Interaction vs. Abstraction: Managed Copy and Paste

Jonathan Edwards
jonathanmedwards@gmail.com
Independent
Boston, USA

Tomas Petricek
tomas@tomasp.net
Charles University
Prague, CZ

Abstract

Abstraction is at the core of programming, but it has a cost. We exhort programmers to use proper abstractions like functions but they often find it easier to copy & paste instead. Copy & paste is roundly criticized because subsequent changes to copies may have to be manually reconciled, which is easily overlooked and easily mistaken. It seems there is a conflict between the generality and reusability of abstraction with the simplicity of copying and modifying code.

We suggest that this conflict arises because we are still thinking in terms of paper-based notations. Indeed the term "copy & paste" originates from the practice of physically cutting and gluing slips of paper. But an interactive programming environment can free us from the limitations of paper. We propose *managed copy & paste*, in which the programming environment records copy & paste operations, along with structural edit operations, so that it can track the differences between copies and reconcile them on command. These capabilities mitigate the aforementioned problems of copy & paste, allowing abstraction to be deferred or reduced.

Managed copy & paste resembles version control as in git, except that it works not between versions of a program but between copies within the program. It is based on a new theory of structural editing and version control that offers precise differencing based on edit history rather than the heuristic differencing of textual version control. We informally explain this theory and demonstrate a prototype implementation of a data science notebook. Lastly, we suggest further mechanisms of *gradual abstraction* that could be provided by the programming environment to lessen the cognitive load of programming.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; **Software configuration management and version control systems**; *Software maintenance tools.*

Keywords: code clones, copy & paste, structure editing, version control

ACM Reference Format:
Jonathan Edwards and Tomas Petricek. 2022. Interaction vs. Abstraction: Managed Copy and Paste. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3563836.3568723

1 Introduction

It is widely held that the essence of programming is abstraction, and that a primary way to express such abstractions is with functions. In his Turing Award lecture Dijkstra [8] said "the notion of the closed subroutine is . . . one of the greatest software inventions . . . because it caters for the implementation of one of our basic patterns of abstraction." Pierce [32] says "Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

However practitioners have found that abstractions are often uncertain at first and can change significantly as a design space becomes better understood, leading to wasteful code churn. Therefore many adopt pragmatic guidelines like the "Rule of Three" [15], stated by Grolemund and Wickham [18] as: "You should consider writing a function whenever you've copied and pasted a block of code more than twice". Note that quote is on page 269 – no functions needed to be defined earlier. In contrast, the first chapter of **SICP** [2] is *Building Abstractions with Functions*. This divergence reveals a disagreement between theory and practice about the role of abstraction.

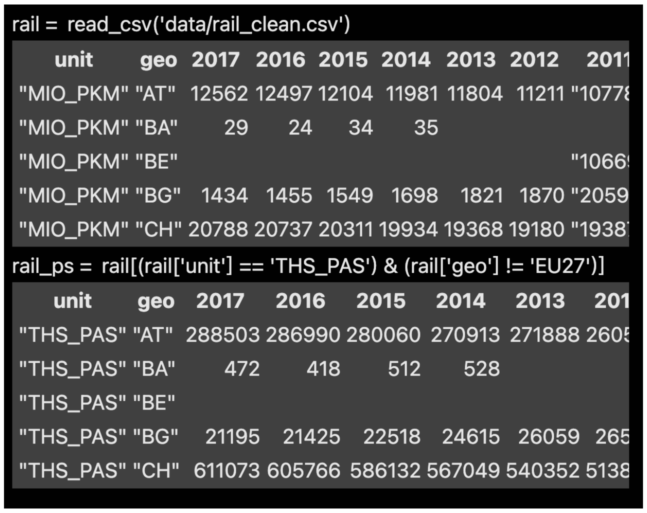
We are faced with a dilemma: premature abstraction leads to wasted effort, but the absence of abstraction makes code harder to understand and maintain. Grolemund and Wickham [18] say:

> Writing a function has three big advantages over using copy-and-paste:
> - You can give a function an evocative name that makes your code easier to understand.

- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e., updating a variable name in one place, but not in another).

We observe that the second of these three problems, needing to duplicate edits in copies, stems from the fact that plain text does not know about copying. We propose to provide a smarter editing experience with structure editing [3, 30, 34, 39]. Our editor records high-level program transformations, like inserting a function call or rebinding a variable reference or renaming a definition. The editor also tracks copy & paste operations so that it can show the differences arising between such copies, and propagate changes between them on command. The programmer can thus "update code in one place" and then propagate those changes semi-automatically. This capability offers a compromise between full abstraction and passive copying. We call it *managed copy & paste.*

Managed copy & paste depends upon our earlier work on a theory of structural version control [13, 14]. That theory showed how version control capabilities as in git could be provided in terms of structure edits, benefiting from more accurate capture of programmer intent and a simpler model of change. The technical contribution of this paper is to extend structural version control so that instead of comparing versions of a program it compares substructures within a single version of a program. Copying can be thus be treated as creating internal versions of regions of code. Our other contribution is to accordingly extend the UI for side-by-side comparison and migration of changes to manage internal copies as well. We will start with a walk through the UI in the next section and describe the underlying theory afterwards.



```
rail =  read_csv('data/rail_clean.csv')
```

unit	geo	2017	2016	2015	2014	2013	2012	2011
"MIO_PKM"	"AT"	12562	12497	12104	11981	11804	11211	"1077
"MIO_PKM"	"BA"	29	24	34	35			
"MIO_PKM"	"BE"							"1066
"MIO_PKM"	"BG"	1434	1455	1549	1698	1821	1870	"2059
"MIO_PKM"	"CH"	20788	20737	20311	19934	19368	19180	"1938

