Clef Design

Thoughts on the Formalization of Program Construction

Klaas van Schelven
klaas@vanschelven.com

ABSTRACT

In *Expressions of Change* modifications to programs replace text files as the primary building blocks of software development. This novel approach yields structured historic information at arbitrary levels of program granularity across the programming toolchain. In this paper the associated questions of Programming Language Design are explored. We do so in the context of s-expressions, creating a modification-based infrastructure for languages in the Lisp family. We provide a framework for evaluation of the relative utility of different formalizations of program construction, which consists of the following: first, a requirement for completeness, meaning that a formalization of program construction should allow for the transformation of any valid program into any other. Second, a preference for succinctness over verbosity; succinctness of both of the formalization itself and typical expressions in the formalization. Third, a measure of the ability to clearly express intent. Fourth, a description of three ways in which the means of combination of the program itself and those of its means of construction may interact. Finally, we give a particular example of a formalization and use the provided framework to establish its utility.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; *Software configuration management and version control systems*; *Software maintenance tools*;

KEYWORDS

program modification, programming language design, s-expressions

ACM Reference Format:
Klaas van Schelven. 2018. Clef Design: Thoughts on the Formalization of Program Construction. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

A large part of software development is concerned with modifying existing software, often over long time spans[6]. The associated need for historic record keeping is reflected in the popularity of *Version Control Systems*. In the mainstream approach, however, such systems are retrofitted rather than integrated: text remains the primary building block of program construction and the main interface shared across the tools in the development toolchain, such as editors and interpreters; the history is managed separately by the *VCS*.

An alternative approach is to take the modifications themselves as the primary building blocks. The set of allowed modifications to program structure is formalized, and such modifications are taken as the inputs and outputs by all tools in the programming toolchain. In short: program modification is reified. Experiments with this approach are bundled in a project called *Expressions of Change*.

We expect that the availability of well-structured historic information across the toolchain will prove invaluable when facing the typical challenges of program modification. As part of the project we have developed a prototype of an editor and a programming language, and early experiments with these indicate that the expected advantages will indeed materialize.

Setting out to reify program modification, one is immediately faced with the rather obvious question: *what does reified program modification look like?* What is the language with which a programmer can express, to the computer and other programmers alike, how a program can be constructed or modified? What are the *expressions of change*?

In more general terms, the question is: given some formalization of program structure, what should the accompanying formalization for program construction be? This is a non-trivial question, as the number of possible such formalizations is infinite, even for a single formalization of program structure, and any practical experiment with reified program modification must choose only a single one. This choice is thus a matter of programming language design.

This paper explores that design space for the single formalization of program structure of s-expressions. S-expressions make for a good initial testbed for this exploration for a number of reasons: the simplicity of their definition, the typical explicitness of their structure when pretty-printed, and the fact that the choice for s-expressions ensures some immediate relevance of any findings for languages in the Lisp family.

The relevance of this exploration for the project *Expressions of Change*, as well as projects which take a similar approach, is self-evident. As far as we know, questions of design of program modification have not been previously described, which is not surprising as they are directly tied to the original approach of reification of program modification itself. The contributions of this paper are the following:

- We develop a framework of criteria for comparison of different formalizations of program modification (section 3) and provide an overview of practical considerations in the design of such formalizations (section 4).
- We present a minimal formalization of program modification (section 5) and show that the chosen formalization ranks reasonably well given the criteria (sections 6 and 7).

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

2 THE DESIGN SPACE: S-EXPRESSIONS

In this paper we explore the question of design of formalizations of program construction in the context of a single formalization of program structure, namely that of s-expressions. An s-expression is recursively defined here as either:

- an atom, which is a symbol or number[1]
- a list of 0 or more s-expressions[2]

S-expressions are typically represented textually by printing the atoms as-is, and the lists between regular parentheses `(` and `)` with white space separating the elements of the list. Thus, the following is the textual representation of an s-expression:

```
(+ (* 6 9) 12)
```

2.1 Why s-expressions?

The choice for s-expressions as an object of study is motivated by a number of reasons.

