coherent Equality

Although `errors` turned out to be a problem with an simple work around, what about the values that go into computing error messages? For example, an extension expression for

```
production error  e::Expr ::= msg::[Message]
{ e.errors = msg;  }

production bridge  e::Expr ::= x::ExtensionAST
{ forwards to if null(x.errors) then x.translation
              else error(x.errors);     }
```

Figure 5. An error production, and its typical use.

$$e.\texttt{host} = \begin{cases} e & \text{if } e \text{ is a terminal} \\ p(\overline{c.\texttt{host}}) & \text{if } e = p(\overline{c}),\ p \text{ non-forwarding} \\ e.\mathit{forward}.\texttt{host} & \text{if } e \text{ has forwarding root} \end{cases}$$

Figure 6. Transforming away forwarding productions.

constructing complex numbers may wish to have a `type` attribute that gives an extension type. That way, error messages involving types would refer to complex numbers, and not the underlying C `struct` that is an implementation detail of the extension. This requires different values, though these values are still related: what the extension type forwards to (`e.type.forward`) is the same type as what the extension expression forwards to has (`e.forward.type`).

Although the "unreasonable" approach bans attribute equations in forwarding productions, its arguments work equally well just so long as attribute values are identical. That is, so long at $t.s = t.\mathit{forward}.s$ is true (which is exactly what the forwarding copy equation does). For primitive types, equality is coherent, these values must therefore be equal, and so nothing changes. But strict equality is incoherent for higher-order [27] (and reference [8]) attributes. This opens up the possibility of different values for attributes between the two trees.

In Figure 6, we define a transformation `host` that eliminates all forwarding productions from a tree. This transformation has a very useful property:

Theorem 6.1. *Given any coherent property $P(t)$,*

$$\forall t.\ P(t) \iff P(t.\mathit{host})$$

Again, the proof is in Kaminski [10], but the intuition is that this is achieved by repeated application of coherence throughout the tree. With this tool, we can observe a useful property about the evaluation relation (R) for a tree-type (higher-order) synthesized attribute s:

$$\begin{aligned} R(t, x.\mathit{host}) &\iff R(t, x) \\ &\iff R(t.\mathit{forward}, x) \\ &\iff R(t.\mathit{forward}, x.\mathit{host}) \end{aligned}$$

The outer two if-and-only-ifs ($\iff$) are a result of this theorem, while the inner one is a result of coherence. As a result, if $t.s.\mathit{host} = t.\mathit{forward}.s.\mathit{host}$, this is enough to know that the evaluation relation is coherent.

This means that instead of explicit equations being pointless (because we need $t.s = t.forward.s$ to preserve coherence of evaluation), we actually can have a meaningfully different value computed for a forwarding production compared to what it forwards to. We have found a way to allow some kinds of useful explicit equations. The values cannot be arbitrary, though: the values must also be related by forwarding. A simple example of this kind of relationship is with types:

$$t.\texttt{type.host} = t.forward.\texttt{type.host}$$

In some sense, we can have different values, but they must still be equal modulo forwarding.

6.3 Closing the World

Although coherent equality gives us a mechanism for allowing differing values between trees, it actually has remarkably little effect by itself. The trouble is that although tree-valued attributes can differ, we can only observe the difference between trees by accessing attributes, which are either primitive or also tree-valued. A short inductive argument leads us to conclude that, in the end, we have to access some primitive-valued attribute to make an observation, and those have to be identical between the two trees!

Fortunately, there is one last trick in the toolbox. We have required that every evaluation relation be coherent and thus preserve all possible coherent properties. This is a good default assumption: we can never know what other property some extension might someday want to rely on. But what if there really is only a few coherent properties we could ever want to know about a specific attribute?

Take the example of "pretty-print," the `pp` attribute and a property like $t.pp.parse.host = t.host$. This property essentially claims that pretty printing composed with parsing is equivalent to the identity function (once again using `host` in order to make this modulo forwarding). That is essentially all we need to know about a pretty printing, extensions do not need to introduce new properties for this attribute.