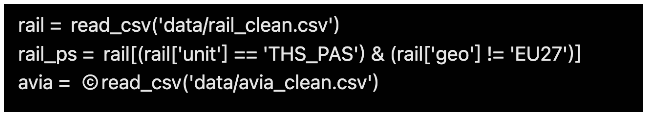
```
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')]
```

unit	geo	2017	2016	2015	2014	2013	201
"THS_PAS"	"AT"	288503	286990	280060	270913	271888	2605
"THS_PAS"	"BA"	472	418	512	528		
"THS_PAS"	"BE"						
"THS_PAS"	"BG"	21195	21425	22518	24615	26059	265
"THS_PAS"	"CH"	611073	605766	586132	567049	540352	5138

Figure 1. loading and filtering data with Python and Pandas

2 Walk through Managed Copy & Paste

In this section we walk through a usage scenario with screenshots from our prototype implementation. This scenario is based on a true story: analyzing transportation accident data in Histogram[31]. Figure 1 shows the result of loading a CSV table of railroad accident data and applying two filters to it. The UI resembles a conventional data notebook like Jupyter with *cells* of code (in a subset of Python) followed by their values (showing the first five rows of a table). The `rail_ps` cell filters the table using the Pandas library idiom of Boolean vectors created by expressions like `rail['unit'] == 'THS_PAS'`. We will call such expressions *predicates.*



```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')]
avia =  ©read_csv('data/avia_clean.csv')
```

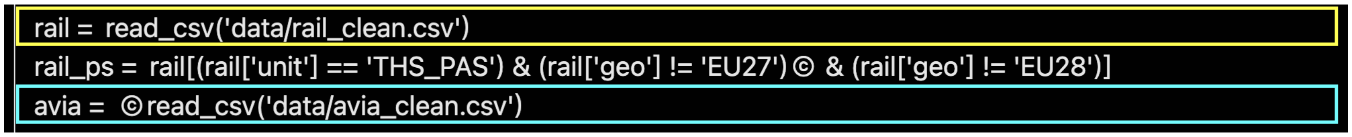
Figure 2. copying `rail` into `avia`

We want to compare railroad accidents to aviation accidents, which are in a somewhat similarly structured CSV file. We copy the `rail` cell, naming it `avia` and changing the file name, resulting in Figure 2. To save space, we have turned off cell values in this figure and subsequent ones. We also will not discuss the user interface for performing edits – our prototype uses a command line as a makeshift substitute for a proper direct-manipulation UI.

Notice the © symbol at the beginning of the `avia` cell, which indicates that it is a copy. Clicking on it selects that cell by outlining it in green, and indicates that the `rail` cell is the source of the copy by outlining it in yellow, as shown in Figure 3.

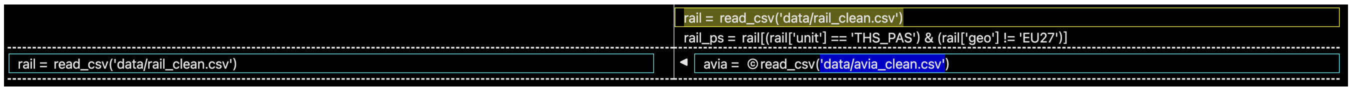
We can see the differences between the copy and its source by double clicking the ©, producing Figure 4, which shows the source of the copy aligned on the left. The filename `'data/avia_clean.csv'` is highlighted in blue on the right, indicating that it was changed in the copy. This side-by-side view is similar to that provided in many version control systems like git, except that we are comparing regions within a file rather than two versions of a file. In general both sides of a copy can show changes, with changed values in blue, insertions in green, and deletions in red. Note that this view is unlike common textual tools because it shows changes at the level of expressions, not by comparing lines of text. Thus the `rail=` on the left is not a difference from the `avia=` on the right because they are outside the expressions being copied. Likewise the filename is marked as having been entirely changed even through the leading text `'data/` is the same.

The left-pointing triangle at the beginning of the right hand side is a button that will *migrate* the changes on the right hand side to the left. When there are changes on the



```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27') © & (rail['geo'] != 'EU28')]
avia =  ©read_csv('data/avia_clean.csv')
```

Figure 3. Showing that `avia` was copied from `rail`



```
rail =  read_csv('data/rail_clean.csv')            | rail =  read_csv('data/rail_clean.csv')
                                                    | rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')]
rail =  read_csv('data/rail_clean.csv')            ◀ avia =  ©read_csv('data/avia_clean.csv')
```

Figure 4. Differences between a copy and its source

left there will be a converse right-pointing triangle. We will return to migration later.

Next we want to filter the new table somewhat like the first. As usual the easiest way to do this is to copy code and modify it. Figure 5 shows the result of making a copy of the `rail_ps` cell, naming it `avia_ps`, changing all references to `rail` to be `avia`. We also modify the first filter predicate to test a different column for a different value. All these changes to the copy are highlighted in blue on the right.

When there are multiple changes on a single line like this, it can be useful to zoom into a finger-grained view, shown in Figure 6. To produce this view we double clicked on `avia_ps`, which expanded the cell into an indented outline of *sub-cells* revealing the parse tree of the expression, breaking out each operation or grouping onto its own line.[1] Because we are displaying `rail_ps` on the left it is also expanded so that each line aligns properly with the corresponding one within `avia_ps`. For consistency the source location of `rail_ps` highlighted in yellow above is also expanded.

This expanded view spreads the changes shown in blue onto separate lines, and presents a migration triangle for each of them, so that they can be individually cherry picked. Another advantage of the expanded view is that the intermediate execution value of the subexpression on each expanded line can be shown underneath it to assist in comprehension and debugging. That is beyond the topic of this paper so we have hidden these intermediate values in the interest of simplicity.

