The Next 700 Semantics: A Research Challenge

Shriram Krishnamurthi
Brown University, Providence, RI, USA
sk@cs.brown.edu

Benjamin S. Lerner
Northeastern University, Boston, MA, USA
blerner@ccs.neu.edu

Liam Elberty
Unaffiliated

Abstract

Modern systems consist of large numbers of languages, frameworks, libraries, APIs, and more. Each has characteristic behavior and data. Capturing these in semantics is valuable not only for understanding them but also essential for formal treatment (such as proofs). Unfortunately, most of these systems are defined primarily through implementations, which means the semantics needs to be *learned*. We describe the problem of learning a semantics, provide a structuring process that is of potential value, and also outline our failed attempts at achieving this so far.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Language features; Software and its engineering → Semantics; Software and its engineering → Formal language definitions

Keywords and phrases Programming languages, desugaring, semantics, testing

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.9

Funding This work was partially supported by the US National Science Foundation and Brown University while all authors were at Brown University.

Acknowledgements The authors thank Eugene Charniak and Kevin Knight for useful conversations. The reviewers provided useful feedback that improved the presentation.

1 Motivation

Semantics is central to the trade of programming language researchers and practitioners. A useful semantics for a system helps us understand it better, guides us in building tools, and provides us a basis for proving valuable properties such as soundness and completeness.

However, there are far fewer semantics than there are objects that need them. A few languages (notably Scheme [5] and Standard ML [16]) are accompanied by a reasonably thorough semantic description, but even these rarely cover the behavior of the language that users use (such as libraries). Nevertheless, these semantics are valuable in that they provide users a standpoint from which to evaluate an implementation: if it diverges from the semantics, they can argue that the fault definitively lies with the implementation, since the semantics was given as part of the process of defining the very language.

However, the need for and value of a semantics extends well beyond what we would conventionally call a "language". Consider, for instance, a Web program: there is value to having a semantics for every layer in the stack, from the hardware's instruction set to the operating system's API to the browser's APIs to the JavaScript implemented by the browser to the numerous frameworks that sit atop JavaScript and effectively create their own ontologies (for page traversal, page transformation, etc.) to the libraries that sit atop frameworks. Even if we draw a line at, say, the browser, we still need meaningful semantics for the upper layers to make sense of what a program does. Thus, reasoning about the security of a Web program effectively demands a "full-stack semantics", as it were.

Unfortunately, virtually none of these levels enjoy the benefits of originating in a formal semantics. They are instead defined by an ad hoc collection of implementations, documents, and test suites. The task of the semanticist then becomes one of *reverse engineering* a semantics in a way that ensures its *conformance* with reality: after all, if there is a divergence between the semantics and the implementation, the implementors would not agree that they are to blame and accordingly "fix" their systems. This is essentially a bottom-up discipline akin to that practiced in the natural sciences.

2 Tested Semantics and a Small Core

Over the past decade, several researchers have begun to create so-called *tested semantics*: one where the semantics shows conformance with the real-world artifact [1, 2, 3, 4, 7, 8, 10, 11, 12, 13, 15, 17, 18, 19]. A notable line of work has addressed the semantics of "scripting" languages, which are ostensibly defined by trying to provide a meaning to as many utterances as possible, as opposed to the less libertine behavior of more academic languages. Because of their enormous set of behaviors, many of which are stumbled into by programmers and exploited by attackers, it becomes especially important that a semantics conform to reality rather than to a researcher's mental idea of how the language *ought* to be.

However, not all tested semantics are equal in value. Some semantics come in the form of hundreds of rules. In contrast, other projects (such as [11, 18, 19]) have followed the pattern shown in Figure 1. In this style, the semantics is broken into two components. The *surface* language P is reduced to an essential *core* version, λ_P, and all P programs are mapped to λ_P using a *desugaring* function. We use the term "desugaring" to evoke the idea that most of the surface language can be easily rexpressed (i.e., in a structured, recursive, compositional manner) as idioms of the core language, and can therefore be dismissed as ergonomic convenience rather than as novel features requiring analysis. (Strictly speaking, this term is slightly misleading, since the core language λ_P might be a *different* language rather than a subset of P itself; nevertheless, this term captures our design intent better

$$\begin{tikzcd}
P \text{ Program} \arrow[r, "\text{desugar}"] \arrow[d, "P \text{ evaluator}"'] & \lambda_P \text{ Program} \arrow[d, "\lambda_P \text{ evaluator}"] \\
P \text{ out} \arrow[r, leftrightarrow, "\text{compare}"'] & \lambda_P \text{ out}
\end{tikzcd}$$

Figure 1 Testing Strategy for λ_P.

than the more general “compilation”.) It is usually very straightforward to implement an interpreter for λ_P (an appropriately-sized exercise for undergraduates), and composing this implementation with desugaring gives us a new implementation of P. Of course, P already has existing implementations. We can therefore then compare the results of running the two implementations on a suitably large suite of programs, tweaking desugaring and λ_P until the result is in harmony with the existing behavior of P. Since existing P implementations may have values with opaque internal structure, we may not be able to translate them directly into λ_P values; instead we may have to use coarser notions of equivalence (such as textually comparing the output of two programs), or even more abstract ones, to check that our desugaring preserves intended behavior.

