No Ifs, Ands, or Buts Uncovering the Simplicity of Conditionals

Jonathan Edwards

MIT Computer Science and Artificial Intelligence Lab edwards@csail.mit.edu

Abstract

Schematic tables are a new representation for conditionals. Roughly a cross between decision tables and data flow graphs, they represent computation and decision-making orthogonally. They unify the full range of conditional constructs, from if statements through pattern matching to polymorphic predicate dispatch. Program logic is maintained in a declarative canonical form that enforces completeness and disjointness among choices. Schematic tables can be used either as a code specification/generation tool, or as a self-contained diagrammatic programming language. They give program logic the clarity of truth tables, and support high-level direct manipulation of that logic, avoiding much of the mental computation demanded by conventional conditionals.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Decision tables; D.2.6 [*Software Engineering*]: Programming Environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—control structures, polymorphism; D.1.7 [*Programming Techniques*]: Visual Programming

General Terms Languages

Keywords decision tables, control structures, polymorphism, visual programming, refactoring

1. Introduction

It is due to our choosing to accept or reject that we do not see the true nature of things. – Seng-ts'an, third Zen patriarch

Conditionals are the backbone of a program. They provide the essential ability to execute different code in different

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada. Copyright © 2007 ACM 978-1-59593-786-5/07/0010...\$5.00 situations, and they establish much of a program's structure. The hardware-given conditional branch instruction has evolved into a variety of programming language constructs: nested if-then-else blocks; switch statements; pattern matching in functional languages; and polymorphism¹ in objectoriented languages.

One reason for this variety of conditionals is that the most basic and general one, the if-statement, is not completely satisfactory. The structure of nested if-then-else blocks forms a tree of decisions which we must mentally compile from our informal understanding. Conversely, when we read nested if-then-else blocks we must decompile the tree back into a higher-level understanding. Walking down the tree, especially down else-branches, quickly becomes disorienting and forces us to reason with deMorgan's law. Making decisions in the wrong order can lead to duplication of subsequent decisions across branches. We often have to refactor the tree to reduce duplication or to find a single resting spot for some new logic. It is hard to tell whether a decision tree covers all possible combinations of cases, and it is hard to tell how different trees logically overlap. Graphical representations of decision trees, such as flowcharts and Nassi-Shneiderman diagrams [30], have the same problems. Studies of the difficulties caused by control structures are surveyed in Pane and Myers [32], p.30–31.

Newer forms of conditionals (switch statements, pattern matching, and polymorphism) respond to these problems in similar ways. They partition code into distinct cases and declaratively specify the conditions in which to invoke them. Declarative conditions tie the cases to data definitions, and allow compile-time analysis to detect errors and offer assistance in an Interactive Development Environment (IDE). The compiler takes care of generating an executable decision tree. However none of these constructs are flexible enough to completely supplant if-statements, which are still needed for the ad hoc glue logic which does not sort out along any simple dimension. This paper introduces *schematic tables*, a new representation that attempts to be flexible enough to replace if-statements, and declarative enough to replace pat-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

¹Throughout this paper, polymorphism will be taken to mean subtype polymorphism.

tern matching and polymorphism, while making all of them easier to understand, and easier to manipulate.

Truth tables have long been used to express logic more clearly than symbolic propositions. A number of tabular specification languages have been found beneficial [16, 18, 24]. Decision tables [34, 39] have been used both for specifications [13] and as a programming language feature [36, 38]. Figure 1 shows an example of a decision table from Kirk [22].

Decision tables express complex logic clearly, but suffer from a number of limitations that have impeded their adoption. Composing decision tables leads to a combinatorial explosion, and worse, the tacit duplication of logic and decisions. The response of splitting and nesting tables re-establishes the tree structure of if-statements, with all its disadvantages. Tables require that exactly one column be true in all cases, a constraint that can be hard to maintain. A study [15] comparing decision tables to if-statements found the former to be preferred but less accurate. Another study [41] found decision tables inferior to English structured by if-statements for comprehension tasks. It is interesting that a study [14] comparing textual and graphical programming notations used an apparatus similar to decision tables as the reference point. Overshadowing these issues is the fundamental limitation of decision tables: they are not complete programs. We must add code to evaluate conditions and execute actions, and we must structure that code to interface with the table. The program is torn between two worlds. Schematic tables attempt to generalize decision tables into a self-sufficient programming language.

The key ideas of schematic tables are:

- 1. Computation and logic are represented orthogonally, in the vertical and horizontal dimensions of a table, separating their different semantics.
- 2. The two dimensions of tables provide an extra degree of freedom over textual syntax for structuring code, and reduce the need to split up related code.
- 3. Maintenance of a single logical invariant (partitioning), drives high level semantic transformations that reduce programming effort.
- 4. Schematic tables are canonical, in a certain sense representing equivalent programs equivalently. Diagrammatic notation is made feasible by eliminating manual layout. The need for refactoring is reduced.
- 5. All forms of conditionals are unified, from ad hoc ifstatements to pattern matching and polymorphic predicate dispatch.
- 6. A declarative language of predicates ties together the notions of types, assertions and conditionals.

Schematic tables have been developed within the Subtext project [10], whose goal is to make programming easier by altering the representation of programs. However they may be applicable apart from Subtext, as a specification language or code generator that goes deeper into implementation details than decision tables and tabular specification languages. They may also be useful as a visualization of conventional code within an IDE that maps table changes into code refactorings and transformations. Such applications are left for future work. This paper will instead present schematic tables in their fullest form, as part of the Subtext language, but will only discuss Subtext in so far as is necessary.

Schematic tables have not yet been implemented, and some details of their design are unsettled. The diagrams in this paper are "paper prototypes" from the user interface design of Subtext 2. They have been placed at the end of the paper so that they can be torn off and followed separately (or viewed in a separate window). A video demo of the implementation will be made available at http://subtextual. org/subtext2.html. Color is an important visual aid in the user interface — a full color version of this paper can be found at http://subtextual.org/OOPSLA07.pdf. Schematic tables will be presented in stages, starting at the level of decision tables, then adding in turn the capabilities of functional programming and polymorphism. The presentation will focus on the improvements to the programming experience.

2. Simple Schematic Tables

The process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.

- Turing [40]

We start by introducing a simplified form of schematic table comparable in power to decision tables, but with advanced IDE support that keeps tables in a canonical form free of logical errors. The programmer is freed from doing Boolean algebra. Figure 2 shows some Java code and the corresponding schematic table. The important features of this table are:

- 1. Each row of the table corresponds to a variable in the program, which is defined in the first column in blue. Each column of the table states what to do in a certain situation.
- 2. The second column corresponds to the Java conditional if (a & b) x = 1; The cells for a and b contain gray boxes labeled y which are called *predicates*. Predicates represent a test applied to their row's value. The y predicate tests whether a Boolean variable is true, n tests for false, and a blank cell doesn't care. The second column is true when all its predicates are true, in other words when both a and b are true.
- 3. The cell for x in the second column contains the number 1, which is called a *value*. When a column is true, the

values in the column are assigned to their row's variables, which in this case sets x=1.