Now copying and pasting is easy and fun, but eventually the bill comes due. Say we discover that we want to filter out rows where `geo == EU28`, and we want to do that on both tables. Normally at this point we would have to make those changes twice, but this is where managed copy & paste shows its worth. We start by adding a new predicate to the `rail_ps` cell. True to form we take the easy way by copying the prior predicate and changing `'EU27'` to be `'EU28'`. Figure 7 shows the result, with the newly copied predicate outlined in green and the source of the copy in yellow.

Note that the © symbol on the `avia_ps` cell has now turned red. This indicates that there have been changes made to the source of the copy.[2] We consider the red © to be a sort of automatic FIXME comment, flagging that the source has changed, which is the main risk of copy & paste programming. To investigate further we double click on the ©, producing Figure 8. That looks like Figure 5, except on the left side we see the `EU28` filter predicate highighted in green, indicating that it was inserted. Now we can click on the right-pointing triangle to migrate that change into the copy, the result of which is Figure 9.

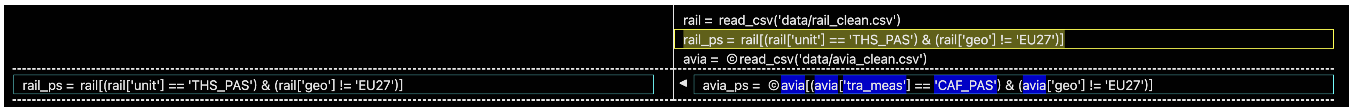
That one click replicated the `EU28` filter predicate into `avia_ps`, but we see that the © symbol in front if it is red. That is a flag that there is still some inconsistency, so we click it to investigate, producing Figure 10. That reveals the `EU28` predicate is a copy of the preceding `EU27` predicate. That is a copy of the copy in Figure 7 when we first created the `EU28` predicate. When we migrated that change from `rail_ps` to `avia_ps`, the internal copying got itself copied. Figure 11 diagrams how these copies relate. We call this *higher order copying*[12]. Now the red © is telling us there is something wrong about the copied copy, so we double click to investigate further, producing Figure 12.

We see that `EU28` was changed from `EU27`, which is expected, but we also see that `rail` on the right was changed to `avia` on the left. That's a bug. The problem is that when we migrated the `EU28` predicate from `rail_ps` it carried along a reference to the `rail` table, whereas we changed the other references in `avia_ps` from `rail` to `avia`. This is the sort of "incidental mistake" that was described as the third problem of copy & paste in the Wickham quote from the introduction. The mistake is being automatically detected as an inconsistency in the copied copy. We can fix this mistake by migrating the `avia` change, resulting finally in Figure 13.

We hope that this narrative shows the potential benefits of managed copy & paste: that by tracking copying and editing

[1]More precisely, we double clicked on the second `avia`, which is now selected inside the green box. The expression tree was expanded down to that node and its siblings. The sibling subexpressions on the following three lines could be further expanded by double clicking within them.

[2]More precisely, the red © indicates there are changes to the source that aren't masked by conflicting changes to the copy.

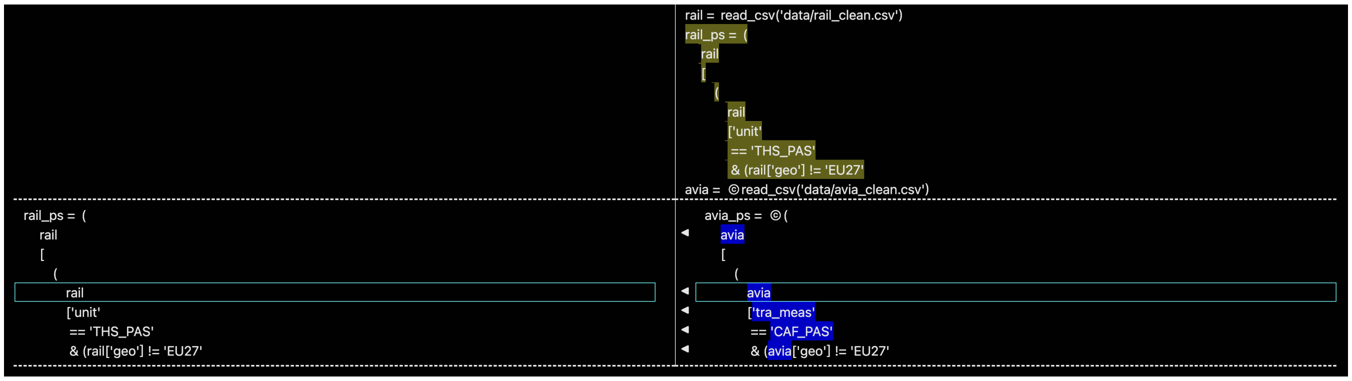


```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')]
avia =  ©read_csv('data/avia_clean.csv')
```

```
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')]
```

```
avia_ps =  ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27')]
```

Figure 5. Copying `rail_ps` into avia_ps

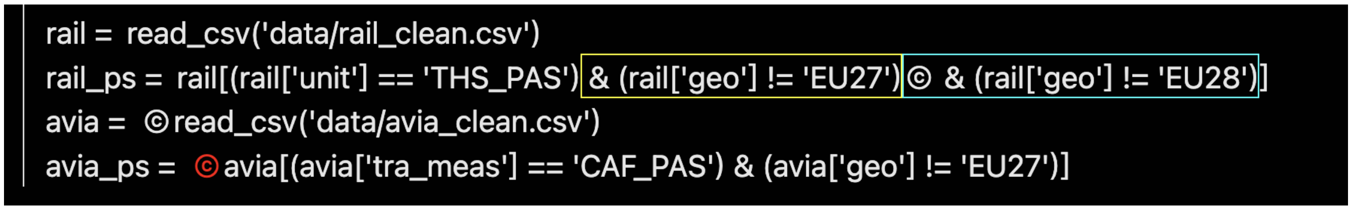


```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  (
    rail
    [
      (
        rail
        ['unit'
        == 'THS_PAS'
        & (rail['geo'] != 'EU27'
avia =  ©read_csv('data/avia_clean.csv')
```

```
rail_ps =  (
    rail
    [
      (
        rail
        ['unit'
        == 'THS_PAS'
        & (rail['geo'] != 'EU27'
```

