$n + 1$ Challenges for Software Language Engineering*

Friedrich Steimann

Fernuniversität in Hagen, Germany
steimann@acm.org

Abstract. This is a free transcription of the author's keynote held at the Workshop on Original and Open Problems of Software Language Engineering.

Keywords: trees vs. graphs · executable metalanguages · editing · empirical evidence vs. game theory

Prologue

Software Language Engineering (SLE) is the study of techniques and tools for SLE. It follows from the recursiveness in this definition of SLE that for any challenge of SLE identified, there will be a next challenge, namely to devise a technique or tool for addressing it.

That would be an SLEish explanation of the title of my keynote. However, the true reason for the ominous $n + 1$ is that at the time of its announcement, I was not sure how many challenges I wanted to present; I only knew that I wanted to present one concluding, down-to-earth challenge that everyone could relate to.

The list of challenges I ended up with has length $5 + 1$, and is a personal one. I am sure that others see other challenges, and also that some may not even regard my challenges as such. However, if the presented challenges cause frowning, I am happy, since frowning usually marks the beginning of progress.

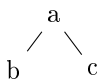
Without further ado, here are my challenges, nicely enumerated and terminated by one banal bonus challenge.

1 Forget about Trees, or: Embrace Relations

Text-based language is inherently linear. This holds for spoken as well as for written language. Graphs are not generally linear, but can be linearized; in fact, they must be linearized when speaking or writing about them (using language, that is). The linearization of non-linear content requires encodings that may add

complexity to interpretation, which is needed to reconstruct the information lost by linearization.

Not all types of graphs suffer equally from linearization. For instance, trees like



are conveniently encoded as terms, which blend seamlessly with linear text, as in

"terms like $a(b,c)$ blend seamlessly with text".

While the same holds for set-notation based lineaer representations of graphs like

$$\{(a,b),(a,c)\} \tag{1}$$

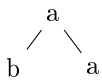
their interpretation requires the build-up of a (mental) environment, or lookup table, here to note that a occurs twice, meaning that the graph has two edges starting from the same node. The term representation, which does not need lookup for its interpretation, is therefore at advantage here.

Term representation is however challenged by graphs that are not trees. For instance,

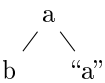
$$\{(a,b),(a,a)\} \tag{2}$$

which is but a slight variation of (1), cannot be directly encoded as a term: whereas (2) delinearizes as

its naive rendering as a term $a(b,a)$, if at all considered legal, delinearizes as



in which the two occurrences of a represent different nodes, meaning that $a(b,a)$ represents a different graph than (2). To encode the same graph as a term, an explicit lookup must be introduced, as in $a(b,$ "a"): while syntactically, this still represents a tree



the quoting can be used to construct the graph

from it. However, now the term representation has lost its above-noted advantage, namely its independence from lookup.

Even if it requires lookup, using terms for the representation of graphs still has the advantage of overlaying the graphs with trees, which requires special markup for set-based representations of graphs like (1) and (2) (see the final challenge for an example). This overlaid tree not only fixes linearization (for the set-based linearization, the order of edges is arbitrary), it also defines a containment relation that can be used, for instance, for restricting lookup. However, most realistic languages have more than one such relation, so that markup will be needed anyway, and lookup cannot generally rely on containment alone (see, e.g., [1] for some standard complications). So, why should we use terms to represent graphs?

When looking at our metalanguages, and also at the metatheories that come with them, we find that terms are *the* predominant data structure: terms encode the syntactic structure of the subjects of the metalanguages, and also drive inductive definitions and proofs. And yet, the general insufficiency of terms becomes all too evident in the omnipresence of environments, or lookup tables (often encoded in Greek, with the usual connotation that it is getting complicated), which are required to account for the inherent non-treeness of the subject matter.

So, are trees a necessary tribute to the use of metalanguages and their theories? Rather not. As has been pointed out by Martin Erwig [2], the same inductive principles that work for trees also work for graphs. And yet, the use of trees predominates. This gives us

The Graph Paradox: *In dealing with graphs, we resort to trees.*

It is a paradox because we express the general in terms of the special, which should be impossible. That it is possible nevertheless is because actually, we resort to trees plus environments, and to lookup for extracting edges on the fly.