This style has significant advantages over the brute-force generation of hundreds of rules. First, it helps us reduce the complexity of the overall problem. Second, it forces us to find general patterns in the language, which reduces the likelihood of overfitting to a particular set of programs against which we run tests. Third, it gives us λ_P, which is a valuable artifact in its own right. From the perspective of constructing proofs, λ_P – which might have five to twenty constructs, in the typical lambda calculus fashion – is a far more accessible target than having hundreds of rules. Additionally, λ_P reduces P to an essence (there may, of course, be multiple λ_P’s that provide different essential views of the language), thus accomplishing one of the jobs of a semanticist: to give *insight*, not merely brutally reconstruct reality. In that light, the desugaring function too is a useful artifact, and we want it to be both readable and terse. An unreadable desugaring function yields no insight into the semantics of constructs in P, and in fact casts suspicion on whether all behaviors have been faithfully captured. Likewise, a readable function that nevertheless has inordinately complicated behavior indicates that the proposed λ_P does a poor job of modeling the intrinsically “interesting” parts of P.

The problem remains: how to create these artifacts? The first author has been involved in several tested semantics efforts, and their labor demands are daunting: for instance, that for Python [19] took approximately 72 person-months, including recruiting participants from a MOOC. This simply does not scale. Can we instead *learn* these semantics?

3 Learning a Semantics

Before we apply machine learning techniques, it helps to agree on the appropriate role for human versus machine. It may be tempting to simply let machine-learning techniques run wild and reconstruct a set of rules that match an existing implementation, but as we have argued in Section 2, this would be counter-productive. Instead, we claim Figure 1 provides us with a clear guide. The place for an expert is in the design of the core λ_P, using an understanding of (a) the surface P itself, (b) general design principles in the design of semantics, and (c) eventual goals for the use of the semantics. In contrast, the act of writing desugaring is where the grunt work lies, and this is the part that would best be automated. In addition, in our experience, engineering λ_P is both interesting and does not consume effort that is not insightful, whereas constructing desugaring can take months and is only very rarely insightful.

In our ideal workflow, then, the semantics engineer (who presumably understands P well) proposes an initial design for λ_P, alongside an implementation for it (which, depending on how λ_P is written, may even be obtained automatically). They also provide a conformance suite, such as existing test suites for widely-used P implementations.[1] A learning system then attempts to construct the desugaring function.

The space of desugaring functions is of course unbounded. We believe a useful constraint is given by Felleisen's expressiveness framework [9], which (loosely) says that if a feature can be expressed by local macro transformations (i.e. structural, compositional changes), it is not "expressive". We believe that (unless a semantics engineer explicitly chooses otherwise) desugaring should *not* be expressive: if it is, a non-trivial feature of the language is not captured by λ_P.

So long as the search for desugaring succeeds, all is well. But when the search has proceeded for too long, it seems reasonable to present the input for which no reasonable desugaring could be found: the semanticist is then in a position either to realize that λ_P must grow, or to guide the desugaring system with a hint, or even to provide the desugaring rule directly. Based on our experience manually constructing semantics in this style, these interruptions grow far less frequent over time, and the rare situations where a desugaring can't easily be found are usually instructive about the nature of P.

Problem Definition

The essential problem, then, is as follows. Assuming we can obtain a parser for P (usually to be found inside P implementations) and similarly for λ_P (which should be straightforward), the goal is to *learn a translation from P trees to λ_P trees* – a tree transducer [6, chapter 6] – such that Figure 1 ensues. This problem is not new: it lies at the heart of natural language translation (NLT).

A critical difference, however, is in the setup of the problem. In NLT, one is typically translating, say, French to German. To do this, the learning system is given millions of pairs of the "same" sentences in both languages: obtained from proceedings of national bodies, book translations, and so on. From this large corpus of examples, an algorithm infers a transducer. However, there is no oracle for ground truth; some use has been made of crowdsourcing for this purpose [20], but it may not be straightforward to handle millions of new proposed translations this way, and it would anyway be slow and somewhat unreliable.

In programming languages, we have the dual situation. The semantics engineer will not have the patience to write more than a few dozen examples of P-to-λ_P translations. However, we *do* have ground truth! A system can easily generate new instances of P programs and run them; and also pass them through the candidate desugarer, compute their answer under the semantics, and compare the two answers. If this succeeds, we have gained further trust in the desugaring. If it fails, we have a P program for which our desugaring must be revised. The system can run this loop automatically, correctly, and at scale (with extreme parallelization).