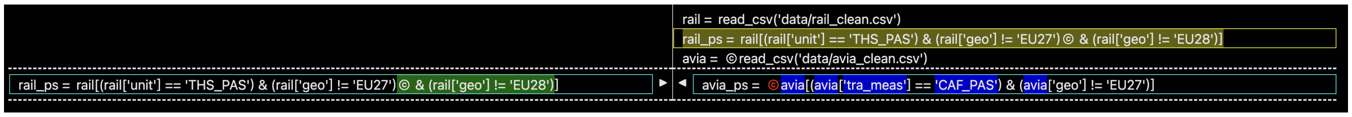
```
avia_ps =  ©(
    avia
    [
      (
        avia
        ['tra_meas'
        == 'CAF_PAS'
        & (avia['geo'] != 'EU27'
```

Figure 6. Expanded view



```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
avia =  ©read_csv('data/avia_clean.csv')
avia_ps =  ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27')]
```

Figure 7. Adding a new filter predicate by copying

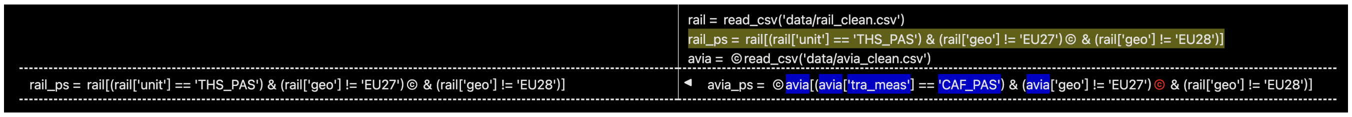


```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
avia =  ©read_csv('data/avia_clean.csv')
```

```
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
```

```
avia_ps =  ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27')]
```

Figure 8. Changes subsequently made to the source of a copy

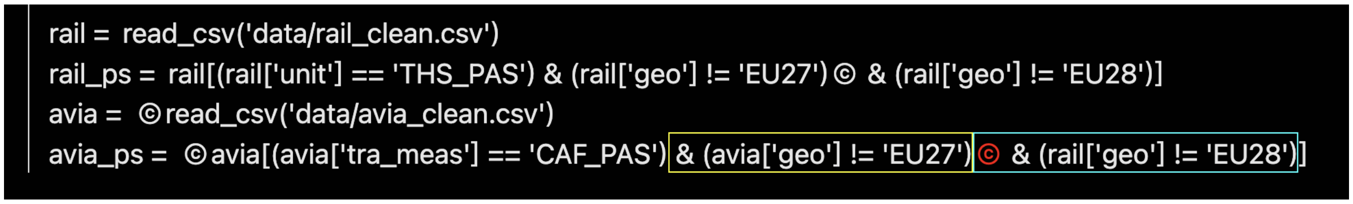


```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
avia =  ©read_csv('data/avia_clean.csv')
```

```
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
```

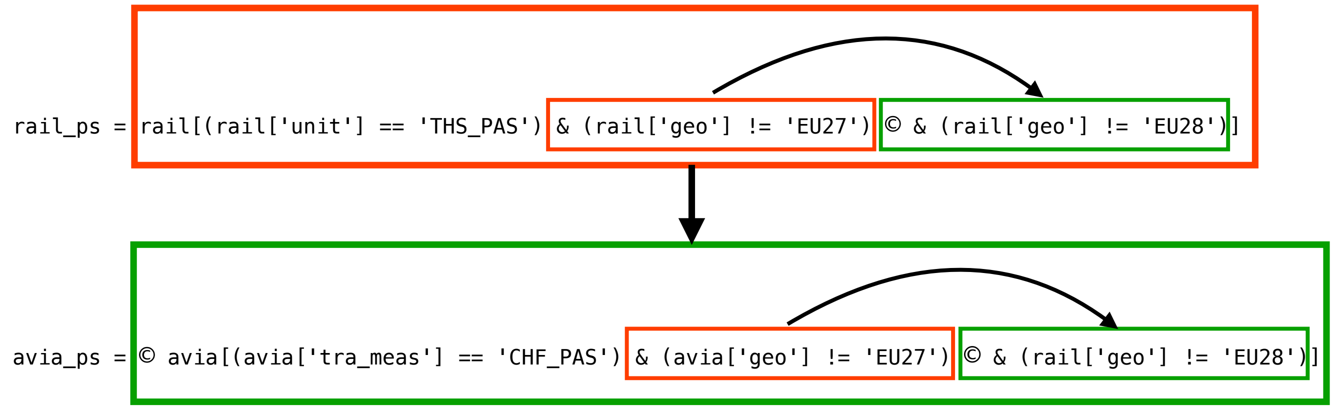
```
avia_ps =  ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
```

Figure 9. Migrating changes in the source into the copy



```
rail =  read_csv('data/rail_clean.csv')
rail_ps =  rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
avia =  ©read_csv('data/avia_clean.csv')
avia_ps =  ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27')© & (rail['geo'] != 'EU28')]
```