First, their absolute minimalism: s-expressions can be both fully defined and illustrated with an example in some ten lines of text. Such a minimalistic definition of program structure allows for maximum focus on the subject at hand: that of program construction. Further, a minimal formalization of program structure is a prerequisite for a minimal formalization of program construction, because each special case of the program structure needs to be somehow accounted for. Smaller formalizations of program construction are preferable over larger ones in general (see section 3.2). For a first exploration of the design space, which this paper represents, this is even more strongly the case.

Second, the mapping between the structure of s-expressions and their visual representation is very direct. The more explicitly the structure is laid out on screen, the easier it is to understand modifications in terms of that structure. When modifications are put central, that is an important property.

Finally, the choice for s-expressions is practical, because s-expressions (or some extension thereof) form the basic syntax of many languages in the Lisp family, making it possible to use artifacts produced by *Expressions of Change* in the practical environment of an actual programming language. It also ensures relevance of the findings of the project for any existing languages in that family.

2.2 Intuitions for formalizations

Having formalized s-expressions, and thus program structure, we are ready to discuss formalizations of program construction. We start by developing an intuition, using a practical example of an expression of program construction in plain English:

Given the example expression given in section 2.1, do the following 3 things sequentially:

- remove the atom `+` from the root expression.
- insert a new atom, `hello-world`, as the first child of the root.
- remove the atom `6` from the sub-expression `(* 6 9)`.

The above list of bullets, although it is useful to provide an intuition for languages of program construction, is not sufficiently formal for our purposes. In particular, one of the explicit goals for such a language is that it can serve as an input for automated processes, i.e. as a shared interface across the development toolchain.

What are the elements that we might expect in any such formalization? Again, let's develop the intuition first. In general, we can at least expect:

- Support for a number of different kinds of modification, e.g. adding, updating, removing, copying and moving structures.
- Each kind of modification will be associated with further information that is specific to it, i.e. for deletions it is sufficient to specify a location only; for insertions we must also specify what must be inserted.
- Specific kinds of structure may be tied to specific mechanisms of construction. In the case of s-expressions: the ways we can modify atoms are not the same as the ways we can modify lists.
- A formalization of the order (if any) in which the operations must take place

Please note that the first two bullet points together form a good fit with Algebraic Data Types[7]: the different kinds can be represented using *Sum Types*, the different attributes using *Product Types*. We will occasionally use this fact as a notational shorthand below, independent of actual concerns of implementation.

Having established what we can expect in a formalization of program construction, let us develop an intuition for the associated design space, in which we need to make decisions such as:

- What kinds of modification should be supported? For example, is "updating" a special kind of operation, or is it enough to have access to a combination of adding and removing items?
- What are the relevant attributes for each kind of modification? For example, when deleting an item, what is the best way to formalize what is to be deleted? Should multiple such formalizations be catered for simultaneously?

Finally, let us establish the fact that this design space is infinite, by noticing that neither the space of kinds of modification, nor the set of relevant attributes has an upper bound. For example, to any formalization of program construction we can always add a kind of modification that inserts a particular s-expression. Since the number of s-expressions is infinite, we can thus create an infinite number of special kinds of modification.

2.3 Terminology and notation

Having established these intuitions, we shall introduce a minimal amount of terminology and notational convention. This is necessary because, in contrast to formalizations of program structure, in which terms such as *grammar*, *term* and *parser* have been well established, formalizations of program construction have not been well studied, and hence such terminology is not yet available. In choosing this terminology, we have taken inspiration from musical notation, being a real-world example of instructions for 'construction' rather than 'structure'. We introduce the following terms:

[1] There is no generally accepted single definition of what constitutes an s-expression. Instead, definitions vary, with support for a variety of possible atoms such as text, symbols, integers and floating point numbers. We restrict ourselves to printable symbols and numbers here without loss of generality.

[2] Lists are typically implemented using (nested) pairs; however, in this paper we shall make no assumptions about their implementation.

- A *clef* denotes a particular formalization of program construction. The analog in program structure is a grammar: just like a grammar describes what valid program syntax is, a clef describes what the valid means of construction are. The analog in natural language is a vocabulary: the clef defines the valid words. The musical metaphor is be understood as follows: just like a musical clef provides semantics for the notes in a score, a clef in program construction provides the semantics for its notes.
- A *note* denotes a specific kind of operation of construction, such as "adding", "removing" or "copying". Borrowing from the terminology of implementation: if a clef is implemented as an Algebraic Data Type, a note corresponds to a single data constructor. The respective analogs are: a term in a grammar, a word in a vocabulary. The term *note* is overloaded to also mean an instance of a *note*, with given values for all attributes, e.g. "delete the 2nd child".
- *To play* a note is to apply it to an existing structure, yielding a new structure.
- A *score* denotes a list of notes. To play a score is to play each note in turn, leading to the step-wise construction of some structure.