Leaving this academic kink aside, the preference of trees over graphs appears to come with a preference of functional metalanguages over relational ones. However, just like trees are more constrained than graphs, functional languages are more constrained than relational languages. While being more constrained can amount to being more powerful, more power usually comes with less expressiveness, and indeed, functions are uni-directional, and dealing with partiality (partial undefinedness) and mapping to more than one value require clumsy workarounds. It is somewhat revealing how the functional community takes pride in their `Maybe` and `List` monads, which effectively make functional languages more relational, yet only at the price of adding indirection through a container, which adds considerable complexity [7].

Language shapes thought. Maybe the linearity of languages leads to a preference of trees; and with it to a preference of functions, over graphs and relations. Maybe we would fare better if we used graphs and relations directly, rather than encode them as trees and functions. We will not know until we try.

2 Tend To Your Metalanguage, or: Use One that Works

In his acclaimed keynote "It's Time for a New Old Language" [5], Guy Steele identified 28 different notations for substitution in POPL papers published between 1973 and 2016. While this diversity would not be such a problem if every paper properly introduced its used notation, there is usually no space for such introductions. That the papers are accepted nevertheless suggests that there is sufficient redundancy in each paper to let the reader infer with certainty what gets substituted by what, which in turn suggests that the presentation could have been terser still — by removing that redundancy — had the community agreed on a standardized notation.

Terseness combined with implicit standardization, or conventions, can be a hindrance to the accessibility of papers. For instance, and as also noted by Steele [5], to represent "unenclosed sequences", the notations e^*, $e_1 \ldots e_n$, and $\overline{e}$ are in common use, with origins that are not easy to trace and with meaning that is not self-evident. For instance, the star in e^*, where e is an expression, is not the Kleene star (even though it is sometimes so called); the Kleene star constructs the set of all concatenations of elements from a base set, but neither e nor e^* stand for sets — they stand for elements of such sets. The overbar (or overscore, or overline) notation of $\overline{e}$ is reminiscent of the mathematical notation of vectors ($\vec{e}$), and indeed, the e_i that $\overline{e}$ represents are assumed to be ordered, yet, $\overline{e}$ is not one object like $\vec{e}$ would be, but instead stands for many objects of the same kind, with no additional meaning imposed (there is no "whole" such as a vector). Also, as syntax specifications, both the e^* and the $\overline{e}$ notation have problems with inserting separator tokens, which leads to some ad hoc conventions such as writing

$$C(\overline{C}\,\overline{f})\ \{\mathrm{super}(\overline{f});\mathrm{this}.\overline{f} = \overline{f};\}$$

(from the specification of Featherweight Java [4, §19]), which is only loosely related to the concrete syntax (expressed in a variant of EBNF with terminals underlined)

$$\underline{C}\,\underline{(}\,[\,C\,f\,\{\,\underline{,}\,C\,f\,\}\,]\,\underline{)}\ \underline{\{\mathrm{super}(}\,[\,f\{\underline{,}f\}\,]\,\underline{);}\,\underline{\{\mathrm{this}.}f\,\underline{=}\,f\underline{;}\}\}}$$

which in turn must be expanded to

$$C(C_1\,f_1,\ldots,C_n\,f_n)\ \{\mathrm{super}(f_1,\ldots,f_n);\mathrm{this}.f_1 = f_1;\ldots\mathrm{this}.f_n = f_n;\}$$

or even, in the most general case, to

$$C(C_1\,f_1,\ldots,C_n\,f_n)\ \{\mathrm{super}(f'_1,\ldots,f'_{n'});\mathrm{this}.f''_1 = f'''_1;\ldots\mathrm{this}.f''_{n''} = f'''_{n''};\}$$

when it comes to specifying semantics. As if this were not occult enough, different papers use different notations, so that accessing any one of these papers is only possible if one already knows which content to expect. Wouldn't it be great to have a single metalanguage whose mastering is sufficient to access the contents of all papers?

Of course, the community has such a metalanguage: Latex. Unfortunately, Latex is a highly underconstrained language that is not at all related to, let alone domain-specific for, solving the problems of SLE.[1] What would be needed instead is a unified language for SLE, or a

Unified SLE Standard (USLES).