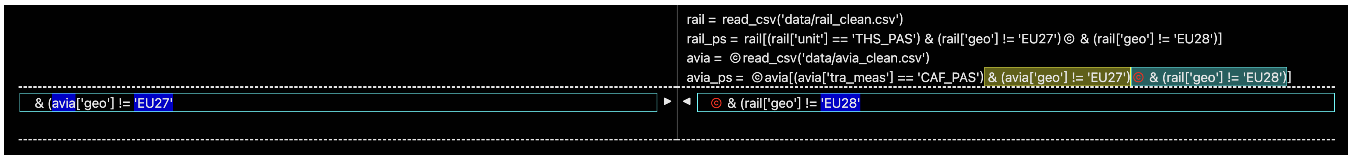
Figure 10. Copy of a copy



```
rail_ps = rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27') © & (rail['geo'] != 'EU28')]

avia_ps = © avia[(avia['tra_meas'] == 'CHF_PAS') & (avia['geo'] != 'EU27') © & (rail['geo'] != 'EU28')]
```

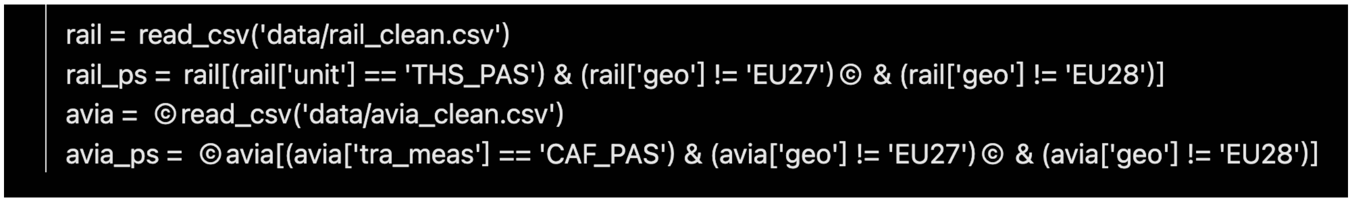
Figure 11. Higher order copying



```
rail = read_csv('data/rail_clean.csv')
rail_ps = rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27') © & (rail['geo'] != 'EU28')]
avia = ©read_csv('data/avia_clean.csv')
avia_ps = ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27') © & (rail['geo'] != 'EU28')]

& (avia['geo'] != 'EU27'    ► ◄    © & (rail['geo'] != 'EU28'
```

Figure 12. Comparing the copy of a copy to its source



```
rail = read_csv('data/rail_clean.csv')
rail_ps = rail[(rail['unit'] == 'THS_PAS') & (rail['geo'] != 'EU27') © & (rail['geo'] != 'EU28')]
avia = ©read_csv('data/avia_clean.csv')
avia_ps = ©avia[(avia['tra_meas'] == 'CAF_PAS') & (avia['geo'] != 'EU27') © & (avia['geo'] != 'EU28')]
```

Figure 13. Migration complete

operations at a high level we can detect and help correct programming errors commonly associated with copying.

3 Internal Version Control with Structure Editing

In this section we will informally explain our theory of version control with stucture editing [13, 14] and how we extend it to do internal version control between copies. In our system a program is a tree, like the AST produced by a compiler, but designed for direct structural editing by the user rather than as an intermediate representation in a compiler.

Figure 14.1 shows two programs *A* and *B* and their edit histories. These programs will typically be variants diverging after one was copied from another, but we do not assume that. Textual version control systems like git depend on finding a previously existing common ancestor of the two versions to serve as the basis for 3-way comparison, but that is only an approximation. Versions may happen to evolve in parallel, and changes may be *migrated* from one to the other (called *cherry picking* in git).

Instead of depending on finding a common ancestor we compute an optimal one. In Figure 14.2 we calculate *A*&*B*, the *maximal agreement* between the programs, generating two histories of *differences* forking from the agreement to match the two programs. The agreement serves as the basis for 3-way comparison [24]. It is maximal in the sense that all of the differences can be migrated to the other side and will actually make a difference. We will skip the details of how the agreement is calculated because it is essentially an iteration of the migration algorithm described below.

We base version control on selecting sets of edits to migrate from one side of the differences to the other. Figure 14.3 shows migrating the last difference of *A* to *B*. Since the agreeement is maximal we know this makes a new version of *B* called *B*′, which also adjusts the agreement to be *A*&*B*′. As an edit is migrated through the differences they may be adjusted along the way so that they correctly express