- 4. The third column corresponds to the Java conditional if (!a | !b) x = 2; To express this disjunction a vertical dotted line divides the predicates within the column into *subcolumns*. The entire column is considered to be true whenever it contains a subcolumn in which all the predicates are true. The left subcolumn contains only the predicate n on the variable b, which tests that it is false. The right subcolumn contains the predicate n on the variable a, testing that it is false. Thus the column is true when either a or b is false, in which case x is assigned 2. Note that values belong to the entire column, not subcolumns.
- 5. The division of columns into subcolumns is known in logic as disjunctive normal form. Every predicate has a complementary form that is its negation (e.g. y and n). The possibly negated predicates within each subcolumn are conjoined (logical AND) and then all of the subcolumns are disjoined together (logical OR). The significance of disjunctive normal form is that it can represent all possible logical propositions. A similar tabular representation of queries is presented in Pane and Myers [31].

The benefit of the table is that it clearly reveals the conditions under which x=2, whereas the Java code leaves it up to us to work out the logic of the else block using deMorgan's law: $\neg(a \land b) = (\neg a \lor \neg b)$. That may not be an undue burden in this trivial example, but as we will see even slightly more complex conditionals can quickly lead to bewilderment.

Unfortunately truth tables do not absolve us of doing mental Boolean algebra. They merely shift the burden onto the construction and maintenance of the table, which can require just as much effort, and be as error-prone. As mentioned previously, it is quite easy to build tables that have gaps or inconsistencies in their logic. The same problem arises in the Java code of Figure 3, which is a transliteration of the table. The conditions for each case are made explicit, but with the maintenance burden of ensuring that exactly one of the if statements executes in all cases. It seems that we are faced with a dilemma: having to do mental Boolean algebra either when we read our code or when we change it. Schematic tables circumvent this dilemma by doing the calculations automatically while editing within an IDE. After all, it is absurd to be doing Boolean algebra in our head when there is a computer sitting right in front of us.

2.1 Logical editing

It has long been realized that decision tables can be automatically validated [21, 33], and some contemporary products [25, 37] do so interactively, immediately revealing errors. Schematic tables instead provide high-level editing operations that keep the table free of gaps and overlaps at all times. Errors are essentially automatically fixed, although the programmer may need to make subsequent adjustments to home in on a desired result. As an example, we will show how the table in Figure 2 could have been created.

- We will start with the table in Figure 4(a), which unconditionally assigns x=2. We need to add logic to assign x=1 when a and b are true. A new column is introduced via a menu operation, resulting in 4(b).
- 2. Next we put y predicates into the cells for a and b in the column with a right-click menu that suggests possible predicates², resulting in 4(c).
- 3. The value to be assigned to x is edited into the proper cell in 4(c). Assigning x creates an overlap conflict with the existing assignment, which is automatically resolved by creating a new column restricted to the complement of the conflicting one, as shown in 4(d).

The IDE calculates deMorgan's law for us in order to keep the table free of overlaps and gaps. The precise invariant is that all columns assigning the same variable must be logically disjoint (no overlaps), and together must completely cover all possible cases (no gaps). These two properties combined are called *partitioning*.³ By automatically maintaining partitioning, the programmer is freed of the drudgery of Boolean algebra, whether they are reading or writing code.

2.2 Transactional editing

Changing program logic often involves a series of coordinated edits in different places. While making these edits we may temporarily disrupt the program, requiring compensating edits elsewhere. Schematic tables support a transactional editing mode that allows us to perform a series of edits while visualizing their cumulative impact, deferring commitment until we are satisfied.

For example, Figure 5 shows some Java code and its corresponding table. Suppose that we discover a bug: the cases when b and c are true should set x=3. We want to preserve the existing behavior in all other cases. Further, we want to make this change without copying the assignment x=3, since that might cause undesirable code duplication. Skeptical readers are invited to solve this mundane problem before reading ahead.

1. Since we want to add new cases that assign x=3, we use a mouse-menu operation to insert a new subcolumn in the right-hand column, resulting in Figure 6(a). This edit is being performed in the transactional mode that displays the cumulative impact of a series of changes and allows the programmer to either complete or cancel them. The red box around the new subcolumn indicates that it is a set of states being added into the column. Since the

 ² The suggestions are based on the variable's type, to be discussed in §3.
 ³ Observe that Dijsktra's *guarded commands* [8] use overlapping to express nondeterminacy, whereas schematic tables are disjoint and deterministic.

subcolumn contains no predicates, it is always true, and thus conflicts with all other columns and subcolumns. All of them are X-d out, indicating that they are states currently marked to be subtracted by the transaction.

- 2. We continue by limiting the tentative new subcolumn to the states of interest by inserting y predicates on b and c as requested. The result is 6(b), which has divided up the other two columns to show precisely which states now need to be subtracted to keep the table partitioned. This information may be important: it tells us, case by case, what the impact of our changes will be on the old logic.
- 3. Clicking on the green arrow completes the transaction, deleting the X'd out cases and leaving us with the final solution in 6(c).

The solution in Java is shown in Figure 7, with the changes in bold. The author's informal sampling has found no one (including himself) who could immediately get this right. The schematic table makes the change straightforward, turning over the mechanical calculation to the machine itself.

2.3 Canonicity

Schematic tables have an important property: they are *canonical*. There is only one way to express each different meaning⁴. The only control the programmer has over the table's layout is the order in which the variables are defined in the left hand column (and similar orderings, such as the order of enumeration constants). These orderings are extended to columns in a modified lexicographical manner, the details of which will be omitted. The important fact is that the predicates within columns are reduced to Blake canonical form [1, 4], which guarantees that the subcolumns represent all of the simplest possible examples of the cases being selected. These examples are called prime implicants [35] in logic, and help to make schematic tables *example centric* [9]. Canonicity will be preserved as we enrich schematic tables throughout the rest of the paper.

3. Functional Schematic Tables

In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds. – Fred Brooks [3]

This section addresses the fundamental limitation of decision tables: that they can not represent arbitrary computations, only logic. Here schematic tables are turned into fullfledged functions that are callable, and in turn call other tables, and even recurse upon themselves. The basic idea is to represent expressions as trees of function calls, laid out vertically across rows of the table, with one function call per row. Useful new views of code are enabled. The standard notions of types and assertions are also introduced, but with the twist that they share the same declarative language of predicates with conditionals, leading to some convenient interactions among what are normally only loosely coupled language features.

We will restrict ourselves to the simple case of functions that have expressions inside conditionals, but not the other way around. The general case will be handled in the next section. Figure 8 shows the Fibonacci function implemented in Java and Figure 9 is the equivalent schematic table. The important features demonstrated by this table can be summarized as follows:

- Function arguments are represented as *input variables* of the table, suffixed by a colon, and function results are represented as *output variables*, suffixed with an equals sign. All functions use the standard input variable in. Binary operators use a second input with. All functions use the standard output variable out. The value assigned by the table to out is the return value of the function.
- Predicates are generalized to test for type membership (∈) and arithmetic inequalities (<, ≱, ≠, etc.). Predicates lacking a relation symbol are equality tests. The implicit left-hand argument of a predicate is the variable defined in its row.
- 3. Predicates are applied to variables in the first column to represent classical types, with compile-time⁵ type checking.
- 4. Predicates can be used as *assertions*, which are run-time tests that throw an exception if they are false (but only if they are in a true column). Assertions do not affect the truth conditions of their containing column. Assertions are displayed as an octagonal box and turn red if violated (like a stop sign).
- 5. Expressions containing nested function calls are represented as inverted trees of boxes. Each box is a function call, and is on its own row of the table. The last column of the table assigns out from the function tree equivalent to the expression fibonacci(in 1) + fibonacci(in 2) whenever the input is ≥ 2.
- 6. Vertical lines are used to connect function calls within an expression. Inputs enter the function box from the top, and outputs leave it from the bottom. The positions of these lines are shown in the key in Figure 10.