So we can have the host language specify this property, and then close the world for this attribute, so that *only* this property need be preserved. No longer must extensions preserve *all* possible coherent properties about this attribute, just this one.

With this change, we can now have observably different values for attributes on extension syntax versus the host language syntax they forward to. Extension developers must only prove $t.pp.parse.host = t.host$ for each forwarding production (essentially showing the explicit pretty printing equations are correct). Combined with the trick for tree-valued attributes like `type` and `defs` (for declarations), this relaxation is quite useful. Error messages that use the pretty printing of types of subexpressions (or variables looked up in the environment) will now be about extension types and extension syntax.

To close the world for a property, we have some constraints:

1. The property in question still has to be coherent, ultimately.
2. Forwarding copy equations must still be valid. It is not possible to guarantee the explicit equation will always be the one used (consider, for example, the `id` extension transforming the tree away before it is accessed.)
3. We must be sure there are not any other properties we might want for this attribute.

In practice, attributes like `pp` are the only ones we apply this technique to, and so additional properties have not really been a concern.

6.4 Summary

Putting all these techniques together, we are able to making extensions feel like native language features. We do this by allowing extensions to perform arbitrary analysis, and then raise custom error messages. These error messages are able to contain extension information by virtue of types, definitions, and environments that can contain extension trees (which are equivalent to some host trees.) But this hard equivalence requirement can be relaxed for `pp` which is the essential feature that allows extension types (for example) to appear in error messages.

7 Property Testing

Although coherence is inherently about properties and verifying language extensions, we do not often do verification in practice. Fortunately, the attribute properties approach gives us a satisfying way to detect interference through ordinary property testing—without writing theorems or proofs.

We are able to show non-interference by simply not stating any incoherent properties and by ensuring attribute evaluation relations are coherent. We showed that it is actually hard (though not impossible) to write incoherent properties when you have coherent evaluation relations. Evaluation relations can be kept coherent by ensuring simple equalities hold for every single attribute. These can be hard equality ($t.errors = t.forward.errors$) or they can be the relaxed variety we develop in the previous section. These equalities are easily amenable to QuickCheck-style [3] randomized property testing.

We will also apply QuickCheck-style testing in a somewhat novel way. We might wish to check that $t.pp.parse.host = t.host$ where t are expressions. In the usual QuickCheck style, we would generate random trees t of that nonterminal type, and check that each property holds on each of those trees. Instead, while we still specify these properties in association with the nonterminal, we will emit QuickCheck-style tests on a per-production basis. Values will be randomly generated for each *child* of each production (and for the inherited

attributes given to the production), and all the nonterminal's properties will be checked on the resulting decorated tree rooted in that production. In other words, we will specify properties at the level of a type, but emit tests at the level of each constructor of that type.

With this approach, properties can be specified by the host language, and new tests will be applied to each production introduced by an extension. By testing these attribute properties, extensions will automatically be checked for interference violations. Even properties as simple as actually testing for $t.errors = t.forward.errors$ can discover interference problems.

8 Application to a Real Compiler

We wish to investigate two questions: first, does enforcing coherence still permit useful extensions, and second, is the testing-based approach effective in spotting violations? To that end, we apply this property testing framework to ableC, a specification of a C11 compiler front-end written in Silver. ableC offers a good, though small, natural experiment for detecting interference. It comes with several language extensions, but some were implemented prior to beginning our work on non-interference. We know from inspection that one of those extensions (one version of an extension introducing algebraic datatypes and pattern matching) has interference problems. (We call this a *small* experiment because there is just one extension of this sort, not because a C front-end is small.) And so we look to see if the testing approach successfully finds this violation.

As it turns out, the testing approach identified the interfering extension in 100 out of 100 trials. Inspecting these failures, we discovered that there were, in fact, two sources of incoherence, while we had only discovered one of them by inspection beforehand. The reason we found the non-interference violations 100% of the time was that one of these violations always occurred (the production was always adding more declarations than the tree it forwarded to, so the lists had different length.) When we restricted ourselves to looking only for the other violation, the testing approach discovered the failure in 86 out of 100 trial runs. This was despite only generating 10 trees per production, a relatively low number for random testing, and so we consider this quite good. This particular violation was explicitly overriding the errors attribute, rather than forwarding to an error production, and so it was occasionally hidden when the check did not raise an error.