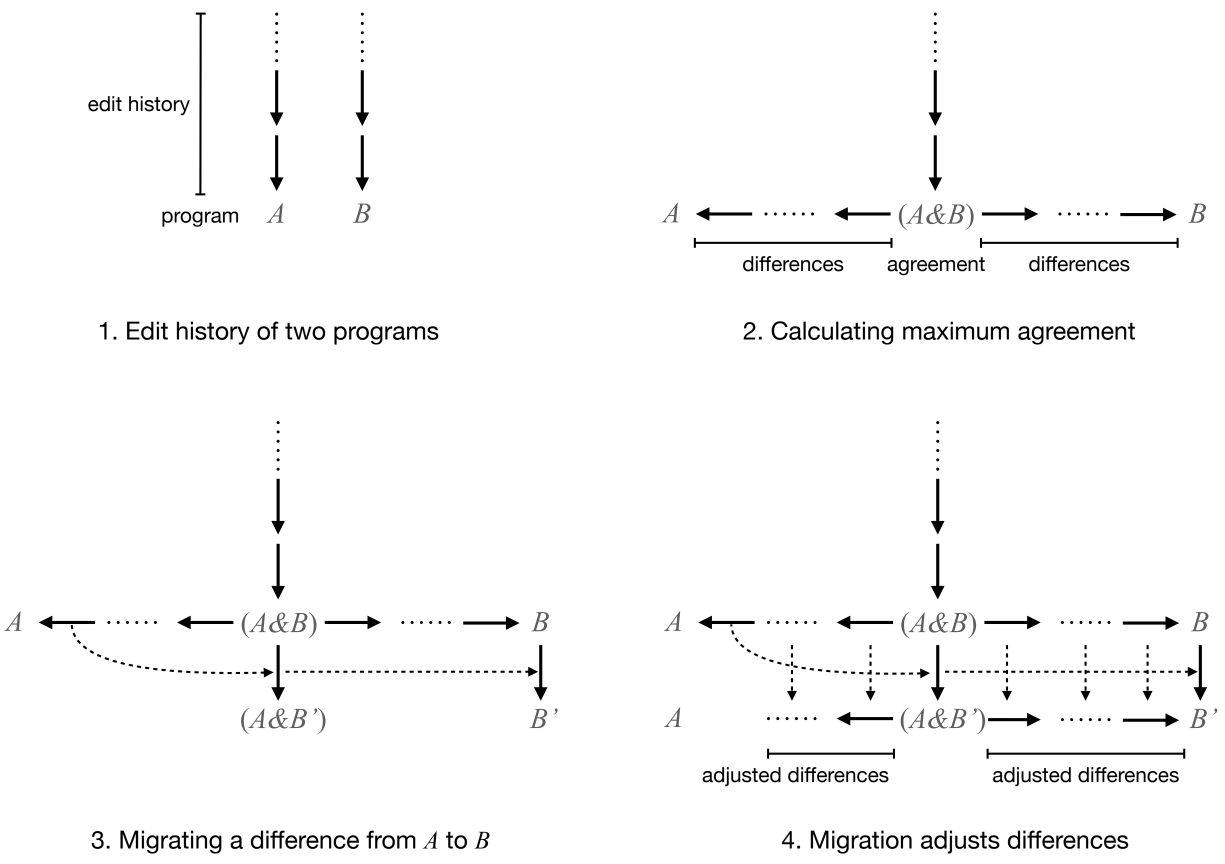

edit history
program
A
B
1. Edit history of two programs
A
(A&B)
B
differences
agreement
differences
2. Calculating maximum agreement
A
(A&B)
B
(A&B')
B'
3. Migrating a difference from A to B
A
(A&B)
B
A
(A&B')
B'
adjusted differences
adjusted differences
4. Migration adjusts differences

Figure 14. Version control with structure editing

differences relative to the new agreement, as shown in the bottom row of arrows in 14.4. Note that the migrated edit has been removed from the differences: migration monotonically decreases differences, eventually converging on two equal programs. However the order in which migration occurs can change the outcome, determining which side wins when edits conflict. The abovr informal narratie is made more precise in Edwards and Petricek [13].

With this background we can now explain how to extend structural version control to work between copies within a single version of a program. We start in Figure 15.1 with the edit history of a program. Say that subtree P within the program tree was copied to subtree Q. Figure 15.2 shows three programs derived analogously to an agreement between two programs. The agreement is replaced with the *separation* $P \mathbin{\overline{\vee}} Q$ (P nor Q) containing all edits that are not within P or Q and all pairs of equivalent edits within the subtrees of P and Q. Equivalency is defined by mapping subtree paths within P and Q. P and Q are equivalent in the separation. Forking from the separation are two histories of edits that create the differences between P and Q. The separation of P and Q serves like the agreement $A\&B$ as the "ancestor" for a 3-way comparison of the differences.

Migration is analogously generalized from the cross program case. In Figure 15.3 migrating a difference in P to Q adds two edits to the separation: an edit corresponding to the P edit and an equivalent edit copied to Q by mapping tree paths $P \mapsto Q$. The mapped edit is then migrated through to Q, generating Q'.

Since internal versioning emulates the 3-way comparison produced by cross-program versioning, the UI used for side-by-side comparison and migration across versions adapted relatively easily to work across copies within a program.

There is one nasty complication: the `move` edit, which moves a subtree to a new location and deletes the source. Moves involve two locations, so can be hard to separate. A move from inside P to inside Q (or the inverse) is reported as an error preventing separation, which seems acceptable because it violates our intuitions about copies. More troubling are the realistic cases of moves into or out of the separate subtrees from their outside. An incoming move is split into

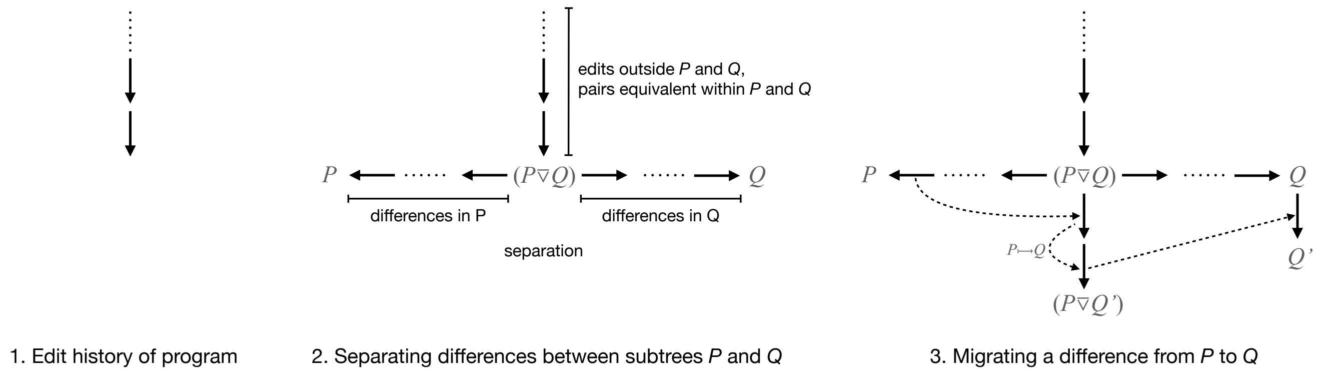

edits outside P and Q,
pairs equivalent within P and Q
P
(P▽Q)
Q
differences in P
differences in Q
separation
P⟼Q
Q'
(P▽Q')
1. Edit history of program
2. Separating differences between subtrees P and Q
3. Migrating a difference from P to Q

Figure 15. Internal version control

a deletion of the outside source in the separation and a sequence of edits recreating it in the differences. An outgoing move is conversely split into a deletion in the differences and recreation in the separation.