And yet, experience with another unified language standard tells us that usefulness may be compromised, rather than increased, by unification. Why, then, not use an existing, versatile, proven language as metalanguage? A language that works not only on paper, but that can actually do something? Like ML, Prolog, or Coq?

In fact, although not generally a requirement, it would be good to have executable metalanguages, since only executable languages can actually do some work for us. This of course would make all our specifications programs, which is great, since programs we know how to test and debug. And, after all, isn't the passion for programming much wider spread than the passion for specification?

3 The Metalevel is Not the World of Static, or: Operationalize Editing

Like object languages, metalanguages are usually constrained by static semantics. The metalanguages in which static semantics are expressed additionally need dynamic, or operational, semantics, for otherwise, the constraints expressed cannot be evaluated. However, these constraint languages are usually not programming languages in the sense that they are unable to construct new objects, or lack a notion of state. Programming languages, on the other hand, can be, and routinely are, used to express both static and dynamic semantics of metalanguages, even though most programming languages are not particularly suited for this purpose. That's why SLE is so rich in DSLs.

For many programming languages, statically type-safe ones in particular, static and dynamic semantics work hand in hand so as to guarantee certain runtime properties of programs. These properties usually include notions of well-formedness of run-time structures ("preservation" [4]); for instance, in a type-safe language, all variables are guaranteed to hold only values conforming to their declared type. This means that all operations creating new objects, or causing state changes, are guaranteed to respect the language's rules of well-typedness, so that no runtime-cheking is required. Other constraints on the static structure (such as those for ensuring object ownership or non-nullness) make other runtime guarantees.

[1] Note that it is not helpful to think of Latex as a meta-metalanguage for SLE, since in SLE, every metalanguage is a language on equal footing, so all problems apply recursively. Specifically, if Latex is not suited for specifying languages, it is not useful for specifying metalanguages. That said, it would however help to have widely shared Latex packages that standardize notation, for instance by providing universally accepted macros for substitutions and unenclosed sequences.

Why not apply this situation to the level of metalanguages? After all, when we run an editor as a program, its operations are the edit operations, and its state is the current shape of the object (program, model) being edited. And yet, even though a semantic editor embodies the static semantics of the object language (i.e., the language of the object being edited), its edit operations usually make only weak guarantees with regard to preserving well-formedness of the object (the runtime structure manipulated by the editor as a program), and leave most of the constraints of the static semantics of the object language to runtime checking. This is so because many well-formedness constraints of the object languages are non-local, and because single primitive edit operations, even in a semantic editor (in which an AST is manipulated directly and hence in a type-safe manner), are generally insufficient to preserve global well-formedness. Yet, this is not a lost case.

For instance, *sound refactoring tools* compute sequences of primitive edit operations required to make the intended change *and* to keep the program well-formed [8]. By dropping the condition of behavior preservation, the techniques of refactoring can be generalized to well-formedness preserving editing [6, 9]. Soundness proofs of such complex edit operations are possible [8], but are generally difficult for imperative implementations of editors. Also, in practice, complementing intended primitive edits with computed well-formedness preserving ones requires user interaction, thus calling for a notion of *edit transactions*.

Edit transactions will also be needed in the context of collaboration, or parallel editing of the same object [10]. Editors thus becomes database programs: a set of primitive update operations is performed in isolation and can only be committed if the so-edited object is well-formed. As above, the task is complicated by the fact that well-formedness is not a local property in the general case, meaning that the locks required for guaranteeing isolation of edit transactions may severely impede parallelism; yet, database technology is rich in methods of dealing with these problems, and it is not at all clear that these methods do not work better for us than our current, optimistic version control systems, especially when considering the commit procrastination caused by the fear of manual merging (edit conflict resolution).

Speaking of version control: what do programming languages have to do with files? Nothing![2] Files are the subjects of programming tools, that's all. There is no theoretical obstacle to storing source programs in databases, and to use more sophisticated representations than plain text for this — on the contrary, there's a lot to be gained from doing so. SLE will know that it has delivered when programmers no longer ask for *copy&paste* of arbitrary (i.e., not structurally bounded) ranges of text.