Unfortunately, there seems to be little literature on this problem in the NLT literature. This is perhaps unsurprising: there is little use in natural languages in creating techniques that assume perfect, mechanized ground truth. Thus, our dual problem is not only valuable but also technically interesting.

[1] Of course, different implementations may differ slightly. This only further reinforces the need for a learning-based approach, because innocent programmers might stumble on these differences while adversaries exploit them.

$$
\begin{array}{rrl}
\mathtt{x}, \mathtt{y} \in & \mathit{SIdentifiers} & \\
n \in & \mathit{Number} & \\
s \in & \mathit{Str} & \\
\mathtt{o} \in & \mathit{SOp} & ::= \mathtt{0-} \mid \mathtt{not} \mid \mathtt{+} \mid \mathtt{-} \mid \mathtt{and} \mid \mathtt{or} \mid \mathtt{<} \mid \mathtt{>} \\
\mathtt{e} \in & \mathit{SExp} & ::= \mathtt{SVar(x)} \mid \mathtt{SPrim(o,\ \vec{e})} \mid \mathtt{SBetween(e_1,\ e_2,\ e_3)} \\
& & \mid \mathtt{SNum(}n\mathtt{)} \mid \mathtt{SStr(}s\mathtt{)} \mid \mathtt{STrue} \mid \mathtt{SFalse} \\
& & \mid \mathtt{SIf(e_1,\ e_2,\ e_3)} \mid \mathtt{SLet(x,\ e_1,\ e_2)} \mid \mathtt{SLetRec(x,\ e_1,\ e_2)} \\
& & \mid \mathtt{SLam(\vec{x},\ e)} \mid \mathtt{SApp(e,\ \vec{e})} \mid \mathtt{SAssign(x,\ e)} \\
& & \mid \mathtt{SList(\vec{e})} \mid \mathtt{SListCase(e_1,\ e_2,\ e_3)} \mid \mathtt{SFor(e_1,\ \vec{b},\ e_2)} \\
\mathtt{b} \in & \mathit{SForBind} & ::= \mathtt{SFBind(x,\ e)} \\
\hline
\mathtt{x}, \mathtt{y} \in & \mathit{CIdentifiers} & \\
n \in & \mathit{Number} & \\
s \in & \mathit{Str} & \\
b \in & \mathit{Bool} & \\
\mathtt{o} \in & \mathit{COp} & ::= \mathtt{0-} \mid \mathtt{not} \mid \mathtt{+} \mid \mathtt{-} \mid \mathtt{and} \mid \mathtt{or} \mid \mathtt{<} \mid \mathtt{>} \\
\mathtt{e} \in & \mathit{CExp} & ::= \mathtt{CVar(x)} \mid \mathtt{CPrim1(o,\ e)} \mid \mathtt{CPrim2(o,\ e_1,\ e_2)} \\
& & \mid \mathtt{CNum(}n\mathtt{)} \mid \mathtt{CStr(}s\mathtt{)} \mid \mathtt{CBool(}b\mathtt{)} \\
& & \mid \mathtt{CIf(e_1,\ e_2,\ e_3)} \mid \mathtt{CLet(x,\ e_1,\ e_2)} \mid \mathtt{CLetRec(x,\ e_1,\ e_2)} \\
& & \mid \mathtt{CLam(\vec{x},\ e)} \mid \mathtt{CApp(e,\ \vec{e})} \mid \mathtt{CAssign(x,\ e)} \\
& & \mid \mathtt{CList(\vec{e})} \mid \mathtt{CListCase(e_1,\ e_2,\ e_3)}
\end{array}
$$

Figure 2 Simple Source and Target Core languages.

4 Initial Non-Progress

In addition to formulating the problem, to date we have tried four different existing techniques, with no real success. The rest of this paper documents what we have learned so far, but we caution that, since we are not experts in this area, one should not read too much into our failure.

4.1 Notation

We denote `source programs` (previously P) in red monospaced text, `core programs` (previously λ_P) in blue monospaced text, and the desugarers themselves in black sans-serif font. Additionally, we prefix the names of source AST nodes with a `S`, and likewise prefix core AST node names with a `C`. We use • to denote a hole in a tree, into which another tree may be merged; when defining a desugarer, subscripts are used to indicate corresponding holes in source and core terms. Finally, for expressions with arbitrary arity, we use $\cdots$ to indicate the repetition of the preceding term (much as $\ldots$ are used in Racket macros).