While this is a modest evaluation of the ability for property testing to find interference problems, ableC also serves as an evaluation of the permissivity of non-interfering extensions. With ableC we were able to develop a number of useful extensions, all without violating our non-interference restrictions. We have built a "regular expression literals" extension and a matrix operations extension, we have added algebraic data types and pattern matching to C, and we developed an extension to aid in the use of the Matlab foreign function interface.

The experience we gained writing these extensions makes it easy to say where non-interference leads to restrictions on what extensions are possible. The central problem is that *there must be an equivalent host language tree* wherever an extension language construct appears. For ableC, this means although we have a good extension for algebraic data types and pattern matching, these data types cannot be parameterized. That is, we cannot handle a type like `List<a>`. C has no type representation for parameterized types, so this becomes impossible. These sorts of restrictions are host language-relative, however. This has lead to the desire to amend the ableC host language with some internal features that enable additional classes of extensions.

9 Related Work

The notion of interference of language extensions is similar to that of feature interaction (e.g. Jackson and Zave [9]) in software product lines [1, 13]. A primary difference is that in software product lines it is typically assumed that an expert in the domain of the software is involved in the composition of various features, and can thus intervene (even though this is undesirable) if some undesired interaction is detected. This differs from our aims in which a programmer that is not an expert in language design or compiler implementation determines what extensions or features are to be composed with the host language and thus intervention by an expert is not possible. We wish to ensure there are no invalid configurations.

The testing of context free grammars [16] and attribute grammars [17] has been studied before. But this work investigates issues of test coverage and focuses on a general notion of correctness and not on non-interference of language extensions. Instead of a testing-based approach to ensuring non-interference, formal verification of these properties is possible as well. These properties could perhaps be expressed in a dependent type system and then verified, for example building on a dependently typed attribute grammar [19]. This particular approach encodes attribute grammars in the dependently typed language Agda [20]. In this sort of framework, non-interference could be proven.

While most extensible language frameworks seem to value expressiveness over reliable composition, not all do. Wyvern [22] and VerseML [21] are the only extensible language system besides Silver, to our knowledge, that support *reliable* composition of independently developed language extensions, at least syntactically, without abandoning parsing in favor of projectional editing. However, their approach does not accommodate introducing new analysis of the host language, and so is similar to syntactic macros in being non-interfering, but more limited.

SoundX [18] takes an interesting twist on macros by defining a desugaring over typing derivations rather than syntax trees. This gives it added power above normal macros systems: the ability to make use of type information in expansion, and the ability to define that type information for the new syntax. Like Wyvern however, this approach is limited because you cannot define new analysis ("judgements") over the host language. As such, the power of this system hovers somewhere between a macro system and our approach. It is able to achieve non-interference by ensuring that extensions are desugared before they are able to interact. Its primary contribution is a process for automatically proving that desugarings introduce no type errors, and so can be safely applied. This approach is like a special case of coherence: the type rule and rewrite rule are required to be correct with respect to each other.

9.1 Relationship to Macro Systems

The "unreasonable" approach we describe can be compared directly to many different kinds of macro systems. Macro systems that do not permit case analysis of subtrees clearly enforce non-interference, though this is a significant restriction to the capabilities of extensions. If a system cannot analyze subtrees, then this is equivalent to not only banning explicit equations on forwarding productions, but also forbidding accessing synthesized attributes on children as well.

When subtree analysis is permitted, it is still possible to do so in a non-interfering way. Bottom-up expansion of macros, for instance, would mean extensions (in the form of macros) are non-interfering because subtree analysis could only ever examine host language trees (all macros in subtrees would already be expanded away). This would be similar to the unreasonable approach in capability.