In this paper, we shall use s-expressions as a means of notation for notes. In particular, each note will be denoted using a list-expression, where the first element denotes its kind (corresponding to a data constructor in an ADT) and further elements denote the values of the attributes. The notation for a score is a list-expression of notes. Thus, the following denotes a score of 2 notes:

```
(
    (delete 3)
    (delete 5)
)
```

Please note that such a choice of notation poses no restrictions on implementation whatsoever, and is not mandated in any way by the fact that the structure under modification is itself an s-expression. It does, however, come with the benefit of directly suggesting a means of implementation in Lisps. An advantage of such an implementation is further that it enables self-applicability, i.e. the modification of notes in our clef in terms of that same clef.

The semantics of a clef are given in terms of a case-analysis on its notes. We use the imperative style, that is, a formulation in terms of how a given s-expression must be modified to reflect the playing of a particular note. If needed, an equivalent definition in the functional programming paradigm can be trivially derived.

3 A FRAMEWORK OF CRITERIA

In the previous section we have provided an overview of the infinitely many possible forms a clef may take. We shall now turn our attention to a comparison between these forms. The goal is to be able to pick a single clef which is shared as a common interface by tools in the programming toolchain[3].

[3]In fact, there being only a *single* clef is not a hard requirement, and future versions of the project may very well support a small set of somewhat related clefs, each one being used in a different part of the toolchain. For the utility of having a framework of evaluation of clefs this makes no difference: such a framework may then be used to select which clefs this small set consists of.

Because different clefs are not equally useful in meaningfully expressing program modification, choosing a particular one is a matter of design. It stands to reason that we need some mechanism of evaluation of the utility of clefs, such that we may compare the utility of different approaches and choose the best one. In the below we present one such mechanism, i.e. a number of criteria that may be used for evaluation of clefs, as well as arguments pertaining to why these particular criteria are useful.

In the construction of this framework, we have in some cases taken inspiration from the evaluation of the relative utility of programming language features more generally, that is, outside the scope of program modification. Where this is the case, we make sure to highlight the aspects that are specific to program construction rather than structure.

3.1 Completeness

The first criterion for a successful clef is that it is *complete*. Given an arbitrary present structure it should be possible to reach an arbitrary desired structure in a finite number of steps. We relax this requirement somewhat in the context of structures that are defined in terms of sum types, stating that it is sufficient to be able to reach any structure defined using the same data constructor. That is, for s-expressions, it is enough to be able to construct any list-expression out of any list-expression, and any atom out of any atom. We assume the utility of this property to be self-evident.

3.2 Clef size

Second, with regards to the size of the clef's definition we make the observation that, all other things being equal, smaller is better.

First, the size of the clef is reflected in the cost of implementation of the automated processes programs that use it as its interface (editors, tools for program analysis, compilers). The larger this language, the larger the implementation-cost across the toolchain.

Second, larger clefs impose larger costs on their human users. In the approach of this project the notes from the clef form the primary building block of program construction. It follows that explicit exposure of the clef to the end-user, the programmer, is a design goal. If we want the programmer to be able to meaningfully interact with elements of the clef, they must understand these elements. The larger the clef's definition, the larger the mental burden of understanding it poses on the programmer.

Finally, a larger clef increases the risk of a distinction without a difference: that multiple equivalent mechanisms exist to construct the same result, even in cases when there is no meaningful underlying reason for this. Such distinctions serve only to confuse, and must be avoided.

The preference for small definition size is entirely analogous with the same preference in programming language design proper. However, it's worth noting that, with respect to text based programming languages, the clef introduces an additional layer of complexity on tools and programmers alike. Thus, the pressure to keep it small is increased.

3.3 Typical expressions' size

A clef that allows for concise expression of typical modifications to programs is to be preferred over one that does not. Please note that

this is a separate concern from the one in the previous subsection, analogously to the cases of both programming language design and natural language, in which the size of the vocabulary is distinct from the size of sentences formed with that vocabulary, and in which the two are often inversely related.