⁴ "Different meaning" is non-equivalence in the decidable logic of predicates discussed in §7.

⁵ There is no difference between compile-time and run-time in Subtext, since all code is always running. To preserve familiar terminology, "compile-time" will be taken to mean a static analysis that ignores the actual values of variables and hence applies to all possible executions, whereas "run-time" properties are specific to a single execution instance.

- 7. The main input and output lines of a function are aligned, and so composing functions through their first argument, as in f(g(h(...))) leads to a vertically aligned stack of boxes. Secondary inputs indent their function stacks and place them under the stacks of left-ward arguments.
- 8. Function arguments that are variables or constants are named inside the box at the same position as a line would enter. Function trees are actually graphs; the devices used to represent cross-cutting edges are discussed elsewhere [10], but will not be used in this paper.
- 9. Editing of function trees can be performed with a number of mouse and keyboard operations [10], the most important of which is the *splice* operation. A function box can be dragged off of a palette, or out of existing code, and then dropped onto one of the vertical lines of the tree. The function will be spliced into the tree at that point, cutting the line and connecting the two ends into the function along its primary input/output axis. Connections between functions can also be altered by dragging lines around, changing the shape of the tree.
- 10. The function tree layout algorithm is canonical, driven by input argument order and the connections between function inputs and outputs. The programmer controls only those connections. There is no manual placement and adjustment as with many visual programming languages [5, 29]. Neither is there fussing with carriage returns and tabs to beautify textual layout.

Function trees offer some advantages over traditional parenthetically nested expressions. They can be read linearly, following the visual cues of the lines, either up or down. Nested expressions can quickly becomes disorienting: after all, we are not stack machines. The vertical layout across rows enables some new ways to visualize execution and browse code, described below.

3.1 Live code

Schematic tables help not only to visualize the logic of code, but also its actual execution. Since every variable and function call has its own row of the table, execution values can be annotated into the rows, as shown in Figure 11 for an input value of 2. The values of every variable and the primary output of every function call are shown prefixed with an equals sign. True predicates are colored green, false predicates gray. False columns are shaded entirely gray, showing that their contents are inactive, and in contrast highlighting the true columns.

In Subtext all code is always a running instance, so displaying these execution values merely reveals what has been kept hidden until now. But the same presentation could be used to illuminate code within a conventional debugger, by selecting a stack frame of the function. Unlike traditional break-and-step debugging, the entire execution of the function is made visible at once. Function trees offer a convenient place to annotate complete traces of execution values within the code itself. A previous paper [9] discussed an implementation technique for Java and the implications for the development process.

3.2 Laminar browsing

Since function calls are arrayed in the vertical dimension, the horizontal dimension can be used to zoom in on them nondisruptively, with a technique called *laminar browsing*. Figure 12 shows how any function call can be expanded rightward to see its internals. This example shows drilling into the recursive structure of fibonacci to explore a particular execution, but it could equally well be used to browse between different functions and definitions during editing. Structured values can also be expanded in the same way.

Multiple expansions are stacked up in columns of the browser window called *layers*. Vertical scrolling is "geared" between the layers, with rightward layers scrolling faster than leftward ones in order to keep related information synchronized on screen. Expanding from the right-most layer opens a new one to the right, and scrolls the entire display leftward. Backtracking is done by scrolling the entire display rightward to see the contextually leftward layers. This approach maps out a tree of traversals, unlike the strictly linear back/forward history of browsers. It also shows the context of where you came from and where you are going, unlike browsing by opening independent windows.

3.3 Types, assertions, and conditionals

Predicates foster a convenient interplay between types, assertions, and conditionals, which are only loosely related in conventional languages. Because assertions and conditionals use the same language of predicates as types, they have meaning at compile-time. Predicates satisfy types within their column, as if they were implicit local downcasts. One can impose specifications on a function by publishing arbitrary predicates as types, obligating the callers to satisfy them with assertions or conditional guards (the aforementioned downcasts generalized to arbitrary properties). Assertions of arbitrary predicates flow out of a function at compile-time, like return types, to latch conditionals or satisfy types, even though they are run-time checks. Assertions may occur in other columns than the first, conditionalized by the predicates of that column to form logical clauses⁶. Overall, the ubiquitous use of predicates enables programs to be more declarative.

⁶ If such clausal assertions could be incorporated into reasoning within the function's callers then we start to get the features of a formal specification language. Future research will explore this direction.

4. From Java to the Outer Limits

We will control the horizontal. We will control the vertical. – The Control Voice⁷

In this section a larger example introduces the full capabilities of schematic tables. Tables will be generalized to support multiple independent conditional tests, and these tests can select the inputs to functions. The central idea is that computation is represented in the vertical dimension of the table, while decision-making is in the horizontal dimension. Orthogonalizing these two concerns allows their different semantics to be manifested directly in the geometry and behavior of the table. Maintaining logical partitioning of the table drives high-level transformations that save labor. The extra degree of freedom offered over textual syntax reduces the need to split code across control structures.

Figure 13 shows the Java implementation of a function that calculates the damage of an attack in a video game, known as hit points. Attacks are categorized by the Attack enum into MELEE and MAGICAL, which draw upon different attributes of the attacking character: its strength or magic. A random number modulates the effectiveness of the attack, unless the attacker possesses the magical Amulet of Puissance, which bestows 80% effectiveness. The damage to the target depends on the type of the attack. A magical attack is damaging in inverse proportion to the target's magical power, while a melee attack is damaging only to the extent that it overpowers the target's armor. To simplify matters, all numeric values are assumed to be normalized to the interval [0 - 1].

Unfortunately, as so often happens, the requirements do not cleanly factor out into a simple switch on the type of the attack, leading to a design dilemma. There is shared logic between the cases. Copying the shared logic across different cases would be undesirable duplication. Yet abstracting it into a separate function is of questionable benefit because it is so intertwined with the rest of the code. Using two switch statements to centralize the shared logic, the choice taken here, is redundant and splits apart related code. The point is not to debate this dilemma which Java thrusts upon us, but rather to show how schematic tables can avoid the need to make such difficult trade-offs.

Figure 14 shows the same function as a schematic table, illuminated by a live example execution. The important features of this table are:

1. More than one column can be true at a time, allowing independent conditions to be tested without a combinatorial explosion of cases. The graying-out of false columns in the live example highlights by contrast the three true columns, plus the first column (which is only false when there is a type error).

- 2. Function inputs can involve choices, indicated by splitting the vertical connector line across multiple columns, connected by a horizontal line. For example the first input to the '*' function in the first column is a choice between .8 and a call to random, based on the value of the attacker.amulet field.
- 3. The partitioning invariant is generalized to require that there be exactly one true column among all the choices for a variable or function input. The question mark in the bottom row adds a "sub-partitioning" constraint, which is explained in §4.5.
- 4. Function inputs and outputs can be named like local variables, as in power and effectiveness. However these names are optional and for documentation purposes only they are not required like in textual languages as a form of exposed plumbing. In the Java code, we wouldn't have thought twice about collapsing power and effectiveness into a nested expression if the switch statement hadn't forced us to split out the code.
- 5. Predicates can test the result of internal computations, shown by the <0 and ≥ 0 predicates evaluating the result of the subtraction to their left. Note that the ≥ 0 predicate allows the value it is testing to flow through to the output without having had to assign it first to a local variable.