In practice, many macro systems use top-down evaluation order, in part for lack of grammar: the system does not know what pieces of a macro invocation's arguments might be host language constructs, within which we might looks for more macros. The lack of grammar makes it difficult to be disciplined in trying to "play nice" with other macros: where might another unknown macro appear, so that (e.g.) we should anticipate evaluating it before matching on the result? (Indeed, the answer could be "almost anywhere.")

Still, a top-down approach can be slightly more permissive than outright banning subtree analysis while still managing non-interference. Racket [7], although it has no resolution to the interference problem, is more capable than other macro systems if one attempts to observe the "banning subtree analysis" discipline. The reason is that, while many macros systems treat trees as simple objects, Racket ascribes a minimal (name-binding) semantics to them, in order to support hygiene [15]. Instead of being equivalent to banning accessing of all synthesized attributes on children, this would instead permit access to a limited set of host language attributes.

10 Future Work

This development made a few critical assumptions, all of which could be relaxed by future work. First, we generally assumed that our properties are proved by induction on decorated trees. Some properties are proved by induction on other types, for example by induction on the derivation of a well-typedness relation (rather than by induction on a term or type). It may be possible to develop a framework for reasoning about such proofs.

Likewise, we restricted propositions (and relations) to a very simplified form. We only permit universal quantification, and we did not allow nesting. Generalizing the notions of "coherent proposition" and "coherent relation" to remove these restrictions would be a useful future direction. Doing so would likely require picking a particular logic, however.

There may be other approaches to non-interference, as we somewhat implicitly noted when making a comparison to macro systems. Our definition of *noninterfering* is just one way of achieving this result, making use of coherence.

This solution, together with previous work on composing grammars and attribute grammars, offers a candidate solution for composing language extensions into a working compiler, without the possibility of conflict. Although an exciting milestone in this research program, there remains more research problems to be solved. It is not enough to have a working compiler, we also need other language tooling, such as debuggers, documentation generators, and IDE tooling. These problems remain future work.

11 Discussion

Coherence in the end provides us with a precise definition of what it means to forward to a "semantically equivalent" tree. We have shown that all coherent properties can be preserved by extensions, and so when extensions rely only on coherent properties, they can compose reliably. Ensuring our extensions are non-interfering places restrictions on the kinds of extensions we can make. These restrictions largely take two forms: we cannot over-promise with incoherent properties, and we cannot violate coherent properties with our extension's behavior.

Returning to our opening example of the `taint` extension, we see an example of an extension that is not possible, due to inherent interference. Note, however, that this extension is not possible *for this particular (primitive) host language*. The host language is so simple, there is no alternative implementation strategy available except interfering ones. However, this is a host language-relative restriction. A language like Java, which has annotations built into the host language, would have no trouble implementing an analysis of this sort.

The testing methodology we propose is imperfect. However, we believe it can effectively inform well-meaning extension developers of mistakes they are making before they have invested too much into an unworkable approach. As for

more hostile extension developers, we still have a notion of blame for interference problems that arise in practice. Users should not be confused about the cause of their problems.

We also note that the typical language extension development methodology is unlikely to yield interference bugs. A typical language extension will start its development in a "macro-like" fashion, without explicit equations on forwarding productions. We called this "unreasonable" ultimately because it does not permit language extensions to behave like built-in language features: error messages in particular suffer. But this problem does not matter during initial development, it only matters when the extension is being "polished" for delivery to end-users. However it does mean that extensions are unlikely to rely on interfering behavior in the first place, especially with the interference tests immediately warning extension developers of their early attempts to do so. As a result, we believe this approach to preventing interference will be quite effective.

As a final note, this system provides a precise boundary between language features that can be introduced as extensions, and those that would instead require modification of the underlying host language. We believe this can be a very helpful tool in future language design, by focusing attention on core features that truly change the language.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1047961. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

References

[1] S. Apel and C. Kästner. 2009. An overview of feature-oriented software development. *Journal of Object Technology* 8, 4 (2009), 49–84.

[2] R. S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica* 21 (January 1984), 239–250.