4 Discussion

Even at this early stage we can reflect upon the apparent benefits and disadvantages of our proposals. On the positive side, our theory of version control for structure editing has succesfully been extended to support internal version control. It handles subtle issues like copies of copies in a plausible manner. On the negative side, the user interface can feel complicated. To make code look familiar a lot of information is being hidden, and revealing that information quickly gets complicated. Further refinement of the UI would help, but this problem gives us pause. In offloading the cognitive burden of abstraction have we just replaced it with a different kind of difficulty?

Data on the use of refactoring tools [28] is not encouraging: renaming alone accounts for the vast majority of refactoring, and more complex refactorings are often done manually, foregoing the use of available tools. Even in our own example narrative it sometimes felt like it would be easier to just manually perform the needed edits than use our tools to automate them. On the other hand, our expertise with textual programming biases us, so experiments with non-experts would be needed to fairly assess the matter.

The elephant in the paper is the dependence on structure editing, which has a long and controversial history [3, 34, 39]. While there have been successes for beginning programmers [26, 33] and domain-specific languages [37] it remains experimental for general purpose programming [19, 30]. Structure editing has inherent problems of *hidden dependencies* and *viscosity* [4]. Professional programmers are deeply invested in text editing and do version control based on heuristic textual differencing [24]. We believed it necessary to accurately capture high level operations like copying, moving, and renaming, but perhaps heuristic text comparison could be made good enough with sufficient effort, indeed our case study in section 2 could probably be handled textually.

The startling ability of Copilot [17] to generate code completions raises the question of whether it couldn't also suggest changes to copies.

5 Related Work

There has been extensive research on *clone detection* [25] but less on managing clones. Kapser and Godfrey [22] discuss pragmatic reasons for cloning. Clone detection is typically done by textual comparison, which limits its accuracy. There have been attempts to extract more information by comparing ASTs to make merging more precise [1, 27, 36]. Duala-Ekoko and Robillard [11] continuously track clones during editing and allow simultaneous editing of pairs of clones. *Linked editing* [40] allows the programmer to identify clones and simulataneously edit them.

Synchronous copy and paste [7] tracks and maintains equality of copies. It presents an outline view of all copies, which might improve our UI. Copies can have text region *holes* that vary across instances, with those variations tracked in the outline view. These parametric copies can be used in place of functional abstraction as well as aspect oriented code adaptation.

Forms/3 [10] used *similarity inheritance* as a mechanism for managing copying in spreadsheets, providing automatic copying of changes from a source to its copies. *Gridlets* [21] provide similar capabilities. Hermans and van Der Storm [20] also track copying within spreadsheets but like our work allow changes to be manually migrated in any direction.

Variolite[23] is closely aligned with our work. It was informed by studies of data scientists confirming the common use of informal internal version control techniques through copying and commenting. Variolite allows a region of text to be associated with multiple alternatives that can be chosen from. These variant regions can be nested within each other, and their alternatives can be coordinated by tagging with named branches. Our work is in a sense orthogonal, exploring the spatial dimension rather than time. Variolite subsitutes alternatives in place at different times, while we capture copying and variation between different places within the same version.

6 Research Vision

"The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer." – Edsger W. Dijkstra

"One can't proceed from the informal to the formal by formal means." – Alan Perlis

Managed copy & paste can defer abstraction by providing interactive assistance for copy maintenance, but eventually there will be situations where abstraction into functions is called for. A key benefit of functions is to isolate and name key concepts. The logical next step in our research is to provide refactoring tools [15] to assist in function abstraction. Narasimhan and Reichenbach [29] have explored this idea with clone detection techniques. Because we have tracked the network of copies and the variations between them, we can parameterize these variations into functions. This refactoring need not occur all at once, but rather incrementally parameterize specific call-site variations while leaving others as the differences between copies of a function. When a function turns out to be parameterized not quite right, we would like to have the inverse refactoring: *de-abstraction*, where parameters are moved back into code customizations in a copy of the function, where it is easier to re-partition those customizations before re-parameterize along different lines.

We call this development process *gradual abstraction*. Traditional programming expects programmers to take large mental leaps to transform code variation into abstract functions. Instead we envision building abstractions via a series of smaller steps with much more interactive assistance. Managed copy & paste allows us to grow and maintain a pattern of code variation. Refactoring then uses that latticework to incrementally refine abstractions in small sound steps.

Managed copy & paste may also offer new alternatives to traditional programming language modularity constructs. Chiba et. al. [7] have also explored this idea. We observe that traits [35], especially in their original proposed form, can be seen as a set of edits that add, delete, and rename class members. That is exactly how we track the differences between copies. We believe that traits and other modularity mechanisms can be removed from the language semantics and instead supplied by the programming environment through managed copy & paste.

7 Conclusion

Our goal is to reimagine programming tools and languages in order to offload the cognitive burden of programming. This paper has prototyped one such tool, managed copy & paste, targeted at reducing the cognitive burden of abstraction. We follow in the philosophical footsteps of others who have questioned whether abstraction is detrimentally over-emphasised in programming theory and practice [5, 6, 9, 16, 38]. We hope to advance that discussion from criticising abstraction to offering a concrete alternative.

References

[1] 2022. SemanticMerge. https://www.plasticscm.com/semanticmerge/documentation/intro-guide/semanticmerge-intro-guide

[2] Harold Abelson and Gerald Jay Sussman. 2022. *Structure and interpretation of computer programs.* MIT Press.

[3] D.R. Barstow, S.G. Guty, H.E. Shrobe, and E. Sandewall. 1984. *Interactive Programming Environments.* McGraw-Hill, USA.