4.2 Example Languages

Consider the languages in Figure 2. We have tried to learn a desugaring from the source to the core. Most expression forms are straightforward to translate, and there are some characteristic "impedance mismatches" between the two languages. The construction `SBetween(lo, mid, hi)` is intended to be the chained comparison $lo \leq mid \leq hi$, where each expression is evaluated at most once. Primitive operations are represented in the source via an

$$
\begin{aligned}
\mathsf{ds}(\texttt{SVar(}\bullet\texttt{)}) &= \texttt{CVar(}\bullet\texttt{)}\\
\mathsf{ds}(\texttt{SPrim(}\bullet_1\texttt{, [}\bullet_2\texttt{])}) &= \texttt{CPrim1(}\bullet_1\texttt{, }\bullet_2\texttt{)}\\
\mathsf{ds}(\texttt{SPrim(}\bullet_1\texttt{, [}\bullet_2\texttt{, }\bullet_3\texttt{])}) &= \texttt{CPrim2(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}\\
\mathsf{ds}(\texttt{SBetween(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CLet(\%t1, }\bullet_1\texttt{, CLet(\%t2, }\bullet_2\texttt{, CLet(\%t3, }\bullet_3\texttt{,}\\
&\qquad\quad \texttt{CPrim2(and, CPrim2(<, \%t1, \%t2),}\\
&\qquad\qquad\qquad\qquad \texttt{CPrim2(<, \%t2, \%t3)))))}\\
\mathsf{ds}(\texttt{SNum(}\bullet\texttt{)}) &= \texttt{CNum(}\bullet\texttt{)}\\
\mathsf{ds}(\texttt{SStr(}\bullet\texttt{)}) &= \texttt{CStr(}\bullet\texttt{)}\\
\mathsf{ds}(\texttt{STrue}) &= \texttt{CBool(true)}\\
\mathsf{ds}(\texttt{SFalse}) &= \texttt{CBool(false)}\\
\mathsf{ds}(\texttt{SIf(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CIf(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}\\
\mathsf{ds}(\texttt{SLet(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CLet(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}\\
\mathsf{ds}(\texttt{SLetRec(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CLetRec(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}\\
\mathsf{ds}(\texttt{SLam([}\bullet_1\cdots\texttt{], }\bullet_2\texttt{)}) &= \texttt{CLam([}\bullet_1\cdots\texttt{], }\bullet_2\texttt{)}\\
\mathsf{ds}(\texttt{SApp(}\bullet_1\texttt{, [}\bullet_2\cdots\texttt{])}) &= \texttt{CApp(}\bullet_1\texttt{, [}\bullet_2\cdots\texttt{])}\\
\mathsf{ds}(\texttt{SAssign(}\bullet_1\texttt{, }\bullet_2\texttt{)}) &= \texttt{CAssign(}\bullet_1\texttt{, }\bullet_2\texttt{)}\\
\mathsf{ds}(\texttt{SList(}\bullet\ \cdots\texttt{)}) &= \texttt{CList(}\bullet\ \cdots\texttt{)}\\
\mathsf{ds}(\texttt{SListCase(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CListCase(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}\\
\mathsf{ds}(\texttt{SFor(}\bullet_1\texttt{, [SFBind(}\bullet_2\texttt{, }\bullet_3\texttt{)}\cdots\texttt{], }\bullet_4\texttt{)}) &= \texttt{CApp(}\bullet_1\texttt{, SLam([}\bullet_2\cdots\texttt{], }\bullet_4\texttt{), [}\bullet_3\cdots\texttt{])}
\end{aligned}
$$

Figure 3 Intended ground-truth translation from Simple Source to Target Core. `%t1` etc are fresh generated names for temporary variables.

arbitrary-arity `SPrim` constructor, and must be translated to either unary `CPrim1` or binary `CPrim2` constructions. Booleans `STrue` and `SFalse` must be translated to actual booleans inside a `CBool` constructor, and other primitive values are simply carried across. The `SFor` construction is unique to the source language, and must be translated into simpler terms in the core. Our intended translation is given in Figure 3.

4.3 First attempt: Naïve Tree Matching

4.3.1 Essential Ideas

As an initial guess, we posit that translating the AST of one language into a corresponding tree in another language will result in a tree "of roughly the same shape": that is, a source node with N children will desugar to a tree with N disjoint descendants that correspond to the source node's children, respectively. For example, $\texttt{SPrim(+, [}\bullet_1\texttt{, }\bullet_2\texttt{])}$ might correspond to $\texttt{CLet(\%t1, }\bullet_1\texttt{, CLet(\%t2, }\bullet_2\texttt{, CPrim2(+, \%t1, \%t2)))}$.

With this as a guide, our initial attempt is to learn a tree transducer that transforms source nodes into core subtrees in a uniform, top-down manner. Learning this transducer from a corpus of tests amounts to learning an explanation for how each test output was produced from its input. We can simplify the problem further under our assumption that tree shape is roughly preserved.