Columns unite code that is used under the same conditions, whereas textual conditionals often force it to be split apart. The dilemma of Figure 13 is resolved: the code that was split across the two switch statements is now unified vertically within columns, while the shared logic in the leftward columns is jointly accessed horizontally. Another common annoyance is having to save a Boolean in a "flag" variable which is then repeatedly tested by conditionals scattered throughout the code. A schematic table would unite the bodies of those conditionals in a single column making a single test. We will see in §5 how columns can also unite code split across polymorphic methods. The more flexible connectivity within a schematic table is because it has two dimensions, whereas text has only one (or perhaps 1.5, counting indentation).

4.1 Controlling the horizontal

The table is still canonical. The programmer can not "place" function calls in cells. Vertical placement is determined by the tree layout algorithm described earlier. Horizontal placement is determined by a *relevancy* analysis.

The downstream consumers of the call's outputs are traversed to determine the union of all cases in which they are used; the call is then placed in a column predicating the computed relevancy conditions. The leftmost '*' call is used in all cases, so it is placed in the first column. The other '*' call is placed in the last column because it is only used for magical attacks. The relevancy analysis frees the programmer from having to conditionalize function calls based on

⁷http://en.wikipedia.org/wiki/Outer_Limits

when they will be needed, as in a lazy language. But unlike lazy languages, the inferred predicates make it clear exactly what those conditions are, which may be important to know. They act like explicit conditionals, including satisfying input types of the called function.

As mentioned earlier, the horizontal ordering of columns is also canonical. An important property of this ordering is that it is a topological sort of the implication ordering. For example, the column which conjoins the predicates melee and <0 implies the column with just melee: columns that are true in a set of cases are implied by those that are only true in a subset of those case. Therefore the melee-only column is placed to the left of the more restrictive ones. The "truthiest" column is always the left-most one containing the types defining all possible cases. Self-contradictory columns will appear on the right-hand side (and be shaded extra-darkly). This ordering property means that the geometry of the table tells us something about its logic.

4.2 The central idea

The central idea of schematic tables is that function trees show computation vertically, while truth tables show logic horizontally. Computation and decision-making are thus represented orthogonally. The geometry and behavior of the table manifests the different semantics of these two dimensions: computation that of data flow graphs; decisionmaking that of Boolean algebra. Separating these two aspects of a program makes it easier to understand, and easier to change. Having two independent dimensions offers an extra degree of freedom over linear text, reducing the need to split apart related code.

4.3 Restriction

Schematic tables can be *restricted* into a partial view that assumes certain predicates are true. Figure 15 shows the restriction on magical attacks (and with live execution values turned off). The restriction predicate is displayed in yellow. All columns that are contradicted by the restriction assumptions are hidden. All predicates that are implied by the restriction assumptions are also hidden. Note that a restriction is a fully editable view of the schematic table.

4.4 Pattern matching

The way that schematic tables divide the code into cases which are guarded by predicates is somewhat like pattern matching [6] in functional languages. The declarative nature of patterns allows completeness to be enforced at compiletime. However pattern matching has some limitations:

 Pattern matching enforces completeness, but not disjointness — overlapping patterns are allowed, with the "first fit"⁸ being executed. First fit is like doing "implicit else" with a return or break statement. It encodes logic tacitly in order, making it terser but harder to understand, and harder to preserve under change.

- 2. Pattern cases are laid out vertically in the source file, while schematic tables do so horizontally, orthogonal to the code. Lining up all the predicates on the same variable in a row makes the differences between patterns more visible, as in truth tables. Note how the attacker.amulet field is "destructured" into a box occupying its own rows of the table, lining up all the predicates on it. Destructuring variable assignments could be likewise represented.
- 3. The hitPoints example would require two separate pattern matches, just as it requires two switch statements in Java: shared code between the cases causes the data flow to wind out and back in to each case. This flow is easily represented in the two dimensions of a schematic table.

4.5 Code transformation

Maintenance of the partitioning invariant on tables drives high-level transformations that save programmer labor and thought. We saw this in §2 for simple decision tables; we will demonstrate it now for general code editing. For example suppose that we introduce new weapons technology into the game: ranged weapons that can attack from a distance. We want the hit point calculation for ranged weapons to be the same as for melee attacks, except that a new ability, acuity, will modulate its effectiveness. We adjust the code in a sequence of simple steps:

- 1. Appending ranged to the Attack enumeration automatically changes the table as in Figure 16⁹. A new column has been added to keep the table complete, but using an exclamation mark as the value of the assignment. The exclamation mark is a special error value that indicates a missing value requiring resolution by the programmer. It is highlighted in red, and also has the effect of violating the Rational type on out, which likewise flares up red. We can resolve this problem by replacing the exclamation mark with the appropriate code for the new case. A pattern matching language would have reported the missing case as a compiler error. The schematic table goes further by also generating the boilerplate for fixing it.
- 2. To share the logic of melee attacks with ranged ones in Java, we would add RANGED onto the case selectors for MELEE in both of the switch statements. We do the same thing in the schematic table by dragging the question mark onto the exclamation mark. The question mark serves here as a group handle on the pair of values imme-

⁸ Hope [6], which first introduced pattern matching, used a "best fit" semantics that arguably overcomes this objection. This feature was discarded in

subsequent functional languages, although it has been revived in a recent proposal [26].

⁹ Note that the two subtraction functions have been placed on the same row. This is an abbreviation to save vertical space by reusing a row for multiple functions when it is known that they are in contradictory columns. This abbreviation is a user-configurable parameter.

diately to its right, which together cover the melee case.¹⁰ The drag action merges the conditions of the target column (ranged) with the actions of the source columns (melee), resulting in Figure 17. The target.armor logic that before was in the melee columns is now predicated on being melee or ranged (using a comma to abbreviate the disjunction). But the choice of attacker.strength for melee attacks was moved into a new column to the right, because it applies in more restricted conditions, in accordance with the canonical ordering of the table.

- 3. The previous change triggered a new incompleteness error. The choice between attacker.strength and attacker.magic does not cover ranged attacks, so an exclamation mark is again automatically introduced to fill the gap. A type error is triggered on the * function on the left, which turns red.
- 4. We fix this error by replacing the exclamation mark with the desired reference to attacker.acuity, producing the final result in Figure 18.

Introducing the new enumeration constant ranged led us by the hand in writing the code to deal with it, step by step, driven by maintenance of the partitioning invariant of schematic tables.

5. Duck Polymorphism

Object oriented languages offer subtype polymorphism as a powerful form of conditional. Polymorphic dispatch decides which method to choose based on class — in other words conditionalizing on class. The power of polymorphism is that it makes class-based conditionality declarative. One can infer things about method dispatch at compile time, which IDE's use to great advantage. Polymorphism imposes structure on the code that helps identify the role of methods just by their containing class. Schematic tables, as already described, support all the essential features of polymorphism, albeit in an unconventional manner. The argument is that schematic tables walk like polymorphism, and quack like polymorphism, and hence provide "duck polymorphism". There are two essential features of polymorphism¹¹:

- 1. Dividing code into methods that are specialized by their containing class and the types of their arguments.
- 2. Overriding of methods via dynamic dispatch.