In terms of automated tooling, larger expressions will typically incur some cost on storage and performance. However, this cost is expected to generally negligible and the size of typical expressions has no implication on implementation cost, which is tied strictly to definition size. The greatest cost of large expressions is thus incurred on the programmer, who will spend more time constructing, reading and understanding such expressions.

One important thing to note in the above criterion is that it assumes some knowledge of what "typical modifications to programs" are. That is, to a large degree, an empirical question.

3.4 Preservation of intent

A fourth desirable quality in a clef is for it to allow for clear expression of programmer intent. Here, again we take inspiration from the design of computer programs and programming languages, in which clear communication of intent is a desirable quality[2, 4, 9]. In that context, the concept of *programmer intent* is more or less understood: it denotes what the programmer wants a particular part of the program to achieve, and the mechanisms for them to communicate this intent with others. But what do we mean when talking about programmer intent in the context of program construction?

We mean approximately the following: when a programmer modifies the program, they typically do not do so at random, but with the intent to achieve a particular desired result. More often than not, this is achieved in a number of steps — whereby each step has some meaning to the programmer. Such a step-wise approach might even be reflected in a *todo* file or a piece of paper on which the steps are crossed off one by one. Similarly, in a pair programming session, one programmer may explain to another what they need to do next in a number of steps. The more closely the expressions built using a particular clef resemble lines in a *todo* file or utterances in a pair programming session, the better they express original intent.

Why do we think this is important? A central hypothesis of *Expressions of Change* is that, by having access to ubiquitous well-structured historical information, programs can be more easily understood. Such understanding is much easier to achieve if those histories are expressed in ways that are close to the thought-process of the original programmer.

As an example, consider the programmer goal of moving the method `get_widget()` from class `Foo` to class `Bar`. A clef that allows for expression of this goal using a single-note called "move" reveals more about the original intent than one that expresses the same modification using two separate and unrelated actions called "insert" and "delete".

A final piece of evidence for the importance of expression of intent in the context of program modification is presented by the currently accepted best practices in Version Control Systems, which are in favor of expressing intent through both mechanisms. In the language of Version Control: "One commit, one change" and "Writing good commit messages".

We make a distinction here between two different mechanisms through which intent may be expressed. First, there is the greater or lesser ability to express intent directly in terms of the notes of the clef, as in the example above. Second, some clefs may allow for expression of intent through informal comments in natural language.

3.5 Means of combination

A final desirable quality in a clef is that it allows for meaningful means of combination. Here, again, we take inspiration from the evaluation of expressiveness of programming languages per se, which may be approached using the question "What are the means of combination?"[1, 10, p. 4]

The importance of this question is a direct consequence of human nature. Human beings, including programmers, can typically hold approximately 7 items in their mind in an active, readily available state at any given time[11]. Thus large problems are approached by dividing them into parts, and combining those parts, and programs are no exception. Examples of such means of division and combination are *modules*, *procedures*, *expressions*, *classes* and *methods*. It is in terms of such parts, and the way that they are combined, that programs are understood.

The central hypothesis of *Expressions of Change* is that programs may also be understood in their historical context. How do these two mechanisms of creating understanding interact? How does the concept of meaningful composition interact with clef-design? We distinguish 3 questions, for 3 different kinds of interaction.

First, what are the means of combination within the clef itself? Can its notes be meaningfully combined at all? Moreover, can the results be used as the elements in further combinations? That is, does this mechanism of combination form a closure, and thus the ability to form arbitrarily structured hierarchical histories?

The second question we can ask of a clef is the following: does its usage enable a meaningful relationship between the program composition and its history? As noted, programs are typically composed of modules, classes, procedures etc. Is the historical information available for each such part? If so, this enables the programmer to get a historic view at any level of the program hierarchy, such that they may get an answer to each of the questions "what's the history of my program?", "what's the history of this module?" and "what's the history of this expression?" equally.

Third, is there a meaningful relationship between the program's composition and the composition of the program's history? Do the histories of parts of the program relate to the histories of their sub-parts? If they do, a programmer may arbitrarily switch between 2 modes while navigating: that of program structure (space) and program history (time).