[2] Of course, some languages are defined as depending of files, like C for instance. Java on the other hand replaces files with the notion of compilation units, which are storage technology neutral. Yet, compilation units are chunks of text, which can be edited just like (text) files.

4 Diversify, or: Gain Relevance

A compiler that compiles itself, a metalanguage that defines itself — SLE is the metacircular discipline of metacircularity. While others[3] consider eating their own dog food an academic exercise, the SLE community *lives on it*.

While such a practice may be considered educational, and certainly is both witty and economic (what an undefeatable combination!), I conjecture that it generates generations of scholars for whom SLE tools are the only programs they have ever seen the inners of. As a corollary to this conjecture, I assume that the same scholars prefer to model everything as an immutable tree. Life can be wonderful if spent inside an ivory tower.

The real world is very different, however. In the society that covers my salary, an estimated 80% of all software created is embedded, and security, real-time, and resource efficiency are as important as safety guarantees. For the remaining 20% of software created, distribution, persistence, and integrity are just as vital.

Almost none of this is the subject of SLE. Yet, SLE is dearly needed in these contexts, as the demands from software, and with it the software, grow rapidly beyond what can be managed without SLE techniques and tools. But would companies whose revenues come from other products than geeks' tools hire software language engineers? If they did, what would they get? From inside an ivory tower, it is easy to judge what is going on out in the wild, yet almost impossible to change it for the better. Only if software language engineers know enough about security, real-time, and resource efficiency, as well as distribution, persistence, and integrity, they will have a real impact. And real impact can be more rewarding than agreeing inside an ivory tower on how the outside world should be.

5 Escape the Empiricism Trap, or: Formalize Utility

Relevance requires utility. After too many papers making too grandiose claims supported by too little, the scientific community agreed that if formal proofs cannot be delivered, empirical evidence will be needed instead. Hard empirical evidence is however hard to obtain in a field in which experiments have strong cognitive aspects and confounding factors are hard to control. Specifically, the performance of humans making use of the latest SLE product depends on experience and education, and memory makes every experiment a confounding factor for the next. It is therefore questionable whether SLE experiments with humans provide more reliable answers than clear judgment common sense reasoning.

So, what can we do to escape the empiricism trap? In economics, very prestigious prizes have been repeatedly awarded to the development of mathematical models of human behavior in decision making problems, aka *game theory*. Programming *is* a decision making problem; more specifically, and when taking tool support into account, it is a game played by the programmer and the IDE, in

[3] It seems that as a tool, metacircularity works only for informatics; in other disciplines, it is either impossible to construct, or considered bogus.

which each take turns: the programmer strikes a key, the IDE responds with immediate feedback such as proposing completions or marking or unmarking errors. In fact, programming with an IDE is a *cooperative game*, one in which the programmer has the idea of what the program should do, while the IDE knows how the program may evolve. The programmer works towards the program becoming useful, and the IDE works towards best use of existing code and error-freeness. Any new aid in this process that is put forward in a scientific context should show how it improves this game, by some self-selected metric. Metrics can be: number of decisions to be made by the programmer until a program is correct; number of choices offered by the IDE at each decision point; information content of the made proposals, etc. If analytical captures of the complexity ("big O") of the decision problems are too hard to derive (which, given the complexity of the nature of programming, would not be surprising), simulation and Monte Carlo techniques may still provide acceptable evidence. Some first work in this direction (albeit restricted to software processes, not tool interaction) has only just appeared [3]; looking into other disciplines for more inspiration might prove worthwhile.

$5+1$ How to Type that Damn Parent Attribute?

Abstract syntax trees are typed and attributed graphs in which each node is an instance of a type (or class) representing a syntactic category, and in which attributes hold either values or references to other nodes. Attributes are also typed; reference-typed attributes provide for the edges of the graph.

Some of the attributes defined for each node type are designated as *child* attributes; they define the spanning trees that let us regard abstract syntax graphs as trees (cf. Challenge 1). This tree structure is of special significance even at the abstract syntax level, namely when it defines the containment relation of a programming language, and with it its lexical scopes. Because of this special role, the containment relation almost always needs to be navigated in both directions: from the owner of the attribute to the child and from the child to the owner, aka *parent*. The parent direction of the relation is usually also represented by an attribute (which may be derived, though) and, therefore, is also typed. The question is, what is its type?