The basic idea is, firstly, that restrictions on schematic tables offer specialized views similar to methods, but without physically dividing up the code. Secondly, that table partitioning induces overriding logic between columns that imply each other, without having to invoke hidden run-time machinery. Duck polymorphism offers some advantages. "Methods" can share code, reducing their fragmentation. Method dispatch is revealed as explicit logic in the table. Inherited behavior and possible dynamic overridings are clearly visualized. The declarative power of polymorphism is made flexible enough to replace ad hoc if-statements.

5.1 Restriction replaces method division

Figure 19 replaces the switch statements in hitPoints with polymorphism in the standard manner. The Attack enum is converted to a class with subclasses Melee and Magical. Seven methods are produced. The arrows show the crossclass transfers of control for a call to Melee.hitPoints. Remember that classes are typically defined in separate files, so even in modern IDE's the programmer will not see them together as in Figure 19. Comprehending how all the little fragmentary methods tie together when one doesn't actually see them together is a central difficulty of polymorphism.

Schematic tables don't need to be divided into little specialized methods. Instead, we can use restriction to view specialized aspects of the table that correspond to methods. Figure 20 shows the trivial conversion of the table, replacing equality by instance membership. Figure 21 shows laminar browsing from a table that calls the hit points function within a column contingent on Magical attacks. The call is expanded to the right, automatically restricted on the predicates of the calling context. The restriction shows us just the parts of the table relevant to Magical attacks, like the method which would be dispatched in this situation.

Restrictions act like methods in the sense that they show a class-specialized fragment of the code. They are specialized declaratively like methods with predicates, allowing static analysis, partitioning constraints, and IDE assists. Restrictions are independently editable, just as if they were physically distinct methods. But restrictions have an advantage over methods: they can share code without extracting it into another method, reducing method fragmentation.

The table contains two "methods": the columns predicated on \in Melee and \in Magical respectively. There is no need to extract the shared logic about amulets and randomness into the Java effectiveness method. That code shows up in both method restrictions, flagged as being shared because it is located in columns to the left of the yellow restriction column. Edits to such shared code are visible in all sharing restrictions, just as editing an extracted method would be. We don't need to extract the power method either, which was done solely to allow it to be called from effectiveness. Now the code that had to be split across hitPoints and power sits in the same column. Polymorphic restriction provides more coherent class-specialization

¹⁰ The question mark establishes a "sub-partitioning" constraint among columns. A question mark is like an "abstract value", in the sense of an abstract method, laying down a marker that obliges the value to be assigned in all columns which logically restrict that column, and which together must cover all cases of that column. This constraint is analogous to the way that an abstract method must be overridden in all non-abstract subclasses.

¹¹ This section will actually show how schematic tables emulate the features of generic functions [28], while using the more familiar terminology of methods as if there were no difference. The difference is primarily one of syntactic packaging: there are well known ways to turn functions into methods, as for example in Python.

of code than methods do, because it is orthogonal to data flows.

5.2 Implication partitioning replaces dynamic dispatch

Dynamic dispatch is the other essential aspect of polymorphism. It allows us to incrementally add code specialized for certain subclasses and have it be invoked when needed. *Predicate dispatch* [12, 27] provided the key insight that method dispatch is driven by logical implication. A subclass method overrides a superclass method because an object being in the subclass implies it is also in the superclass. Predicate dispatch adds a decidable logic of predicates that allow dispatching on conditions other than class membership, with the rule that implicants override. Predicate dispatch also integrates overloading, by including predicates on arguments in the implication. This logical interpretation of polymorphism fits right into schematic tables.

As an example, we will add the ranged attack from §4.5 polymorphically. Since it differed from melee attacks only in the acuity parameter, it would be simplest to add Ranged as a subclass of Melee. But ranged attacks are not conceptually instances of melee attacks, so the proper thing to do is to introduce a new abstract class Physical which has subclasses Melee and Ranged. Figure 22 shows the resulting Java code. We have decided that the default for physical attacks should be to rely on the attacker's strength, and only override this in the case of ranged attacks. The same change is easily made to the schematic table. Too easily — instead we will make Physical non-abstract to expose more general dispatch logic. The steps are as follows:

- Figure 23 shows the effects of altering the class hierarchy. Completeness induces a column showing there is a gap in the logic for non-Melee Physical attacks, indicated as before with the exclamation mark error value. This error would be reported in Java as the failure to override the abstract method Attack.hitPoints in Physical. The table error identifies precisely what cases are missing, not just their superclass. It also provides the boilerplate for a fix, in the same way as was seen earlier with enumerations.
- 2. As was done in Java, the existing Melee logic needs to be promoted up to the Physical class. This is done by dragging the ∈Melee predicate and dropping it in the error column. The cases of the columns are merged (overwriting the conflicting exclamation mark), producing Figure 24, which differs from the starting point only in that the ∈Melee predicate has turned into ∈Physical. The equivalent change in Java would typically be done by renaming the Melee class to Physical.
- 3. Next the overriding behavior for Ranged attacks must be added, as in the Java method Ranged.power. Overriding is introduced by editing within a restriction. Figure 25 shows the restriction on Ranged. There is no code spe-

cific to ranged attacks yet, so the restriction column is empty. Note that the "inherited" behavior is shown in the columns to the left, predicated by source class (in this case Physical). The value of attacker.strength is overridden within the restriction by directly editing the proper cell in the asserted column, as shown by the text field in Figure 25. This edit will cause attacker.strength to disappear from the restricted view, as it has been overridden.

4. To see the big picture, Figure 26 lifts the restriction up to the superclass Physical. There is a column on the right hand side predicated on Ranged containing the overriding value just entered. The value it overrides, attacker.strength, is in the column to its left, predicated on being Physical but not Ranged. These predicates reveal precisely the dispatch logic of the power method in Java: the Physical.power method will be overridden only by Ranged.power. This logic is shown in a pleasingly symmetric form using the negative predicate ∉. The method dispatch algorithm has essentially been compiled into the predicates, induced by the partitioning invariant on columns that specialize (imply) others.

The schematic table interpretation of polymorphism has some limitations compared to languages designed expressly for the purpose. For example, there is currently no mechanism equivalent to Java's super. But schematic tables also offer some benefits. The specialization of code into methods is achieved more flexibly by restricted views, permitting code sharing without method fragmentation. The logic of method dispatch is revealed explicitly, and induced by the partitioning constraint on tables. Overriding is introduced by editing within a restriction. Restrictions offer a view from one point in the class hierarchy of both inherited and overriding code. The power of predicate dispatch is provided, but given an extra degree of freedom so that it can apply to ad hoc internal logic, not just at method calls.

6. Untangling time and logic

Up until now we have ignored a crucial issue: execution order. The conventional approach is to delicately intertwine actions into the control-flow of conditionals. Schematic tables allows us to untangle time and logic, representing them orthogonally.

The Subtext approach to time and state has been discussed in detail elsewhere [11]. A brief sketch will be made of how time can be represented in schematic tables in a way compatible with either a conventional semantics or that of Subtext. The basic idea is to represent execution order as an explicit data flow. All operations with side-effects are represented as functions called *actions* that transform *states*. By convention they take an *input state* on their primary input argument and produce an *output state* on their primary output. These states can be seen as symbolizing points in time¹². Actions are ordered by being stacked up along their primary axis. All state-inspecting operations, like pointer dereferences, are considered to inspect the input state of their containing function, or equivalently, that they all happen before any internal actions are executed.