4 PRACTICALITIES OF DESIGN

Before introducing the clef proper, we present some further considerations of clef design. These cannot not included in the framework from the previous section, because the trade-offs associated with any particular approach do not obviously point towards a particular design. Nevertheless, they do represent design decisions and are therefore relevant to discuss.

4.1 Sum type structure

An s-expression, as defined in section 2, is precisely one of two things: an atom or a list, i.e. it is defined as a sum-type. The fact that the way we can modify each of these two things is different has immediate consequences for an approach to structured modification. For lists, for example, we can reasonably speak about the insertion of an element, but for an atom such a modification is meaningless because an atom has no elements[4].

For a clef of s-expressions, the practical result is that some of its notes will be playable exclusively on list-expressions, and others exclusively on atoms[5]. To play such notes on a structure of the wrong kind, e.g. to add a child to an atom, is not allowed by definition[6].

4.2 Initial notes

As noted in section 2.3, notes are typically defined in terms of modification of an existing s-expression. When a note is played as the first note in the score, this raises the question what the existing s-expression to modify would be. We present three possible answers:

- We choose a particular single structure, such as the empty list-expression `()`, as the initial structure as a matter of definition.
- We specify the initial structure explicitly as needed, i.e. when playing a score; we keep track of what the initial structure is in some location external to the score, i.e. as "metadata"
- We alter the definition of the semantics of notes slightly, such that special semantics may be assigned in case they are played as the initial note in a score. Further, we assign such special semantics to one or more notes in the clef. That is, the initial structure is defined to be some special sentinel value denoting nothingness, and one or more notes are defined to be construct a particular s-expression out of such nothingness.

All of these approaches have some drawbacks: the first elevates one particular kind of structure over the others by making it the initial one, even if no natural order exists. This lack of natural order applies to s-expressions: it isn't quite clear whether we should consider list-expressions or atoms to be the most natural initial s-expression. In the second approach, the task of identifying the initial structure is pushed out of the score, but we must still keep track of it somehow. In practice, this means we need to associate this information with all children-creating notes, i.e. push the information "one level up". In some sense this moves the problem elsewhere rather than solving it. The third approach introduces a degree of asymmetry in the clef: some notes, but not all, may be used as the initial note.

For clefs of s-expressions in particular, the drawbacks of the third approach seem to be most limited, hence we have chosen it. In the chosen approach we introduce one or more special notes for structure-creation, which leads to a final question of clef-design: should playing such notes as anything but the initial note be an error, or should it have the effect of transforming whatever previous structure there was into the initial structure of the designated kind? We have chosen to disallow such midway reinitializations, judging that there cannot be a meaningful historical connection between an atom and a list-expression. Again, this decision may or may not generalize to other types of structures than s-expressions.

[4]We take the indivisibility of atoms to be their defining property by definition (from Greek – the prefix "a" meaning *not* and the word "tomos" *to cut*). The fact that an atom may be represented as a string of textual characters is, in this view, an implementation detail that is encapsulated.

[5]This conclusion does not generalize to clefs for any structure which happens to be defined as a sum-type: if various kinds of structure are similar in the way they can be modified a single note may be applicable to more than a single kind of structure.

[6]This is not to say that any note for list-expressions is playable on any list-expression, as the set of playable notes may be constrained by properties of the list-expression its played on. For example: removal of the 3rd element of a list is only possible if the list has such an element.

5 A MINIMAL CLEF

In this section, we present a minimal clef for s-expressions. As noted in section 2.3, we use the imperative style for a description of the semantics. For each item, any expectations about the previous s-expression are noted first. Any non-defined behavior such as playing a note on the wrong kind of s-expression, playing an initial note non-initially and vice versa is considered disallowed by definition.

- `(become-atom` ⟨*atom*⟩`)` — initial note only — constructs the given atom out of nothing.
- `(set-atom` ⟨*atom*⟩`)` — playable on atoms — modifies the atom into the given atom.
- `(become-list)` — initial note only — constructs an empty list expression out of nothing.
- `(insert` ⟨*index*⟩ ⟨*score*⟩`)` — playable on list-expressions — constructs a new s-expression by playing all notes in the given score in order, and inserts this newly constructed s-expression as a new child element at the provided index[7].
- `(delete` ⟨*index*⟩`)` — playable on list-expressions — deletes the element at the index.
- `(extend` ⟨*index*⟩ ⟨*score*⟩`)` — playable on list-expressions — constructs a new s-expression by playing the score, using the child currently at the provided index as an initial s-expression, and replacing the child with the result.
- `(chord` ⟨*score*⟩`)` — playability depending on the first note of the provided score — plays the provided score sequentially.