[3] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. https://doi.org/10.1145/357766.351266

[4] T. Ekman and G. Hedin. 2007. The JastAdd extensible Java compiler. In *Proc. Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/1297027.1297029

[5] S. Erdweg, P. G. Giarrusso, and T. Rendel. 2012. Language composition untangled. In *Proc. Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. ACM, New York, NY, USA, 7:1–7:8. https://doi.org/10.1145/2427048.2427055

[6] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proc. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

[7] M. Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Inc. http://racket-lang.org/tr1/.

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

[9] M. Jackson and P. Zave. 1998. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering* 24, 10 (1998), 831–847.

[10] T Kaminski. 2017. *Reliably composable language extensions.* Ph.D. Dissertation. University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA. http://hdl.handle.net/11299/188954

[11] T. Kaminski and E. Van Wyk. 2011. Integrating attribute grammar and functional programming language features. In *Proc. Conf. on Software Language Engineering (SLE '11)*, Vol. 6940. Springer, 263–282.

[12] T. Kaminski and E. Van Wyk. 2012. Modular well-definedness analysis for attribute grammars. In *Proc. Conf. on Software Language Engineering (SLE '12)*, Vol. 7745. Springer, 352–371.

[13] C. Kästner, S. Apel, and K. Ostermann. 2011. The Road to Feature Modularity?. In *Proc. Workshop on Feature-Oriented Software Development (FOSD '11)*. ACM, New York, NY, USA, 5:1–5:8.

[14] D. E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. Corrections in **5**(1971) pp. 95–96.

[15] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. 1986. Hygienic macro expansion. In *Proc. Conf. on LISP and Functional Programming*. ACM Press, 151–161. https://doi.org/10.1145/319838.319859

[16] R. Lämmel. 2001. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001 (LNCS)*, Vol. 2029. Springer-Verlag, 201–216. http://homepages.cwi.nl/~ralf/fase01/

[17] R. Lämmel and J. Harm. 2000. Testing Attribute Grammars. In *Proc. Workshop on Attribute Grammars and their Applications (WAGA 2000)*. 79–98. http://citeseer.ist.psu.edu/297413.html

[18] F. Lorenzen and S. Erdweg. 2016. Sound Type-dependent Syntactic Language Extension. In *Proc. Conf. on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 204–216. https://doi.org/10.1145/2837614.2837644

[19] A. Middelkoop, A. Dijkstra, and S.D. Swierstra. 2011. Dependently Typed Attribute Grammars. In *Implementation and Application of Functional Languages*, Jurriaan Hage and MarcoT. MorazÃąn (Eds.). Lecture Notes in Computer Science, Vol. 6647. Springer Berlin Heidelberg, 105–120. https://doi.org/10.1007/978-3-642-24276-2_7

[20] U. Norell. 2009. Dependently Typed Programming in Agda. In *Proc. Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, 1–2.

[21] C. Omar. 2017. *Reasonably Programmable Syntax.* Ph.D. Dissertation. Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, USA. https://www.cs.cmu.edu/~comar/omar-thesis.pdf

[22] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. 2014. Safely Composable Type-Specific Languages. In *Proc. European Conf. on Object-Oriented Programming (ECOOP '14)*. Springer-Verlag New York, Inc., New York, NY, USA, 105–130. https://doi.org/10.1007/978-3-662-44202-9_5

[23] A. C. Schwerdfeger and E. Van Wyk. 2009. Verifiable composition of deterministic grammars. In *Proc. Conf. on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1542476.1542499

[24] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54.

[25] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proc. Conf. on Compiler Construction (LNCS)*, Vol. 2304. 128–142.

[26] E. Van Wyk and A. C. Schwerdfeger. 2007. Context-aware scanning for parsing extensible languages. In *Proc. Conf. on Generative Programming and Component Engineering (GPCE '07)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1289971.1289983

[27] H. Vogt, S. D. Swierstra, and M. F. Kuiper. 1989. Higher-order Attribute Grammars. In *Proc. Conf. on Programming Language Design and Implementation (PLDI '89)*. ACM Press, 131–145.