Figure 27 shows an example of a simple Java method that executes some other methods which are presumed to have side-effects and interdependencies. The equivalent schematic table is shown to the right¹³. The columns can be imagined as "highway lanes", with time driving downward, and predicates causing lane switches. Note that this is not some special mode, but the same representation of functions that we have been using all along.

By representing time and logic orthogonally, they can be seen as separate aspects, and changed independently. Altering the conditions gating an action while preserving its relative execution order can be achieved by dragging it horizontally in the table. Conversely, altering the execution order of an action while preserving its gating conditions can be achieved by dragging it vertically in the table.

For example, suppose that we want to execute I in all cases but without changing its execution order relative to the other actions. Modifying the Java code requires some care. Figure 28(a) shows the necessary changes in bold, with deletions struck through. The equivalent change to the schematic table is performed in 28(b) by dragging the action I horizontally over to the first column, which is the condition in which we want to execute it, while keeping its vertical position unchanged, to preserve execution order. The red line is a popup drop guide. Figure 28(c) shows that the required cuts and splices have been automatically performed to achieve the desired result.

7. Decidability

The power of schematic tables is that they make conditionals declarative, providing the knowledge at compile-time and edit-time that informs all of the advanced features described in this paper. But the use of predicates like ≥ 0 may seem concerning — type systems carefully avoid such properties because they are in general undecidable. Luckily schematic tables are in a simpler situation than type systems, which must reason about the effects of arbitrary computations. Schematic tables need only reason about the partitioning properties of predicates on sets of fixed values. Such reasoning is essentially propositional, and decidable.

Formally, partitioning requires only formulas of the form $\forall x_1, \ldots, x_n.F$ where F is a quantifier-free formula of first-

order logic with equality and constants as the only function symbols. The type-membership predicate \in is axiomatized by encoding the subtype hierarchy into partitionings. The partitioning properties of integer comparison can be axiomatized using formulas of the same form, such as

$$\forall x \in Int. \ \forall y \in Int. \ (x > y) \Leftrightarrow \neg (x \le y)$$

and a finite number of instances of schemata like

$$\forall x \in Int. \ (x \ge \lceil n \rceil) \Leftrightarrow (x = \lceil n \rceil) \lor (x \ge \lceil n + 1 \rceil)$$

sufficient to cover the integer literals in use. These formulas belong to a decidable fragment of first order logic, the Bernays-Schönfinkel-Ramsey class [2], whose satisfiability can be checked in NP. There are known decision procedures [17, 23] for similar problems. Subtext will employ a SAT-solver [7] as a flexible testbed for the logic of predicates.

8. Conclusion: Representation Matters

In symbols one observes an advantage in discovery which is greatest when they express the exact nature of a thing briefly and, as it were, picture it; then indeed the labor of thought is wonderfully diminished. – Leibniz

The conclusion offered by schematic tables is that representation matters. Their flexibility comes from the extra degree of freedom offered by a two-dimensional notation. Their power comes from directly manifesting Boolean algebra in their geometry and behavior. It is a truism in all technical fields that the key to solving a problem is choosing the right notation. The hallmark of a good notation is that it mirrors its domain: that equivalent situations get represented equivalently, and that different representations refer to different situations: in other words, canonicity.

The need to manually refactor — having to laboriously make literally meaningless changes before being able to express a solution — is a textbook symptom of poor notation. Another warning sign is the need for difficult mental calculation to understand and manipulate the notation. Enormous effort and creativity has been invested into IDE's like Eclipse, yet in the end there is something gravely unhealthy about a notation that requires such a complex life-support system.

Schematic tables raise obvious concerns. Will they scale to large programs? Will the variety of editing operations be too complex? Will canonicity cause disturbing volatility as we edit? These are all serious matters of user interface (UI) research. There is some reason for optimism, however. Textual languages are anchored to a grammar, limiting IDE's to being enhanced text editors. Schematic tables, as a computer-mediated representation, make no commitment to a fixed encoding. They are not viable outside of an IDE, but they leave the door wide open for clever UI innovation.

The significance of schematic tables may not rest upon whether or not they are the one true way to represent con-

¹² This is similar to the approach taken by monads [20, 42], though without the higher-order cleverness used to accomplish it within an expression language. On the other hand, Subtext treats states as virtual versions of the entire global state.

¹³ Flow graphs [19] use similar diagrams (though with a conventional treatment of conditionals) in a categorical semantics of imperative computation.

ditionals. It may rather be as witness that there exist interesting new ways to represent programs outside of the wellexplored space of grammars. The history of programming languages is not over.

Acknowledgments

Daniel Jackson, Derek Rayside, Rob Seater, Emina Torlak, and Viktor Kuncak provided helpful discussions, and the Software Design Group at MIT provided a creative environment. Constructive criticism was offered by Jacques Carette, Matt Hellige, Macneil Shonle, and the anonymous referees. Ethan Edwards helped formulate the examples.

References

- [1] A. Blake. *Canonical expressions in Boolean algebra*. PhD thesis, University of Chicago, 1938.
- [2] E. Börger, Y. Gurevich, and E. Grädel. *The classical decision problem*. Springer, 2001.
- [3] F. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.
- [4] F. Brown. Boolean Reasoning. Kluwer Academic Publishers Boston, 1990.
- [5] M. Burnett and A. Goldberg. Visual object-oriented programming. Manning, 1995.
- [6] R. Burstall, D. MacQueen, and D. Sannella. HOPE: An experimental applicative language. *Proceedings* of the 1980 ACM conference on LISP and functional programming, pages 136–143, 1980.
- [7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7): 394–397, 1962.
- [8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18 (8):453–457, 1975.
- [9] J. Edwards. Example centric programming. SIGPLAN Not. (OOPSLA '04 Onward! Proceedings), 39(12):84– 91, 2004. ISSN 0362-1340.
- [10] J. Edwards. Subtext: Uncovering the simplicity of programming. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pages 505–518. ACM Press, 2005.
- [11] J. Edwards. First Class Copy & Paste. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, May 2006. URL http://hdl.handle.net/1721.1/32980.
- [12] M. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *ECOOP*, 98:12th, 1998.

- [13] D. L. Fisher. Data, documentation, and decision tables. *Commun. ACM*, 9(1):26–31, 1966.
- [14] T. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. *Human-Computer Interaction*, 1992.
- [15] R. Halverson Jr. An Empirical Investigation Comparing IF-THEN Rules and Decision Tables for Rrogramming Rule-based Expert Systems. System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on, 3, 1993.
- [16] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science and Engineering*, 20(1):19–35, 2005.
- [17] M. Ibramsha and V. Rajaraman. Detection of logical errors in decision table programs. *Commun. ACM*, 21 (12):1016–1025, 1978.
- [18] R. Janicki, D. L. Parnas, and J. Zucker. Tabular representations in relational rocuments. In *Relational meth*ods in computer science, pages 184–196. Springer-Verlag, 1997.
- [19] A. Jeffrey. Premonoidal categories and a graphical view of programs. Technical Report 98-004, CTI, DePaul University, August 1998. URL http://facweb.cs.depaul.edu/research/ techreports/abstract98004.htm.
- [20] S. Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreignlanguage calls in Haskell. *Engineering theories of software construction, Marktoberdorf Summer School*, pages 47–96, 2000.
- [21] P. J. H. King. Ambiguity in limited entry decision tables. *Commun. ACM*, 11(10):680–684, 1968.
- [22] H. Kirk. Use of Decision Tables in Computer Programming. Communications of the ACM, 8(1):41–43, 1965.
- [23] V. Kuncak and M. Rinard. Decision procedures for setvalued fields. *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL* 2005), 2005.
- [24] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *Software Engineering*, 26(1):15–35, 2000.
- [25] LogicGem 3.0 QuickStart. URL http: //www.catalyst.com/products/logicgem/ quickstart.pdf.
- [26] A. Löh and R. Hinze. Open data types and open functions. Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming, pages 133–144, 2006.