5.1 Example usage

Consider the following score:

```
(
    (become-list)
    (insert 0 (
        (become-list)
        (insert 0 ((become-atom *)))
        (insert 1 ((become-atom 6)))
    ))
    (insert 0 ((become-atom +)))
    (insert 2 ((become-atom 12)))
    (extend 1 (
        (insert 2 ((become-atom 9)))
    ))
)
```

The stepwise construction of an s-expression according to this score is summarized in the table below.

[7]Indices are, of course, 0-based[3].

lines	note	result
2	`(become-list)`	`()`
3 - 7	`(insert 0 (... * ... 6 ...))`	`((* 6))`
8	`(insert 0 ((become-atom +)))`	`(+ (* 6))`
9	`(insert 2 ((become-atom 12)))`	`(+ (* 6) 12)`
10 - 12	`(extend 1 (... 9 ...))`	`(+ (* 6 9) 12)`

5.2 Chords

Chords may be used to hierarchically structure a score, as in the example below. Please note that grouping notes by means of chords does not alter the semantics of construction; as such, the below example may be flattened into a single score without altering what s-expression would be constructed as a result.

```
(
    (become-list)
    (chord (
        (insert 0 ((become-atom +)))
        (insert 1 ((become-atom 3)))
        (chord (
            (insert 2 ((become-atom 8)))
            (insert 3 ((become-atom 4)))
        ))
    ))
)
```

6 EVALUATION OF UTILITY

In this section we examine the provided clef in terms of the first four means of evaluation of the framework; that is in terms of completeness (3.1), clef size (3.2), succinctness of expression (3.3) and means of expression of intent (3.4).

We first note that the clef is complete, i.e. it may be used to construct arbitrary list-expressions from arbitrary list-expressions (although, because reinitializations have been disallowed as per section 4.2, it is not the case that arbitrary atoms can be created from arbitrary list-expressions and vice versa). A trivial, albeit inefficient, mechanism to do so is: first, construct the empty list-expression from the given list-expression by deleting the first element until no more child-elements exist. Then, from the empty list-expression, construct the desired list-expression by, for each child, creating the score that constructs it recursively, and inserting it. Arbitrary atoms can be created trivially by the clef's definition.

Regarding the size of the presented clef, we note that at present no alternative exists, which is to say that a quantitative comparison is hard to make. We thus content ourselves with the observations of some basic facts: the clef has a total of seven notes. Two of these are specific to atoms, four to list-expressions and one is a generic means of combination. With respect to the number of types of expressions (atoms and lists), this represents a factor 3.5. In any case, seven is by no means the minimal amount, which is one[8]. We must point out though, that such a reduction of clef-size comes at considerable cost in terms of the other criteria of evaluation.

The following two criteria, namely whether the clef can be used for succinct expression of typical program modification, and whether it allows for clear expression of programmer intent are discussed together below. As noted, those criteria imply further questions which can only be answered empirically: what is the nature of typical program modification, and what is the kind of intent that a programmer typically wants to reveal? To answer these questions, we have implemented, as part of the project "*Expressions of Change*", a prototype of an editor that implements the given clef. Informal experiments indicate that the presented clef scores quite well.

In addition to this empirical observation with an actual prototype, we make some observations about the general nature of editing. The most basic ways to interact with any kind of data, are to add, update and delete[9]. The presented clef provides direct support for all of these; that is: an edit-session comprising of such actions alone can be expressed succinctly, and without loss of intent. However, these are by no means the only operations available in typical (text) editors. We mention a few:

- moving pieces of text around
- copy-pasting
- search and replace
- advanced code refactoring

For such operations, no direct counterparts are available in the clef. However, annotated chords may be used to preserve, to some degree, expression of intent. For example, a *move* might be expressed as a single chord that deletes a sub-expression in one place, and inserts the score representing the moved sub-expression's method of construction elsewhere.

7 MEANS OF COMBINATION

Finally, we turn our attention to the means of combination. In section 3.5, we distinguished 3 questions about the interaction between means of combination of structure and construction. Here we shall evaluate the presented clef in terms of those 3 questions.