4.3.2 Worked Example

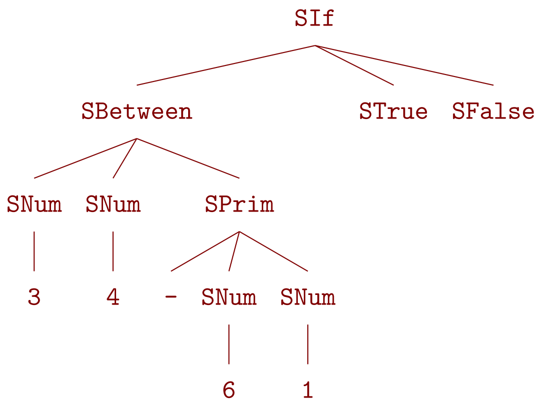
Consider the source program

```
SIf(SBetween(SNum 3, SNum 4, SPrim(-, [SNum 6, SNum 1])), STrue, SFalse)
```

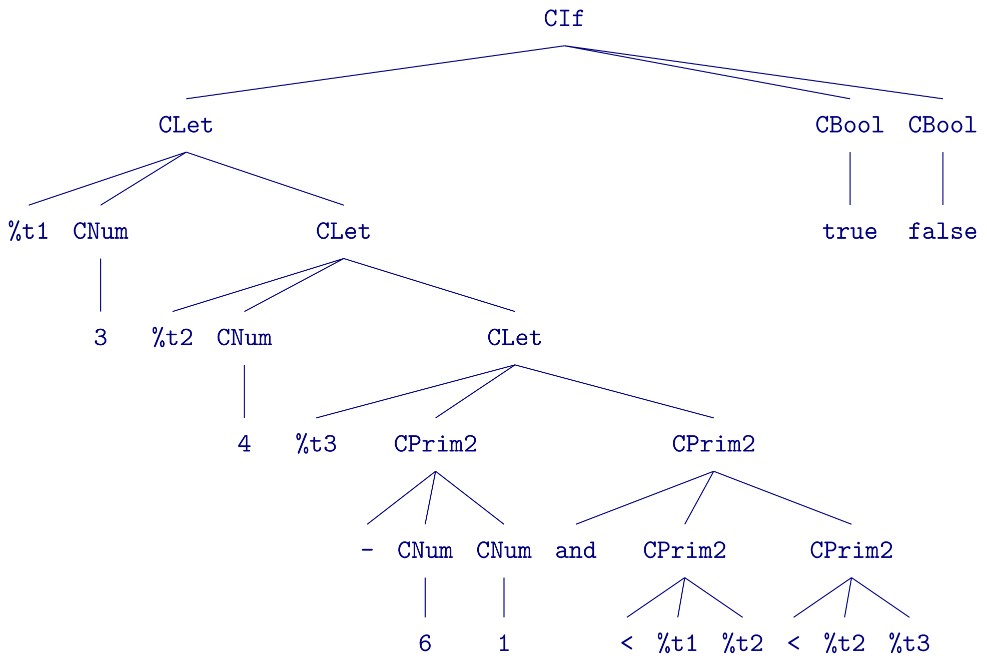
Our ground-truth desugaring results in the desugared expression

```
CIf(CLet(%t1, CNum 3, CLet(%t2, CNum 4, CLet(%t3, CPrim2(-, CNum 6, CNum 1),
          CPrim2(and, CPrim2(<, %t1, %t2), CPrim2(<, %t2, %t3))))),
    CBool(true), CBool(false))
```

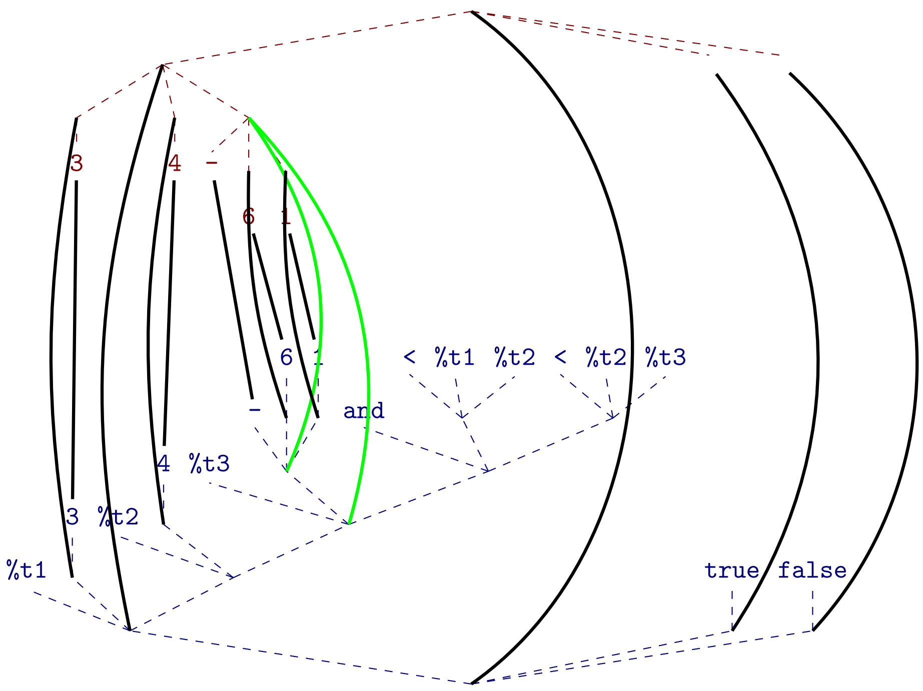
Graphically, these two trees are:



and



Our goal is to compute all possible alignments between these two trees, ignoring the labels on the internal nodes. (We draw the core tree inverted, to make drawing the alignments easier to read.) An alignment is simply a mapping of nodes in one tree to nodes in the other that preserves ancestry: if a source node S maps to a core node C, then all descendants of S must map to descendants of C.