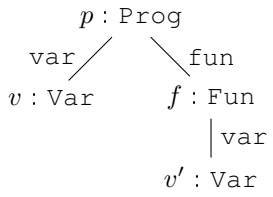
Naively, it has the type of the owner of the corresponding child attribute, but this does not work if different parent node types share the same child node type. This however is routinely the case; for instance, for programs comprised of (global) variables and functions, where functions may have local variables, we may find the node type declarations

```
node type Prog =
  child var : Var;
  child fun : Fun
end;
```

```
node type Var =
  parent() : Prog
end;
```

```
node type Fun =
  child var : Var;
  parent() : Prog
end;
```

(where `parent()` means that the parent is derived). Now given the nodes



the expression v'.`parent()`.`parent()`, which would get one from v' to f to p, is ill-typed, because the type of v'.`parent()` is `Prog`, and `Prog` does not have a `parent()` attribute.

While this problem is a very standard one, it seems that it is solved by every implementation of abstract syntax trees anew: some use union types (here: `Prog#Fun` for the parent attribute of type `Var`) or least common supertypes for parent attributes, and resort to type tests and downcasts for accessing attributes specific to an actual parent node (here: to access `parent()` on v'.`parent()`), while others may use more sophisticated type systems or bypass static typing entirely. Why is that so?

Epilogue

We have:

$$\frac{\textit{the world depends on software} \qquad \textit{software depends on software languages}}{\textit{the world depends on software languages}}$$

We get:

SLE will never run out of challenges

References

1. van Antwerpen, H., Neron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A constraint language for static semantic analysis based on scope graphs. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 49–60. ACM (2016). https://doi.org/10.1145/2847538.2847543, https://doi.org/10.1145/2847538.2847543
2. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. **11**(5), 467–492 (2001). https://doi.org/10.1017/S0956796801004075, https://doi.org/10.1017/S0956796801004075
3. Gavidia-Calderon, C., Sarro, F., Harman, M., Barr, E.T.: Game-theoretic analysis of development practices: Challenges and opportunities. Journal of Systems and Software **159**, 110424 (2020). https://doi.org/https://doi.org/10.1016/j.jss.2019.110424, http://www.sciencedirect.com/science/article/pii/S0164121219301980
4. Pierce, B.C.: Types and programming languages. MIT Press (2002)
5. Steele Jr., G.L.: It's time for a new old language. In: Sarkar, V., Rauchwerger, L. (eds.) Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017. p. 1. ACM (2017), https://www.youtube.com/watch?v=7HKbjYqqPPQ
6. Steimann, F.: Befactoring: preserving non-functional properties under behavioural change. In: Murphy-Hill, E.R., Schäfer, M. (eds.) Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013. pp. 21–24. ACM (2013). https://doi.org/10.1145/2541348.2541354, https://doi.org/10.1145/2541348.2541354
7. Steimann, F.: None, one, many — what's the difference, anyhow? In: Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA. LIPIcs, vol. 32, pp. 294–308. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.SNAPL.2015.294, https://doi.org/10.4230/LIPIcs.SNAPL.2015.294
8. Steimann, F.: Constraint-based refactoring. ACM Trans. Program. Lang. Syst. **40**(1), 2:1–2:40 (2018). https://doi.org/10.1145/3156016, https://doi.org/10.1145/3156016
9. Steimann, F., Frenkel, M., Voelter, M.: Robust projectional editing. In: Combemale, B., Mernik, M., Rumpe, B. (eds.) Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017. pp. 79–90. ACM (2017). https://doi.org/10.1145/3136014.3136034, https://doi.org/10.1145/3136014.3136034
10. Steimann, F., Kurowsky, N.: Transactional editing: giving ACID to programmers. In: Nierstrasz, O., Gray, J., d. S. Oliveira, B.C. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019. pp. 202–215. ACM (2019). https://doi.org/10.1145/3357766.3359536, https://doi.org/10.1145/3357766.3359536