7.1 Combining notes with chords

First, what are the means of combination within the clef itself? The most straightforward combination is to take a linear sequence of changes. Such a sequence is part of our definition; we've called this a *score*.

A more profound means of combination is provided by the note `chord`: it combines multiple notes, in the form of a score, into a single note. Thus, it can be used to form arbitrarily structured hierarchical histories.

The practical use of this ability is to structure a stream of changes into time-wise "chapters" and "sub chapters" for the human reader, i.e. to express intent. For example, the introduction of a new feature may require the refactoring of a certain part of the program, which is in turn an operation that consists of a number of further restructurings. Chords allow for a programmer to express precisely such a hierarchical structuring of history.

[8] The clef with the single note become, which takes an s-expression and changes the entire s-expression under consideration into the given s-expression is an example of such a 1-note clef.

[9] Reflected, for example, in the acronym CRUD[8]. (Please note that the *R* in that acronym, for *Read*, is not a data-altering operation and has therefore no relevance to a clef).

7.2 Combining structure and construction

Postponing the discussion of the second question for a moment, we turn our attention to the third: is there a meaningful relationship between the program's composition and the composition of the program's history?

Please note that the means of combination for the program structure under consideration, s-expressions, are list-expressions: list-expressions combine s-expressions into new s-expressions. As an instance of this, consider the s-expression constructed in section 5.1. It is a list-expression consisting of the further s-expressions `+`, `(* 6 9)` and `12`.

As we have seen in the previous section, the most basic mechanism for composition of program construction is the score. In the presented clef scores show up as an attribute to the 3 notes `insert`, `extend` and `chord`.

What, then, is the relationship between the scores of list-expressions, and the scores of the s-expressions that they are composed of? In terms of the provided example: what is the relationship between the score for the construction of `(+ (* 6 9) 12)` and the respective scores for the construction of its parts? In particular: can the listing in section 5.1, which corresponds to the score of the expression as a whole, be used to reconstruct the scores for that expression's sub-expressions, such as `(* 6 9)`?

Indeed it can be. The key observation is that all mechanisms that affect sub-expressions, namely `insert` and `extend`, are expressed in terms of a `score` that describes modifications to the sub-expression. In general, the process for extracting a lower-level score is to extract the relevant scores from the higher level expressions and concatenate them.

For the atoms `+` and `12`, the result is somewhat trivial: their histories consist solely of those atoms coming into being. The history of the list-expression `(* 6 9)` is more interesting; it can be constructed by concatenating the scores as found on lines 4 – 6 and 11, resulting in the following score[10]:

```
(
    (become-list)
    (insert 0 ((become-atom *)))
    (insert 1 ((become-atom 6)))
    (insert 2 ((become-atom 9)))
)
```

This is precisely what we were looking for: a meaningful relationship between the program's composition and the composition of the program's history.

Incidentally, this relationship also implies a positive answer to the second question: it enables the programmer to get a historic view at any level of the program hierarchy. Thus, our rather minimal clef has meaningful composition on all 3 levels.

8 FUTURE WORK

The central hypothesis of *Expressions of Change* is that structured historic information at arbitrary levels of program granularity is extremely useful in the software development practice. As part of the project we have built a prototype of an editor and a small interpreter of a subset of *Scheme*. Experiments with this editor in an informal setting confirm the practical applicability of the findings in this paper. However, to substantiate the central claim, much work remains to be done. In this work, we may distinguish between production and consumption of the clef. That is, first further experimentation with the editor to ensure a fluent editing experience, and second, experimentation with the utilization of this structured information in the rest of the toolchain. An example of the latter is to approach various forms of static analysis from the perspective of program construction, that is: incrementally.

In this paper we have explored formalization of program construction for a single particular formalization of program structure, namely that of s-expressions. Whether it is possible to define clefs of practical utility for arbitrarily complex formalizations of program structure is an open question, although there are some reasons to believe that it isn't. In particular, the arguments in favor of s-expressions as an object of study, presented in section 2.1, do not generally extend to arbitrary definitions of program structure. Our conclusion is that future languages should be designed with structured modification in mind.