Note that in this example, one node (the root of SPrim(-, [SNum 6, SNum 1])) has two possible alignments (shown in green). This leads to two possible desugarings for SBetween and SPrim nodes: the correct one presented earlier, and the incorrect

```
ds(SBetween(•₁, •₂, •₃)) ⇒ CLet(%t1, •₁, CLet(%t2, •₂, •₃))
 ds(SPrim(•₁, [•₂, •₃])) ⇒ CLet(%t3, CPrim2(•₁, •₂, •₃)
                               CPrim2(and,
                                      CPrim2(<, %t1, %t2),
                                      CPrim2(<, %t2, %t3)))
```

Such a mistake is possible in this test case, because only one instance of either SBetween or SPrim is seen. With a larger test corpus, this mistaken desugaring could only appear as part of a non-deterministic desugarer, and so would eventually be discarded by the algorithm.

4.3.3 Requirements on Developer

Developers must provide a corpus of tests that adequately covers the language semantics, and also manually write the desugared ground-truth versions of the tests. This is potentially as tedious as writing the desugaring itself, and not a viable long-term improvement.

4.3.4 Shortcomings

Unfortunately, of course, this algorithm is potentially absurdly expensive: if the core tree is larger than the source (as it most likely will be), there are many possible alignments for each node, and therefore exponentially many possible alignments in total.

Additionally, this algorithm cannot learn the intended desugaring for our full language. The `SLam` and `SApp` expressions are variable-arity, which means our desugaring would require an infinite corpus to learn all the desugarings for every possible arity – when clearly the desugaring is uniformly defined. Further, the desugaring of `SFor` expressions examines the grand-children of the `SFor` node, and moreover does not preserve the ordering of children. This is simply beyond the expressive power of this algorithm.

4.4 Second Attempt: Learning a Tree Transducer by Gibbs Sampling

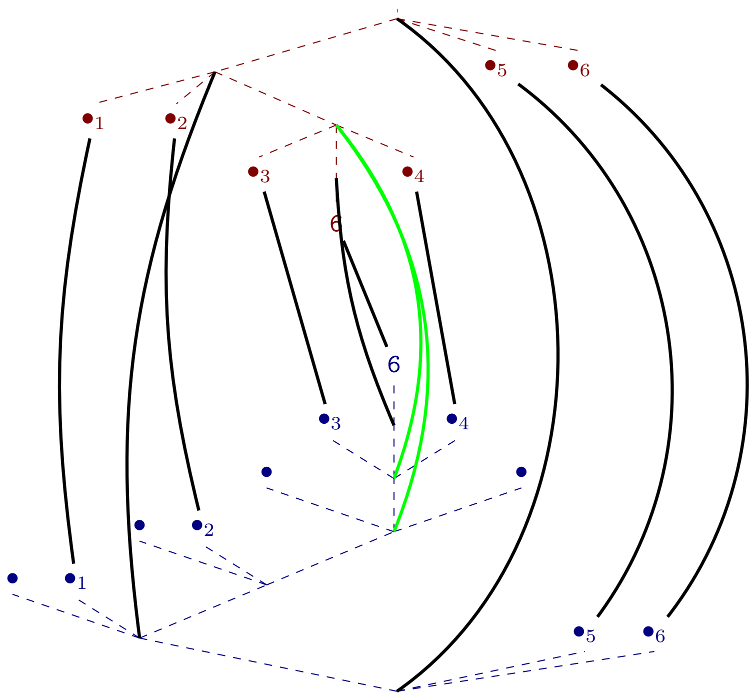
4.4.1 Essential Ideas

To constrain the search-space explosion of the simple algorithm above, we try a sampling approach instead: we relax our requirement that a desugarer be deterministic, and start with *one* alignment, which induces some desugarer. We then perturb that desugarer by picking some path in the `source tree`, from root to leaf, and attempting to "re-explain" it, hoping to find an explanation that is more deterministic than the current one. We then iterate this process until it converges on a deterministic desugarer that can explain the full test corpus. Specifically:

1. As before, we start with a corpus of `source` and ground-truth `core` programs.
2. For each test, we compute an inclusion from `source` to `core`. Combining all these inclusions produces a non-deterministic desugarer, where each translation rule is weighted by how often it was used in the current explanations of the tests. Each translation from `source` to `core` yields a *derivation tree*, whose nodes are labeled by desugarer rules and whose shape is the same as the `source` tree. Moreover, these weights induce a probability that describes how likely this derivation tree was to have occurred.
3. Repeat until convergence:
   a. Pick a derivation tree of lowest probability. Pick a path within that tree of lowest probability.
   b. "Truncate" the corresponding `source` and `core` trees by removing all side branches from that derivation path, and all `core` subtrees corresponding to the removed `source` nodes. This produces two narrow trees.
   c. Compute all possible inclusions of the truncated `source` into the truncated `core` tree. The roots must obviously correspond, as must the sole remaining leaf and its counterpart; the only nodes whose alignments could change are therefore the nodes along the `source` path. This greatly limits the combinatorial explosion to a manageable level.
   d. Compute the probability of each of these new alignments, using weights that have been adjusted to remove the existing alignment's contribution, and choose a new alignment based on these probabilities. This is essentially assigning a Dirichlet prior to the rules, and it will tend to reuse explanations that have been used many times in other tests, and tend to avoid explanations that have not been useful in many other cases.
   e. Update the weights of all the relevant rules.

4.4.2 Worked Example

Consider again the example expression in Section 4.3.2, and suppose by step 2 we had computed the mistaken desugaring shown there. In step 3.a, we would pick a path of lowest probability: in this example, we choose the path leading to the leaf `6`. We then truncate both the `source` and `core` trees and prune away all branches not relevant to that path. (We give the pruned nodes distinct labels, so we don't have to worry about mis-aligning them.)



Now the two possible alignments are very easy to compute; in fact, they are the only two possible alignments of these two trees (compared to potentially exponentially many alignments in the original, non-truncated trees). We then compute the probability of either alignment occurring given the frequencies with which the relevant desugaring rules have appeared in other test cases. The correct alignment is seen more frequently in other tests, leading to a higher probability here, which in turn reinforces the algorithm's confidence in the correctness of that alignment.

4.4.3 Requirements on Developer

Just like the naïve algorithm, this one requires the developer to supply a sufficient test corpus and its ground-truth accompaniment. However, it does not take quite so long to run.

4.4.4 Shortcomings

There are four key failings with this approach. First, just as with the naïve approach, this algorithm makes the key assumption that `source` trees will map to `core` trees of a similar shape, and `SFor` breaks that assumption.

Second, as above, the algorithm cannot handle arbitrary-arity nodes such as `SLam`. (We could shoehorn languages into fitting a fixed-arity model since no single program will ever have unbounded-arity source expressions, but even so it is subjectively wrong to provide completely disparate desugaring rules for binary functions from ternary ones, etc. Moreover, while any individual program is of fixed arity, the desugaring must handle all possible programs: the correctness of a single uniform rule is supported by all the programs of all the arities that it translates correctly, while a set of arity-specific rules necessarily are each supported by much smaller, independent sets of sample programs.)

Third, the algorithm cannot handle unbounded state during the desugaring process. In particular, it is quite common to need to generate new names (i.e., gensym) for use in the translated program. Nothing in the tree-transducer approach – regardless of how the transducer is constructed – can support such a construction.

Fourth, translation might sometimes need to inspect "deeply" into the `source` tree, i.e., examine more than just the root node and its immediate children, in order to translate it correctly (as, for example, with `SFor`). An *extended* tree transducer can handle this case, but the inference problem for extended tree transducers is known to be hard, and the algorithms above have no representation for such deep rules.

4.5 Third attempt: Genetic Programming

4.5.1 Essential Ideas

We aim to fix the first, second, and fourth problems above, while still remaining tractably expensive. Our main idea is to develop a domain specific language for describing translations. Our desugarers are now explicit programs in this language, rather than implicit in the mappings between two sets of trees. Because they are now explicit, we can mutate them in various ways, and this leads us to a genetic algorithm for evolving a correct translator. Our key observation is that rather than relying on extensive programmer effort to supply the ground truth, we can instead use the evaluators to compute ground truth.

4.5.2 Requirements on Developer

Unlike the algorithms above, we no longer require the programmer to supply a comprehensive corpus of `source` or `core` programs. Instead we can systematically generate `source` programs, or generate new `source` programs from a small number of exemplars via fuzzing. Additionally, the developer need *never* manually desugar a complete program.

Instead, our algorithm may ask the developer:

1. To say whether a given desugaring rule is at fault, or else provide a smaller `source test program` that still translates incorrectly, and
2. To provide the correct desugaring of a particularly troublesome `source language` construct, or suggested mutations to try when searching for such desugarings.

This latter point is *precisely where the programmer's attention should be focused* – on constructs whose semantics are not readily captured by "natural" tree transformations.

4.5.3 Shortcomings

Unfortunately, we were unable to make this scale well enough to find our desugaring. In general, because it is a randomized algorithm that depends heavily on the history of a particular run, convergence is troublesome, and like all genetic algorithms depends strongly on the fitness function we use. Since correctness is a boolean property with no gradient against which to optimize, our attempts at "smoothing" this into a useful fitness function amounted to counting what fraction of programs were translated correctly. However at the small scales we could attempt, this function was still too discrete to work well.