- [27] T. Millstein. Practical predicate dispatch. Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, pages 345–364, 2004.
- [28] D. A. Moon. Object-oriented programming with flavors. In OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications, pages 1–8, New York, NY, USA, 1986. ACM Press.
- [29] B. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages* and Computing, 1(1):97–123, 1990.
- [30] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, 1973.
- [31] J. Pane and B. Myers. Tabular and Textual Methods for Selecting Objects from a Group. *Proceedings of VL* 2000: IEEE International Symposium on Visual Languages, pages 157–164, 2000.
- [32] J. Pane and B. Myers. Usability Issues in the Design of Novice Programming Systems. Technical Report CMU-CS-96-132, School of Computer Science, Carnegie Mellon University, August 1996.
- [33] S. Pollack. Analysis of the Decision Rules in Decision Tables. Technical Report RM-3669-PR, Rand Corp, 1963.

- [34] U. Pooch. Translation of Decision Tables. ACM Computing Surveys (CSUR), 6(2):125–151, 1974.
- [35] W. Quine. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly*, 59(8): 521–531, 1952.
- [36] B. Silberg. Detab/65 in third-generation cobol. SIG-PLAN Not., 6(8):4–8, 1971.
- [37] StateStep Tutorial. URL http://statestep.com/ tutorial/RuleInitial.html.
- [38] R. F. Sterbenz. Tabsol decision table preprocessor. SIGPLAN Not., 6(8):33–40, 1971.
- [39] D. Thomas. Agile Programming: Design to Accommodate Change. *IEEE Software*, 22(3):14–16, 2005.
- [40] A. M. Turing. (1946) Proposed Electronic Calculator, report for National Physical Laboratory, Teddington. In A. M. Turing's ACE Report of 1946 and Other Papers. MIT Press, 1986.
- [41] I. Vessey and R. Weber. Structured Tools and Conditional Logic: an Empirical Investigation. *Communications of the ACM*, 29(1):48–57, 1986.
- [42] P. Wadler. Comprehending monads. Proceedings of the 1990 ACM conference on LISP and functional programming, pages 61–78, 1990.

Credit limit OK	Y	Ν	Ν	N
Pay experience favorable	-	Y	Ν	Ν
Special clearance obtained	-	-	Y	Ν
Do approve order	Х	Х	Х	
Don't approve order				Х

Figure 1. Decision table.

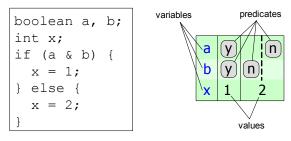


Figure 2. Schematic decision table.

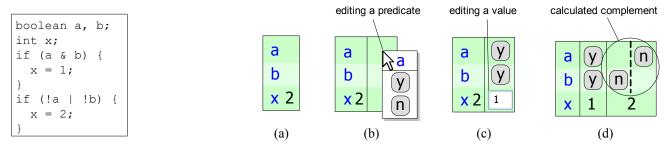
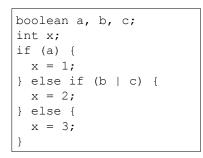


Figure 3. Java decision table.



а	y	n	\square
b		y	(\mathbf{n})
С		y	(\mathbf{n})
x	1	2	3



n

n

3

Figure 4. Logical editing.



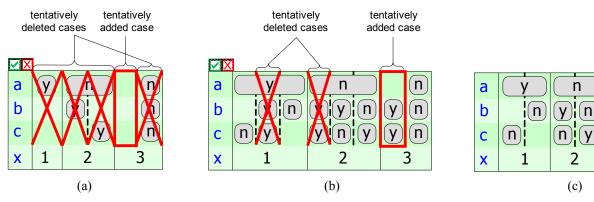


Figure 6. Transactional editing.

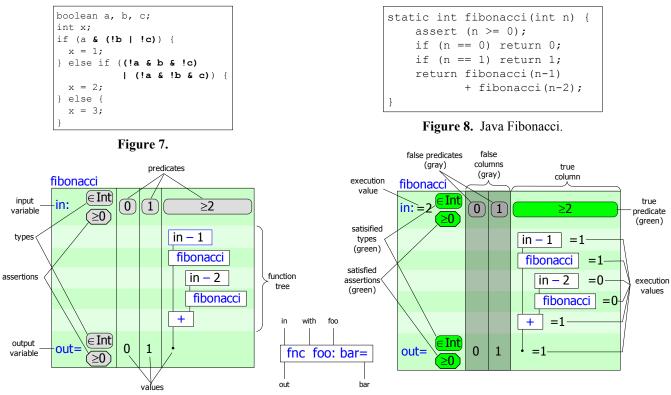
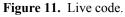


Figure 9. Schematic Fibonacci.

Figure 10. Key.



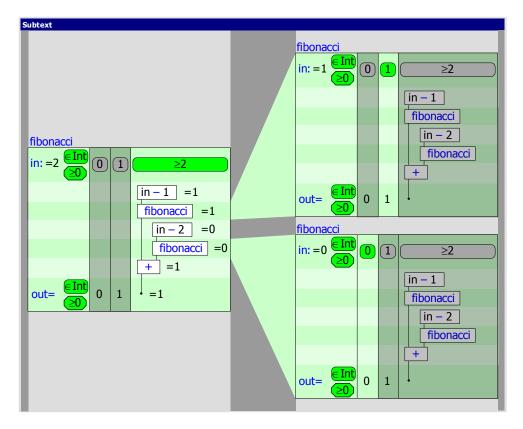


Figure 12. Laminar browsing.

```
enum Attack {MELEE, MAGICAL};
double hitPoints (Attack attack,
                 Character attacker,
                 Character target) {
    double power;
    double effectiveness;
    double hitPoints;
    switch (attack) {
      case MELEE:
          power = attacker.strength;
          break;
      case MAGICAL:
          power = attacker.magic;
          break;
    }
    effectiveness = power * (attacker.amulet ? .8 : random());
    switch (attack) {
      case MELEE:
          if (effectiveness >= target.armor) {
           hitPoints = effectiveness - target.armor;
          } else {
            hitPoints = 0;
          }
          break;
      case MAGICAL:
          hitPoints = effectiveness * (1 - target.magic);
          break;
    }
    return hitPoints;
```

Figure 13. Java hitPoints.