In the prototype of the editor, user actions correspond directly to notes in the clef, and its output is a score representing the actual edit-session. However, programming is to some degree an exploratory activity and an actual edit-session may contain many dead ends. A log including all those dead ends is not the clearest possible way to communicate intent. Thus, there is likely a need for an extra step, that is analogous with a *commit* in a VCS, in which some of reality's details are hidden in the interest of clarity. We imagine that various automated tools will be usefull in this step, e.g. to automatically detect such dead ends and prune them as required.

The properties of formalizations of program construction in the context of a collaborative programming effort have not yet been researched. Briefly, we can say the following: one key question when collaborating is how the diverging work of multiple programmers can be joined together with some degree of automation (that is: "merging"). With respect to the mainstream approach, Version Control of text files, our approach creates both advantages and challenges. On the one hand, the availability of fine-grained well-structured information makes automated merges easier, and error messages more precise. On the other hand, the introduction of a historical dimension raises new questions, because merging needs to take place on the level of program construction as well as program structure.

9 RELATED WORK

The approach to program construction presented in this paper presupposes a structured approach to program editing. Examples of recent projects that take this approach are *Lamdu* and *Unison*, both with a program syntax inspired by Haskell and a strong focus on static typing, and *Cirru*, which is a structured editor of s-expressions. Program *construction*, however, has not been given a central role in those projects. To our knowledge, the only other project which has an explicitly defined semantics of program construction is

[10] The fact that the notes (insert 0 ...) and (extend 1 ...) are indeed modifying the same sub-expression might not be immediately apparent because the indices differ. This shift in indices is caused by an intermediate insertion at index 0 on line line 8. In the UI of practical applications, the connection between such apparently unrelated notes may be clarified by using some unique and unchanging identifier, such as the order of creation of a child.

Hazelnut[13], a project that focusses on the interaction between program construction and static typing.

Mechanisms of recording program history are wide-spread in mainstream software development, most notably in the form of Version Control Systems. However, such systems typically operate on unstructured text. A more structured approach to diffing is taken by Miraldo et. al.[12]. Our approach differs because it puts program modification central in the design, rather than trying to extract information about program modification after the fact.

Finally, and most generally, some ideas analog to those presented in this paper are captured in the design pattern of *Event Sourcing*, described by Martin Fowler[5]. In that pattern, the idea is to capture all changes to an application state as a sequence of events. That is, the pattern captures the idea of construction-over-structure in the domain of Enterprise Application Architecture rather than programming.

10 CONCLUSIONS

Following from the observation that computer programs will be changed, and that managing those changes themselves forms a large part of a computer programmer's work, *Expressions of Change* presents a novel approach: to put the modifications themselves central in the programming experience.

The results of the first step in that approach have been presented in this paper: how to approach the design of the formalization of programming construction? We have shown that important goals in such design are: to enable intent-revealing primitives and succinct expression, while keeping a minimal footprint. Further, we have shown various possible levels of combination, both within the formalization of program construction and in its interaction with program structure.

Finally, we have shown a particular formalization of program construction for s-expressions, and how this minimal mechanism allows for expression of programmer intent, and all 3 levels of possible means of combination.

ACKNOWLEDGMENTS

We would like to thank Wouter Swierstra and the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper.

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, Upper Saddle River, NJ, 1997.

[3] Edsger W. Dijkstra. Why numbering should start at zero. circulated privately, 8 1982. URL http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[5] Martin Fowler. Event sourcing, capture all changes to an application state as a sequence of events, 2005. URL https://martinfowler.com/eaaDev/EventSourcing.html.

[6] Robert L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Softw.*, 18(3):112–111, 2001. ISSN 0740-7459. doi: 10.1109/MS.2001.922739. URL http://dx.doi.org/10.1109/MS.2001.922739.

[7] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238856. URL http://doi.acm.org/10.1145/1238844.1238856.

[8] James Martin. *Managing the Data Base Environment*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1983. ISBN 0135505828.

[9] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

[10] Erik Meijer. "composition is the essence of programming", 23 November 2014. URL https://twitter.com/headinthebox/status/536665449799614464.

[11] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, March 1956. URL http://www.musanim.com/miller1956/.

[12] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 2–15, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5183-6. doi: 10.1145/3122975.3122976. URL http://doi.acm.org/10.1145/3122975.3122976.

[13] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. *CoRR*, abs/1607.04180, 2016. URL http://arxiv.org/abs/1607.04180.