[4] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and Richard M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind (CT '01).* Springer-Verlag, Berlin, Heidelberg, 325–341.

[5] Alan F. Blackwell, Luke Church, and Thomas R. G. Green. 2008. The Abstract is an Enemy: Alternative Perspectives to Computational Thinking. In *Proceedings of the 20th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2008, Lancaster, UK, September 10-12, 2008.* Psychology of Programming Interest Group, 5. https://ppig.org/papers/2008-ppig-20th-blackwell/

[6] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming* 11, 2 (2001), 155–206. https://doi.org/10.1017/S0956796800003828

[7] Shigeru Chiba, Michihiro Horie, Kei Kanazawa, Fuminobu Takeyama, and Yuuki Teramoto. 2012. Do We Really Need to Extend Syntax for Advanced Modularity?. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development* (Potsdam, Germany) *(AOSD '12).* Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2162049.2162061

[8] E.W. Dijkstra. 1972. The humble programmer [1972 ACM Turing Award Lecture]. *Commun. ACM* 15, 10 (1972), 859–866. https://doi.org/10.1145/355604.361591

[9] Andrea A. diSessa. 2018. Computational Literacy and "The Big Picture" Concerning Computers in Mathematics Education. *Mathematical Thinking and Learning* 20, 1 (2018), 3–31. https://doi.org/10.1080/10986065.2018.1403544

[10] R.W. Djang and M.M. Burnett. 1998. Similarity inheritance: a new model of inheritance for spreadsheet VPLs. In *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254).* 134–141. https://doi.org/10.1109/VL.1998.706156

[11] Ekwa Duala-Ekoko and Martin P Robillard. 2007. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE'07).* IEEE, 158–167.

[12] Jonathan Edwards. 2006. *First Class Copy & Paste.* Technical Report MIT-CSAIL-TR-2006-037. MIT. http://hdl.handle.net/1721.1/32980

[13] Jonathan Edwards and Tomas Petricek. 2021. Typed Image-based Programming with Structure Editing. https://doi.org/10.48550/ARXIV.2110.08993 Presented at Human Aspects of Types and Reasoning

Assistants (HATRA'21), Oct 19, 2021, Chicago, USA.

[14] Jonathan Edwards and Tomas Petricek. 2021. Typed Image-based Programming with Structure Editing. https://vimeo.com/631461226

[15] Martin Fowler. 1999. *Refactoring* (2 ed.). Addison Wesley, Boston, MA.

[16] Gabriel. 1998. *Patterns of Software.* Oxford University Press, New York, NY, Chapter Abstraction Descant.

[17] GitHub. 2022. GitHub Copilot: Your AI pair programmer. https://copilot.github.com

[18] Garrett Grolemund and Hadley Wickham. 2017. *R for Data Science.* O'Reilly Media, Sebastopol, CA.

[19] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18).* Association for Computing Machinery, New York, NY, USA, 654–664. https://doi.org/10.1145/3180155.3180165

[20] Felienne Hermans and Tijs van Der Storm. 2015. Copy-Paste Tracking: Fixing Spreadsheets Without Breaking Them. In *ICLC 2015 - The first International Conference on Live Coding.* Leeds, United Kingdom. https://hal.inria.fr/hal-01261473

[21] Nima Joharizadeh, Advait Sarkar, Jack Williams, and Andy Gordon. 2020. Gridlets: Reusing Spreadsheet Grids. In *38th Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI '20 Extended Abstracts).* ACM. https://www.microsoft.com/en-us/research/publication/gridlets-reusing-spreadsheet-grids/

[22] Cory Kapser and Michael W. Godfrey. 2006. "Cloning Considered Harmful" Considered Harmful. In *2006 13th Working Conference on Reverse Engineering.* 19–28. https://doi.org/10.1109/WCRE.2006.1

[23] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17).* Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[24] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, V. Arvind and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 485–496.

[25] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13).* Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/1081706.1081737

[26] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15).* Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2818314.2818331

[27] Simon Larsen, Jean-Remy Falleri, Benoit Baudry, and Martin Monperrus. 2022. Spork: Structured Merge for Java with Formatting Preservation. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/tse.2022.3143766

[28] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[29] Krishna Narasimhan and Christoph Reichenbach. 2015. Copy and Paste Redeemed. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15).* IEEE Press, 630–640. https://doi.org/10.1109/ASE.2015.39

[30] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).*

[31] Tomas Petricek. 2019. Histogram: You have to know the past to understand the present. http://tomasp.net/histogram/

[32] Benjamin C Pierce. 2002. *Types and Programming Languages.* MIT Press.

[33] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[34] Erik Sandewall. 1978. Programming in an Interactive Environment: The "Lisp" Experience. *ACM Comput. Surv.* 10, 1 (March 1978), 35–71. https://doi.org/10.1145/356715.356719

[35] Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. 2003. Traits: Composable Units of Behaviour. *Proceedings ECOOP 2003* 2743, 248–274.

[36] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A Refactoring-Aware Software Merging Technique. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 170 (oct 2019), 28 pages. https://doi.org/10.1145/3360596

[37] JetBrains s.r.o. 2021. MPS: The Domain-Specific Language Creator. https://www.jetbrains.com/mps/

[38] Friedrich Steimann. 2018. Fatal Abstraction. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Boston, MA, USA) *(Onward! 2018).* Association for Computing Machinery, New York, NY, USA, 125–130. https://doi.org/10.1145/3276954.3276966

[39] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573. https://doi.org/10.1145/358746.358755

[40] Michael Toomim, Andrew Begel, and S.L. Graham. 2004. Managing Duplicated Code with Linked Editing. 173–180. https://doi.org/10.1109/VLHCC.2004.35

Received 2022-09-01; accepted 2022-10-02