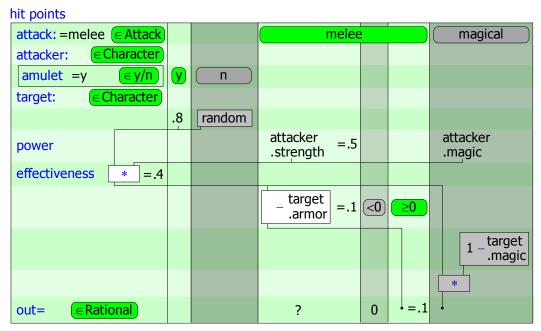
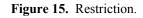


Figure 14. Schematic hit points.

hit points				
attack: EAttack			magical	restriction predicate
attacker: ∈ Character				
amulet ∈y/n	y	n		
target: (∈Character)				
	.8	random		
power			attacker .magic	
effectiveness *				
			1 – target .magic	
			*	
out= (Rational)			•	



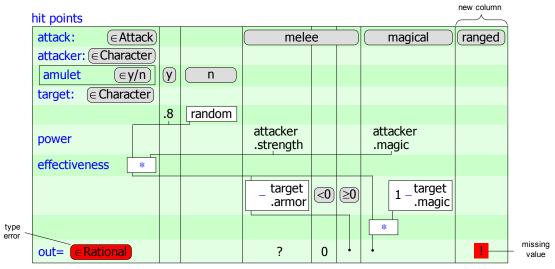


Figure 16. Adding ranged to enum.

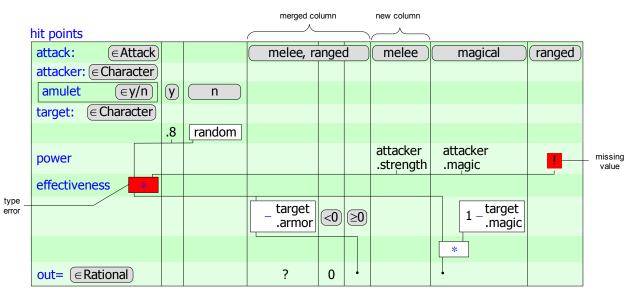


Figure 17. Merging columns.

hit points		_			_				
attack:	Attack			melee, ra	ange		melee	magical	ranged
attacker: (∈ Cha	racter								
amulet	∈y/n)	y	n						
target: (∈Char	acter								
		.8	random						
power							attacker .strength	attacker .magic	attacker .acuity
effectiveness	*								
				_ target .armor	<0	≥0		1 – target .magic	
								*	
out= (∈ Rationa	al			?	0			•	

Figure 18. Final version.

```
abstract class Attack {
    abstract double power(Character attacker);
    double effectiveness(Character attacker) {
        return power(attacker) * (attacker.amulet ? .8 : random();
    }
    abstract double hitPoints(Character attacker, Character target);
class Melee extends Attack {
    double power(Character attacker) {
        return attacker.strength;
    }
    double hitPoints(Character attacker, Character target) {
        double theEffectiveness = effectiveness(attacker);
        if (theEffectiveness >= target.armor) {
            return theEffectiveness - target.armor;
        }
        return 0;
    }
```

```
class Magical extends Attack {
   double power(Character attacker) {
      return attacker.magic;
   }
   double hitPoints(Character attacker, Character target) {
      return effectiveness(attacker) * (1 - target.magic);
   }
}
```

Figure 19. Polymorphic Java.

hit points							
attack: EAtt	ack)			∈Mele	ee	\square	€ Magical
attacker: (∈ Charac	ter						
amulet (∈y	r/n	y	n				
target: (∈Charac	ter						
		.8	random				
power				attacker .strength			attacker .magic
effectiveness	*						
				_ target .armor	<0	≥0	1 – target .magic
							*
out= (ERational)				?	0		•

Figure 20. Polymorphic table.

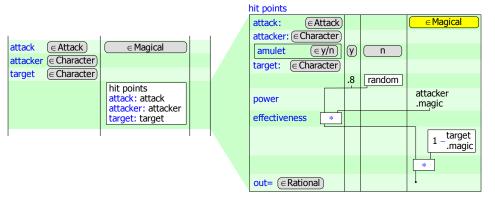


Figure 21. Contextual restriction.

```
abstract class Attack {
                                                                  class Melee extends Physical {
   abstract double power(Character attacker);
                                                                  }
   double effectiveness(Character attacker) {
       return power(attacker)
               * (attacker.amulet ? .8 : random();
                                                                  class Ranged extends Physical {
   abstract double hitPoints (Character attacker,
                                                                      double power(Character attacker) {
                              Character target);
                                                                          return attacker.acuity;
                                                                  }
abstract class Physical extends Attack {
   double power(Character attacker) {
       return attacker.strength;
                                                                  class Magical extends Attack {
   }
                                                                      double power(Character attacker) {
   double hitPoints (Character attacker,
                                                                          return attacker.magic;
                     Character target) {
                                                                      }
        double theEffectiveness = effectiveness(attacker);
                                                                      double hitPoints (Character attacker,
        if (theEffectiveness >= target.armor) {
                                                                                       Character target) {
           return theEffectiveness - target.armor;
                                                                          return effectiveness(attacker)
        }
                                                                                 * (1 - target.magic);
        return 0;
                                                                      }
   }
                                                                  }
```

Figure 22. Restructured classes.

hit points							
attack: EAttack				\in Physical	\in Magical		
			∉ Melee	∈Mel	ee	\square	
attacker: ∈ Character							
amulet $(\in y/n)$	y	n					
target: (∈ Character)							
	.8	random					
power				attacker .strength			attacker .magic
effectiveness *							
				_ target _armor	<0	≥0	1 - target .magic
out= ∈ Rational			!	?	0	•	•

Figure 23. Result of restructuring classes.

hit points							
attack: $\in A$	ttack			(∈ Physical)			∈Magical
attacker: (∈ Char	acter						
amulet	y/n	Y	n				
target: (∈Char	acter						
		.8	random				
power				attacker .strength			attacker .magic
effectiveness	*						
				target armor	<0	≥0	1 – target .magic
							*
out= (∈ Rationa	D			?	0		

Figure 24. Merging cases.

hit points					re	striction predicat	e
attack: (EAttack)			∈Physi	ical	\Box	∈ Ranged	
attacker: (∈ Character)							
amulet $(\in y/n)$	y	n					
target: ECharacter							overriding
	.8	random					edit
power			attacker .strength			attacker.	acuity
effectiveness *							
			_ target .armor	<0	≥0		
out= (Rational)			?	0	·		

Figure 25. Overriding from a restriction.

hit points							
attack: EAttac	k		E Physical				
						∉Ranged	∈Ranged
attacker: ∈ Characte	r						
amulet (∈y/n		n					
target: (∈Character	อ						
	.8	random					
power						attacker .strength	attacker .acuity
effectiveness *							
			target armor	<0	≥0)		
out= (∈ Rational)			?	0			

Figure 26. Restriction on superclass.

void F(boolean a,]	F	
		in:	
boolean b) {			
G();		a: (∈y/n)	y n
if (a) {		b: (∈y/n)	y n
H();		in C	
if (b) {		in G	
II (0) (I();			H
J();			
}			
,			
}			
}		out=	

Figure 27. Untangling time and logic.

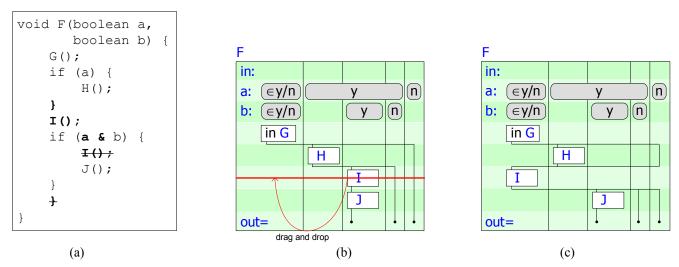


Figure 28. Changing logic while preserving execution order.