4.6 Fourth attempt: Synthesis

The first two approaches tried to exploit the tree-structure of the source and destination languages. The third approach tried to incorporate the fact that desugaring should be *a computable function*, and therefore a program in its own right. Perhaps techniques from program synthesis might help generate this desugaring? We know of only one attempt in this direction so far [14]. We highlight both the achievements and simplifications of this work, to emphasize what makes this problem quite so challenging.

The starting observation is that, because desugaring should not be expressive, and hence essentially be a tree homomorphism, the skeleton of any such desugarer is a pattern match across the source-language AST, followed by recursive desugaring calls to the subtrees, and finally some core-language constructor(s) to connect the resulting pieces. To make the problem tractable, the authors deeply embed the correctness criterion from Figure 1 as the specification of a correct synthesis result. To derive a desugarer such that the results of evaluation match up, when the authors encounter a subexpression of the outer term they leave it marked as a symbolic input, so that when attempting to interpret the resulting translated program, they replace the unknown output with the interpreted original source program. This "inductive decomposition" provides an order-of-magnitude improvement in performance.

However, the approach there takes some crucial simplifying steps. They rely on a shared value space for the interpreted answers for both languages; a more accurate representation of Figure 1 would account for the translation of values across the languages, which defeats the inductive decomposition property. Additionally, even in the sample language in Figure 3, the desugaring of `SFor` is not a trivial tree homomorphism: the translation needs to reach more deeply into the tree to rearrange binders and bound values. This form of manipulation is commonplace for languages, and should be part of a full solution here. Finally, the authors note that incorporating stateful semantics into their interpreters exploded the runtime cost of their approach, yet state is a necessary component of nearly all language designs.

5 Conclusion

Programming language research is caught between two extremes. On the one hand, we aspire to have pure, clean-slate designs that are designed well, accompanied by rich specifications, and implemented faithfully. On the other hand we have the messy world of contemporary systems, which have evolved haphazardly and democratically, and are encumbered with awkward edge cases and legacy constraints.

While we applaud those who work to make the former world a reality (and hope to make our own modest contributions to this effort), we want to call attention to the latter, which is at least our present reality. In this world, errors and attacks abound. The rich toolkit of programming languages can do much to defend against these problems, but only if we can bring these systems within our ambit. A semantics seems a necessary and important step in that direction. We believe the research program outlined here, with an emphasis on core languages and on a great deal of accidental complexity swept away by desugaring, is a smart way to proceed.

Earlier, we referred to (informal) semantics being defined in terms of "implementations, documents, and test suites". Our work shows how we can leverage implementations and test suites to reverse-engineer the formal semantics. It is natural to wonder what documents can contribute to this process. Do they make things worse, or do they actually make them better? Might we be able to curtail the combinatorial searches demanded by this paper by using information in prose? We do not know the answer, but hopefully others far more qualified can answer that authoritatively.

References

1 Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 87–100, New York, NY, USA, 2014. ACM Press. doi:10.1145/2535838.2535876.

2 Martin Bodin, Tomás Diaz, and Éric Tanter. A Trustworthy Mechanized Formalization of R. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 13–24, New York, NY, USA, 2018. ACM. doi:10.1145/3276945.3276946.

3 Denis Bogdanas and Grigore Roşu. K-Java: A Complete Semantics of Java. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 445–456, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676982.

4 Arthur Charguéraud, Alan Schmitt, and Thomas Wood. JSExplain: A double debugger for JavaScript. In *The Web Conference 2018*, pages 1–9, Lyon, France, April 2018. doi:10.1145/3184558.3185969.

5 William Clinger and Jonathan Rees. The Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

6 Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. http://tata.gforge.inria.fr/, 2007.

7 Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2015. ACM Press.

8 Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 533–544, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103719.

9 Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, 17:35–75, 1991.

10 Daniele Filaretti and Sergio Maffeis. An Executable Formal Semantics of PHP. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 567–592, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-44202-9_23.

11 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

12 Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 336–345, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737979.

13 Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, July 2018. doi:10.1109/CSF.2018.00022.

14 Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 247–263, 2017.

15 Ali Kheradmand and Grigore Roşu. P4K: A formal semantics of P4 and applications. Technical report, University of Illinois at Urbana-Champaign, April 2018. arXiv:1804.01468.

16 Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

17 Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 346–356, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737991.

18 Joe Gibbs Politz, Matt Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Dynamic Languages Symposium*, 2012.

19 Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The Full Monty: A tested semantics for the Python programming language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2013.

20 Omar Zaidan and Chris Callison-Burch. Crowdsourcing Translation: Professional Quality from Non-Professionals. In *Association for Computer Linguistics*, pages 1220–1229, 2011.